Logo

Description automatically generated

Group Assignment (50%)

|  |  |  |
| --- | --- | --- |
| **Module Code** | **:** | CT065-3-3-ALG-L-5 (Algorithmics) |
| **Intake Code** | **:** | APD3F2411CS(CYB) |
| **Lecturer Name** | **:** | DR. VAZEERUDEEN ABDUL HAMEED |
| **Date Assigned** | **:** | 1st Jan |
| **Date Submitted** | **:** | 10th March |

|  |  |
| --- | --- |
| **Student TP number** | **Full Name** |
| TP071012 | ABDULRAHMAN GAMIL MOHAMMED AHMED |
| TP072306 | HAZIQ IRFAN BIN RADZALI |
| TP069210 | YAZEN ABOBAKR AHMED AL-MEHDHAR |

Contents

[1. PART A: Algorithms and Complexity 3](#_Toc191017675)

[1.1 Introduction to the Problem 3](#_Toc191017676)

[1.2 Algorithm 1: Bubble Sort 4](#_Toc191017677)

[1.2.1 How Bubble Sort Works 4](#_Toc191017678)

[1.2.2 Time Complexity of Bubble Sort 6](#_Toc191017679)

[1.2.3 Practical Implications 7](#_Toc191017680)

[1.3 Algorithm 2: Quick Sort 8](#_Toc191017681)

[1.3.1 How Quick Sort Works 8](#_Toc191017682)

[1.3.2 Time Complexity of Quick Sort 10](#_Toc191017683)

[1.3.3 Practical Implications 11](#_Toc191017684)

[1.4 Suitability Comparison 12](#_Toc191017685)

[2. PART B: Complexity Classification 13](#_Toc191017686)

[2.1 Introduction 13](#_Toc191017687)

[2.2 P (Deterministic Polynomial Time) Problems 13](#_Toc191017688)

[2.2.1 Definition and Concept 13](#_Toc191017689)

[2.2.2 Characteristics 13](#_Toc191017690)

[2.2.3 Example: Linear Search 14](#_Toc191017691)

[2.2.4 Significance 14](#_Toc191017692)

[2.3 NP (Non-deterministic Polynomial Time) Problem 15](#_Toc191017693)

[2.3.1 Definition and Concept 15](#_Toc191017694)

[2.3.2 Example: Traveling Salesman Problem (TSP) 15](#_Toc191017695)

[2.4 Discussion 16](#_Toc191017696)

[3 Conclusion 17](#_Toc191017697)

Table of Figures

[Figure 1: Bubble Sort PSEUDOCODE 5](#_Toc191018201)

[Figure 2: Quick Sort PSEUDOCODE 9](#_Toc191018202)

# 1. PART A: Algorithms and Complexity

## Introduction to the Problem

Sorting a list of numbers is one of the most fundamental computer science issues, underlying a wide range of applications that rely on ordered data. From transforming an unsorted list like [8, 4, 1, 7, 3] into a neat [1, 3, 4, 7, 8] for a database query, to preparing data to be searched quickly, to displaying ranked results in a user interface, sorting is a fundamental task. The efficiency of a sorting algorithm quantified in its time complexity determines its viability, especially as the input size, **n**, varies from small lists with a few elements to big data with millions of items. Time complexity computes how the algorithm's runtime increases with **n**, providing a rationale for assessing its performance under various circumstances.

In this assignment, will compare two distinct algorithms for resolving the sorting problem: **Bubble Sort**, a simple and intuitive approach, and **Quick Sort**, a sophisticated and highly efficient solution. In the following, we explain their functionality, compare their time complexity in best-case, worst-case, and average-case scenarios, and establish their suitability to different circumstances by considering factors such as data size, requirement for processing speed, and implementation constraints.

## Algorithm 1: Bubble Sort

**Bubble Sort** is one of the simplest sorting algorithms, commonly presented to beginners ‎due to its simple concept and simplicity. It does this by repeatedly scanning the list, comparing two ‎neighboring items, and swapping them if they are out of order, pushing larger values to the end of the array in a manner akin to bubbles rising to the surface of a liquid thus its name (GeeksforGeeks, 2014).

### 1.2.1 How Bubble Sort Works

Let's consider the array [8, 4, 1, 7, 3]. The algorithm proceeds as follows in its first pass:

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **Condition** | **Action** | **Array** |
| - Compare 8 and 4 | 8 > 4 | swap them | [4, 8, 1, 7, 3] |
| - Compare 8 and 1 | 8 > 1 | swap them | [4, 1, 8, 7, 3] |
| - Compare 8 and 7 | 8 > 7 | swap them | [4, 1, 7, 8, 3] |
| - Compare 8 and 3 | 8 > 3 | swap them | [4, 1, 7, 3, 8] |

After this pass, the largest item, 8, is moved to its position at the end. The second pass deals with the rest of the sub-array [4, 1, 7, 3]:

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **Condition** | **Action** | **Array** |
| Compare 4 and 1 | 4 > 1 | swap them | [1, 4, 7, 3] |
| Compare 4 and 7 | 4 < 7 | no swap | [1, 4, 7, 3] |
| Compare 7 and 3 | 7 > 3 | swap them | [1, 4, 3, 7] |

The third pass sorts [1, 4, 3]:

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **Condition** | **Action** | **Array** |
| Compare 1 and 4 | 1 < 4 | no swap | [1, 4, 3] |
| Compare 4 and 3 | 4 > 3 | swap them | [1, 31, 4] |

The fourth iteration considers [1, 3, 4], finds no swaps needed, and the array is sorted fully as [1, 3, 4, 7, 8]. This step-by-step process indicates Bubble Sort's step-by-step nature.

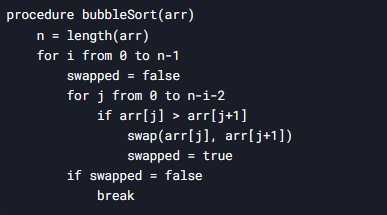


Figure 1: Bubble Sort Pseudocode

**Explanation of the Optimization:**

**Swapped Flag:**

A boolean flag (swapped) is included to track whether any swaps occur in a pass.

If no swaps are performed in a full pass through the array, then the array is sorted, and the algorithm can be stopped early.

**Early Termination:**

The if swapped = false condition checks if no swaps were performed in the inner loop.

If there are no exchanges, the break statement exits from the outer loop without redundant iterations.

**Efficiency Improvement:**

Under the best circumstances (e.g., when the input array is already sorted), the algorithm operates in O(n) time instead of O(n²).

This optimization reduces comparisons and exchanges by a huge margin for partially or fully sorted arrays.

### 1.2.2 Time Complexity of Bubble Sort

Time complexity is a measure of the efficiency of an algorithm in terms of execution time ‎as a function of input size **n** (*Khan Academy*, 2023). Bubble ‎Sort has greatly differing performance in different scenarios:‎

|  |
| --- |
| **Best Case: O(n)** |
| If the array is already sorted, e.g., [1, 2, 3, 4, 5], the improved version performs ‎**n-1** comparisons in the initial pass (4 comparisons for **n = 5**), finds no swaps needed, and stops. This provides us with a time complexity of O(n) for linear. For ‎‎**n = 100**, only 99 operations are needed a fast sprint. Without optimization, Bubble Sort completes all **n-1** passes in either case, costing O(n²), but ‎we look at the optimized algorithm here to note its potential efficiency in optimal conditions.‎ |

|  |
| --- |
| **Worst Case: O(n²)** |
| A reverse-ordered list, [5, 4, 3, 2, 1], is the most work. The first pass ‎takes 4 swaps: [4, 3, 2, 1, 5]. The second pass: 3 swaps to [3, 2, 1, 4, 5]. The ‎third: 2 swaps to [2, 1, 3, 4, 5]. The fourth: 1 swap to [1, 2, 3, 4, 5]. The total ‎number of comparisons is the sum of the first **n-1** integers: **(n-1) + (n-2) +. + 1 = n(n-1)/2**  For **n = 1000**, it will be 499,500 comparisons and the same number of swaps in the worst case, which verifies O(n²) complexity. For **n = 10,000**, it will be approximately 50,000,000 operations, indicating its quadratic growth. |

|  |
| --- |
| **Average Case: O(n²)** |
| In an array randomly sorted, like [3, 6, 1, 5, 2], half of the comparisons are indeed swaps, yet the nested loop design still consumes about ‎**n²/4** steps. For **n = 5000**, that's about 6.25 million steps, and O(n²) complexity. This bias on most inputs ‏hides Bubble Sort's inherent inefficiency at large **n**. |

### 1.2.3 Practical Implications

Bubble Sort's benefit is that it is straightforward and has minimal overhead. It takes only O(1) ‎additional space (for the temporary exchange), which makes it attractive for memory-constrained systems, i.e., embedded devices or microcontrollers. For small inputs, like ‎**n = ‎‎20**, its worst-case 190 operations on modern hardware take microseconds, which makes it suitable for sorting a short list of student grades or a small inventory catalog. However, its O(n²) time requirement is a crippling limitation as data size grows. Sorting 100,000 objects takes around 5 billion steps, consuming minutes or hours on a standard processor far too long for applications like sorting bank system transaction records or ordering huge datasets in a scientific study. While its ease of use makes it a valuable educational tool, its practical use is only available in circumstances where performance is not essential.

## Algorithm 2: Quick Sort

Quick Sort, invented by Tony Hoare in 1959, is a divide-and-conquer algorithm ‎widely renowned for its performance and popularity. It works by selecting a pivot ‎element, partitioning the array around it, and recursively sorting the resulting sub-‎-arrays, a stark difference from Bubble Sort's iterative approach and surpassing it in ‎average-case performance (GeeksforGeeks, 2014).‎

### How Quick Sort Works

Consider the list [8, 4, 1, 7, 3] with pivot 8. Partitioning step ‎shuffles the elements: elements smaller than 8 (4, 1, 7, 3) get moved to the left, and there are no elements larger ‎here, resulting in [4, 1, 7, 3, 8].

|  |  |  |
| --- | --- | --- |
| **Step** | **Action** | **Outcome** |
| **Initialization** | Pivot eight partitions [8, 4, 1, 7, 3] into [4, 1, 7, 3, 8]. | [4, 1, 7, 3, 8] |
| **Recursive Sort** | Sorting sub array [4, 1, 7, 3] with pivot four. | [1, 3, 4, 7] |
| **Recursive Sort** | Sorting sub array [1, 3] with pivot one. | [1, 3] |
| **Updated sorted array** | Combine the result to [1, 3, 4, 7, 8] | [1, 3, 4, 7, 8] |

A computer screen shot of white text

AI-generated content may be incorrect.

Figure 2: Quick Sort PSEUDOCODE

**quickSort Procedure**

**1- Base Condition:**

If low < high, the sub-array has more than one element and must be sorted.

Else, the sub-array is already sorted.

**2- Partitioning:**

Call the partition method to divide the array into two sub-arrays:

* Left sub-array: Elements ≤ pivot.
* Right sub-array: Elements > pivot.

**3- Recursive Calls:**

Recursively call the quickSort method for the left sub-array (quickSort(arr, low, pivotIndex - 1)).

Recursively call the quickSort method for the right sub-array (quickSort(arr, pivotIndex + 1, high)).

**partition Procedure**

**1- Pivot Selection:**

The pivot is chosen as the last item of the sub-array (arr[high]).

**2- Partitioning Logic:**

Initialize i = low - 1. This is the position of the last item in the left sub-array.

Iterate through the sub-array by index j:

If arr[j] <= pivot, then increment i and swap arr[i] and arr[j].

After the loop, put the pivot at its rightful position by swapping arr[i + 1] with arr[high].

**3- Return Pivot Index:**

The pivot is in its final sorted location now and then it will send the pivot index (i + 1) back to divide the array for recursive sort.

### 1.3.2 Time Complexity of Quick Sort

‎

|  |
| --- |
| **Best Case: O(n log n)** |
| When pivots consistently partition the array into nearly equal halves, recursion depth is logarithmic. In case of **n = 1024**, the array is halved at each level (1024, 512, 256, ‎‎., 1), and it requires log₂ 1024 = 10 levels. Every level performs **n** comparisons, totaling 1024  **10 = 10,240** operations O(n log n). In case of **n = 1,000,000** , log₂ ‎‎1,000,000 ≈ 20, and it will take around 20 million steps, showcasing its ‎scalability.‎ |

|  |
| --- |
| **Worst Case: O(n²)** |
| When the pivot is always the smallest or the largest number, like [1, 2, 3, 4, 5] with ‎pivot 5, partitioning results in [1, 2, 3, 4] and an empty sub-array, occurring **n-1** ‎times. The total comparisons sum to:  ‎ \[‎  ‎ (n-1) + (n-2) +. + 1 = \frac{n(n-1)}{2}‎  ‎ ]]  ‎For **n = 1000**, this is approximately 499,500 operations O(n²). This occurs with already ‎sorted or reverse-sorted arrays with a poor pivot strategy, but techniques like ‎randomization mitigate this likelihood in practice‎.  ‏‏Partitions are evenly balanced with random pivot selection, with an average of log **n** ‏levels. When **n = 10,000**, log₂ 10,000 ≈ 13.3, which means approximately 133,000 |

|  |
| --- |
| **Average Case: O(n log n)** |
| Operations that's a small percentage of Bubble Sort's millions for the same quantity. This O(n log n) ‏average efficiency makes Quick Sort highly efficient on a majority of real-world inputs. |

### 1.3.3 Practical Implications

Quick Sort's O(n log n) average-case performance is brilliant with big sets. Sorting 1 million items takes approximately 20 million operations, completing in a few seconds on modern hardware, compared to Bubble Sort's billions, which would take hours. Its in-place partitioning minimizes memory usage beyond the O(log n) stack space needed for recursion, which makes it perfect for systems with average resources. Although its worst-case O(n²) is a theoretical limitation, real-world implementations ‎‎(e.g., randomized pivots) make it an exceptional case, guaranteeing reliability. Quick Sort is the basis for standard libraries and drives applications such as sorting search results or database ‎records, where speed and scalability are paramount.

## Suitability Comparison

The choice between Quick Sort and Bubble Sort relies on some features:

‎**Small Data (n < 50):**

Bubble Sort is a choice for extremely small inputs. For \*n = 50\*, its worst-case 1225 operations are comparable in size to Quick Sort's roughly 282 (50 \* log₂ 50 ≈ 5.64 \* 50). Its simplicity and non-recursivity also recommend it for sorting a small list of names or scores in a classroom setting where setup simplicity outweighs performance gains.

‎**Medium Data (n = 1000)**:

Quick Sort gains the lead. Bubble Sort's 499,500 ‎operations dominate Quick Sort's ~10,000 (1000 \* log₂ 1000 ≈ 10), so it's preferable ‎to employ for sorting user profiles in a mid-sized social network or products in an online ‎store catalog.

‎**Big Data (n > 10,000)**:

Quick Sort performs by far. For \*n = ‎‎100,000\*, Bubble Sort's 5 billion operations are contrasted with Quick Sort's 1.7 million ‎‎(100,000 \* log₂ 100,000 ≈ 17)—a 3000-fold reduction—perfect for sorting server ‎logs, genomic sequences, or big transaction data within real-time systems.

**Desired Speed of Processing**:

Programs that have stringent timing requirements, such as live data dashboards or trading platforms in finance, demand Quick Sort's speedy average-case performance, while Bubble Sort is sufficient for non-time-critical applications such as a one-off sort in a prototype script.

**Implementation Constraints**:

Bubble Sort's O(1) space complexity favors it in memory-limited environments like small embedded systems with small \*n\*, whereas Quick Sort's performance and flexibility make it the default option for general-purpose machines with ample resources.

Bubble Sort and Quick Sort offer polar opposite solutions to the sorting problem. Bubble Sort's O(n²) complexity restricts it to small, non-performance-critical applications where its simplicity of implementation and minimal memory usage are advantages. Quick Sort's O(n log n) average case performance, underpinned by practical optimizations, makes it the optimal choice for large-scale, performance-critical applications that drive modern computing. Between them, the preference is determined by a balancing act between data quantity, processing rate needs, and system ‎constraints, and Quick Sort often proving to be the functional norm in most ‎practical applications outside academic settings.

# 2. PART B: Complexity Classification

## 2.1 Introduction

Complexity classification in computer science is the division of problems according to ‎the resources, particularly time to calculate them, providing a context for measuring the computational ‎viable. There are two core classes, i.e., **P (Deterministic Polynomial time)** and **NP (Non-deterministic Polynomial time)**, and they differ according to the ease with which solutions can be calculated and verified. P covers problems that are solvable quickly and in a deterministic way, while NP covers problems where solutions are easily verifiable quickly, even though their determination is not always quick. This section explains P using an example, followed by the identification and explanation of an NP problem, highlighting their significance in algorithm design and theoretical computation (*Automata, Computability, and Complexity | Electrical Engineering and Computer Science | MIT OpenCourseWare*, 2025).

## 2.2 P (Deterministic Polynomial Time) Problems

### 2.2.1 Definition and Concept

P, or Deterministic Polynomial time, is the collection of problems which can be solved by a deterministic Turing machine, a theoretical computer model that goes through a definite set of operations, in polynomial time. Polynomial time is described as O(n^k), where **n** is input size and **k** is a constant, for example, O(n), O(n²), or O(n³). Such problems are "tractable" as their runtimes grow ‎at an acceptable pace, remaining manageable even for sufficiently large values of **n**. In contrast with exponential-time problems (e.g., O(2^n)), which are unmanageable with large inputs, P problems are compatible with contemporary computers and form the foundation of efficient ‎computation (Dean, 2015).

### 2.2.2 Characteristics

P algorithms are deterministic, that is, they are fully predetermined with no guesswork or randomness. Such determinism ensures runtime is polynomial with performance predictable. An O(n²) algorithm, for example, has 1 million steps for **n = 1000**, doubling **n** to 2000 increases the steps to four times as many, 4 million enormous but still tractable—while an O(2^n) algorithm jumps from 2^1000 to 2^2000, an unimaginably huge number outside human reach. ‎This planned growth characterizes P's practicality and efficiency.‎

### 2.2.3 Example: Linear Search

‎

**Problem Description**:

For an unsorted list, such as [4, 7, 2, 9, 5], and a ‎target value, such as 9, give a boolean indication if the target is in the list.‎

**Algorithm**:

Linear Search searches every element in order: 4 (no match), ‎‎7 (no), 2 (no), 9 (match) stopping upon finding the target. In searching for 6 (not ‎present), it checks all elements: 4, 7, 2, 9, 5, stopping after **n** steps that 6 is ‎missing (GeeksforGeeks, 2016).

**Time Complexity**:

O(n). Worst case target at end or absent triggers **n** comparisons. For **n = 10,000**, it's 10,000 steps, a linear progression which is still polynomial. Even for **n = 100,000**, it's 100,000 steps, runable in under a second on modern hardware.

‎**Why it's in P**:

Linear Search is deterministic: it has a single definite, cut-and-dried ‎path, looking at each item once in order. With [1, 2, 3, 4, 5] and target 5, ‎it takes 5 steps; with target 6, 5 steps too always O(n). No shortcuts or ‎guesswork are required, which fits well with P's constraints.‎

### 2.2.4 Significance

Linear Search is an example of P's determinism and ease. It is convenient to use small or unsorted data, e.g., finding a specific entry in a brief log file or a name in an unsorted list of participants. Although better optimal algorithms such as binary search (O(log n), also in P) exist for sorted data, Linear Search's O(n) promise makes it still a candidate solution when preprocessing (e.g., sorting) is not feasible. Its existence in P underscores the application of the class to everyday computation, where yeah, polynomial-time algorithms drive the tedious work of data access and elementary processing (GeeksforGeeks, 2016).

## 2.3 NP (Non-deterministic Polynomial Time) Problem

### 2.3.1 Definition and Concept

NP, Non-deterministic Polynomial time, has problems in which a proposed solution can be **verified** in polynomial time by a deterministic Turing machine, even though it may take much longer to come up with that solution maybe exponential. A non-deterministic Turing machine could in theory "**guess**" an answer right away and then verify it rapidly in polynomial time. NP has all P ‎problems (as polynomial-time solvable problems can be verified in polynomial ‎time) but also some harder problems, some of which are NP-‎complete some of the most difficult in NP. The difference between solving and verifying ‎is at the heart of NP's definition and its theoretical significance (Dean, 2015).

### 2.3.2 Example: Traveling Salesman Problem (TSP)

**Problem Description**:

With **n** cities let's say, A, B, C, D and distances between them (e.g., A-B: 4, B-C: ‎‎5, C-D: 6, D-A: 7) find the shortest possible tour that goes to each city once ‎and back to the beginning (a Hamiltonian cycle). For example, is A → B → C ‎‎→ D → A (total distance 22) the shortest, or is there a better route?‎ (GeeksforGeeks, 2013)

**Verification in Polynomial Time**:

**‎ If a path is suggested:** A → B → D → C → A. Checking takes two ‎steps: ‎

‎ **1. Ensure it goes through all cities once:** A, B, D, C, back to A—4 cities, O(n) ‎time to look. ‎

**‎ 2. Calculate the total distance: 4** (A-B) + 8 (B-D) + 6 (C-D) + 10 (D-A) = 28, also ‎O(n). ‎

‎ In the case of **n = 50**, it would take about 50 steps to check the route's validity and calculate ‎its length—self-evidently polynomial, O(n). TSP is therefore an NP problem because verification is efficient.

**Solving the Problem**:

Finding the shortest tour is much harder. The naïve approach examines every ‎possible tour, which are (n-1)! since the home city is fixed. For **n = 5**, this is 4! = 24 tours; for **n = 10** , 9! = 362,880; for **n = 20**, 19! ≈ 1.2 × 10¹⁷. Time complexity is O(n!), exponential and too expensive for large ‎‎**n**. For **n = 100**, the number of tours is larger than the atoms in the observable ‎universe, so exhaustive search is not possible. There is no deterministic polynomial-time algorithm known for TSP in general form, which makes it distinguishable from P problems.

## 2.4 Discussion

‎**NP-Completeness**:

TSP is NP-complete, i.e., as hard as the hardest ‎problems in NP. It can be reduced to other NP problems (e.g., the Hamiltonian Cycle ‎problem), and a polynomial-time solution would imply P = NP a huge, unsolved problem in computer science since 1971. Everyone believes P ≠ NP, so TSP's intractability persists.

**Practical Solutions**:

Due to its complexity, practical solutions in the real world rely on ‎approximations or heuristics. The nearest neighbor heuristic (start at a city, go to the ‎nearest unvisited city, repeat) has O(n²), resulting in a path rapidly but not necessarily ‎the shortest. For example, in our 4-city problem, starting at A might lead to A → B → C ‎‎→ D → A (22), failing to catch a better one. There are also other methods, like the Christofides ‎algorithm, that guarantee a 50% of optimal solution in polynomial time, ‎compromising speed for quality.‎

‎**Applications**:

TSP occurs in logistics (shortest delivery routes for companies like FedEx), circuit board design (minimizing lengths of wire), and bioinformatics (assembling DNA fragments). For small \*n\* (e.g., 5 cities), brute force is possible, but big cases (e.g., 50 cities) use these approximations, as exact solutions are infeasible to compute.

**Theoretical Implications**:

TSP embodies the nature of NP: it's simple to verify a solution but not always simple to obtain efficiently. This disconnect motivates research into various computation models (e.g., quantum computation) and approximation algorithms, pushing the boundaries of what can be computed.

# 3 Conclusion

In conclusion, the choice between **Bubble Sort** and **Quick Sort** helps to highlight the importance of selecting the right algorithm based on the size and performance requirements of the problem. Bubble Sort, with its simplicity and memory efficiency, is ideal for small, non-critical applications, while the performance and scalability of Quick Sort make it the default option for large-scale, performance-critical applications. Similarly, the distinction between **P** and **NP** problems underscores the broader challenges in computational complexity, where no efficient solutions to certain problems have been discovered, driving ongoing search and innovation in computer science. Together, these concepts emphasize the value of pragmatism, efficiency, and theoretical understanding in addressing real-world computational issues.

# 4 References

* GeeksforGeeks. (2012, November 15). *Introduction to Divide and Conquer Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm/

‌

* *Automata, Computability, and Complexity | Electrical Engineering and Computer Science | MIT OpenCourseWare*. (2025). MIT OpenCourseWare. https://ocw.mit.edu/courses/6-045j-automata-computability-and-complexity-spring-2011/
* GeeksforGeeks. (2016, October 20). *Linear Search Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/linear-search/

‌

* GeeksforGeeks. (2013, November 3). *Travelling Salesman Problem using Dynamic Programming*. GeeksforGeeks. https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/

‌

* Dean, W. (2015, July 27). *Computational Complexity Theory*. Stanford.edu. https://plato.stanford.edu/entries/computational-complexity/

‌

* *Khan Academy*. (2023). Khanacademy.org. https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation

‌

* GeeksforGeeks. (2014, January 7). *Quick Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/quick-sort-algorithm/

‌

* GeeksforGeeks. (2014, February 2). *Bubble Sort Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/bubble-sort-algorithm/

‌