DESIGN ANALYSIS AND ALGRITHMS

A Comprehensive Notes

COMPLEXITY ANALYSIS

Complexity Theory is a branch of theoretical computer science that focuses on:

"Classifying computational problems based on their inherent difficulty and the resources required to solve them."

It helps us understand:

- Which problems can be solved efficiently.
- Which problems are hard or unsolvable.
- The best possible algorithms and their limitations.

In **Complexity Theory**, we typically talk about two main types:

---

1. Algorithmic Complexity

(Also called **Time & Space Complexity**)

This focuses on how much **time** or **memory** an algorithm uses, depending on the size of its input (n).

Types:

- **Time Complexity**: How the number of steps grows with input size. Example: O(n), O(n log n), etc.
- **Space Complexity**: How much memory is required.

Example:

For a linear search:

- Time: O(n)
- Space: O(1) (constant space)

---

2. Computational Complexity

This is more theoretical. It classifies **problems**, not algorithms, based on the minimum resources needed to solve them.

Focuses on:

- **Problem hardness**
- **Complexity classes** like P, NP, NP-Complete, NP-Hard, etc.

Example:

- Sorting is in class **P** (easy to solve).
- The **Travelling Salesman Problem** is **NP-Hard** (hard to solve optimally).

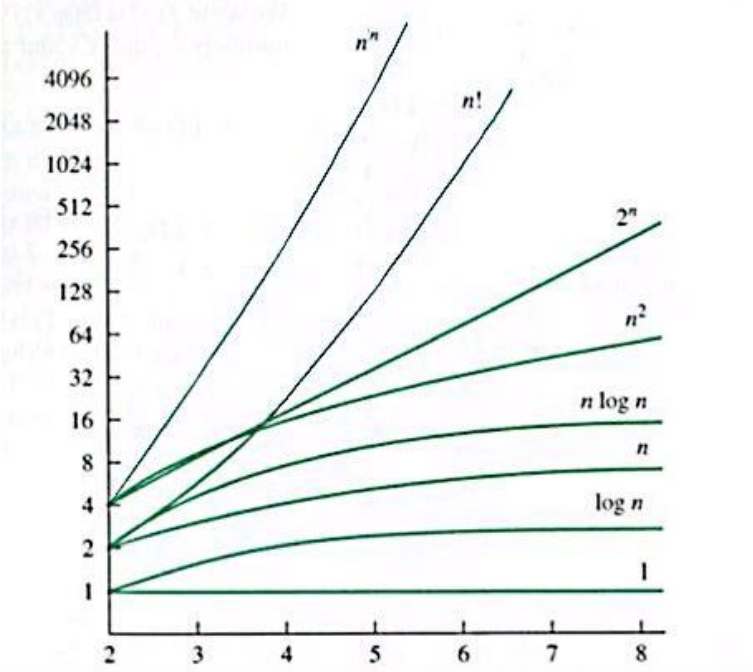
Asymptotic Notations in Complexity Analysis*:*

Asymptotic notations are **mathematical tools** used to describe the **efficiency of an algorithm** in terms of **input size (n)**, especially for **large inputs**.

They help analyze **Best-case, Average-case**, **Worst-case** performance.

Asymptotic Growth Rate:

The **asymptotic growth rate** describes how the running time or space requirements of an algorithm increase as the size of the input increases. It provides a way to classify algorithms based on their performance in terms of input size, usually for large inputs. The concept is important in **complexity analysis** because it helps to compare the efficiency of different algorithms by focusing on their behavior when the input size tends to infinity.



📌 Types of Asymptotic Notations:

---

1. Big-O Notation (O) – *Upper Bound*

It represents the **worst-case** time complexity.
Describes the **maximum time** an algorithm can take.

- 🔹 **Notation:**

f(n) = O(g(n))
Means: f(n) grows **at most as fast as** g(n)

✅ **Example:**

For Bubble Sort → O(n^2)

---

2. Omega Notation (Ω) – *Lower Bound*

It represents the **best-case** time complexity.
Describes the **minimum time** an algorithm will take.

- 🔹 **Notation:**

f(n) = Ω(g(n))
Means: f(n) grows **at least as fast as** g(n)

✅ **Example:**

Best case of Bubble Sort (already sorted) → Ω(n)

---

3. Theta Notation (Θ) – *Tight Bound*

It represents the **average-case** or **exact bound**.
Describes when time is **both upper and lower bounded**.

- 🔹 **Notation:**

f(n) = Θ(g(n))
Means: f(n) grows **exactly as fast as** g(n)

✅ **Example:**

Merge Sort → Θ(n log n)

Deterministic and non-deterministic algorithms

Deterministic Algorithm:

A **deterministic algorithm** is an algorithm that, for a given input, **always produces the same output** and follows a **single, predictable sequence of steps** during its execution.

It does not involve any randomness or guessing; each step is clearly defined and uniquely determined.

Non-Deterministic Algorithm:

A **non-deterministic algorithm** is an abstract algorithm that can, at each step, **choose from multiple possibilities** and may follow **many different paths** simultaneously to find a solution.

It is mainly used in **theoretical computer science** and assumes the ability to "guess" the correct solution path instantly.

Aspect	**Deterministic Algorithm**	**Non-Deterministic Algorithm**
Definition	Follows a single, definite sequence of steps for a given input	May follow multiple paths at the same time or make "guesses"
Output	Always gives the same output for the same input	May produce different outputs theoretically for the same input
Execution Path	Only **one path** of execution	**Multiple possible** execution paths
Predictability	Fully predictable and repeatable	Not predictable in execution (in theory)

Aspect	Deterministic Algorithm	Non-Deterministic Algorithm
Usage	Used in **real-world programming** and practical algorithms	Used in **theoretical models**, like Non-deterministic Turing Machines
Speed	Time taken depends on step-by-step logic	Can "guess" correct solutions instantly (theoretically faster)
Example Problems	Binary Search, Sorting, Graph traversal	NP-Complete problems like Travelling Salesman Problem, SAT problem
Real-world existence	Yes, used in daily computing	No, exists only in theory (helps define complexity classes like NP)

🔷 Decision Problems

A **decision problem** is a problem with a **yes/no (boolean)** answer.

- Often used in **complexity theory** and **NP-completeness**
- Easier to analyze mathematically

✅ **Example:**

- Is there a path from A to B in the graph with total weight less than 10?
- Is this number a prime?
- Whether a given graph can be coloured using 4 Colours?

🔷 Optimization Problems

An **optimization problem** is a problem where the goal is to **maximize or minimize** a certain value.

- Requires finding the **best solution** among many valid ones
- Usually more complex than decision problems

✅ **Example:**

- “What is the **shortest path** from A to B?”
- “What is the **maximum profit** from a set of jobs with deadlines?”
- Finding min no of colours to colour a graph.

P and NP Class Problems

P Class (Polynomial Time)

The class **P** consists of problems that can be **solved by a deterministic algorithm** in **polynomial time**, i.e., O(n), O(n^2), etc.

☑ **Key Points:**

- **P = "Easy to solve".**
- These problems are called **tractable.**
- The solution can be **found and verified** in polynomial time.
- Algorithm exists to solve the problem efficiently

📌 **Example Problems in P:**

- Binary Search → O(log n)
- Merge Sort → O(n log n)
- Dijkstra’s Algorithm → O(V^2) or better

NP Class (Non-deterministic Polynomial Time)

The class **NP** consists of problems whose solutions can be **verified** in **polynomial time**, even if **finding** the solution might be hard.

☑ **Key Points:**

- **NP = "Easy to verify"**

- Solutions are **easy to verify**, but may be **hard to find**.
- If a solution is given, we can check its correctness in polynomial time

📌 **Example Problems in NP:**

- Sudoku (given a filled board, easy to check)
- Travelling Salesman Problem (TSP)
- Boolean Satisfiability Problem (SAT)

	P class	**NP class Problem**
1.	P problems are a set of problems that can be solved in polynomial time by deterministic algorithms.	NP problems are problems that can be solved in nondeterministic polynomial time.
2.	P Problems can be solved and verified in polynomial time.	The solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time.
3.	P problems are a subset of NP problems.	NP Problems are a superset of P problems.
4.	All P problems are deterministic in nature.	All the NP problems are non-deterministic in nature.
5.	It takes polynomial time to solve a problem like n, n^2, n*logn, etc.	It takes non-deterministic polynomial time to quickly check a problem.
6.	The solution to P class problems is easy to find.	The solution to NP class problems is hard to find.
7.	**Examples of P problems are:** Selection sort, Linear Search	**Examples of NP problems are** the Travelling salesman

		problem and the knapsack problem

NP Hard and NP Complete Class Problems

NP-Hard:

An NP-Hard problem is a problem that is at least as hard as the hardest problems in NP. In other words, an NP-Hard problem is one that is at least as difficult to solve as the hardest problems in NP.

- At least hard as NP.
- Sol may or may not be verifiable in Polynomial Time.
- Example: TSP, Chess

NP-Complete:

An **NP-Complete** problem is a problem that is both:

1. In **NP** (the solution can be verified in polynomial time).
2. **NP-Hard** (it is at least as hard as any problem in NP, i.e., every problem in NP can be reduced to it in polynomial time).

Special problems of NP class where, every problems in NP can be reduced to them in polynomial time.

Example: Vertex cover, Hamiltonian Cycle, etc.

Property	**NP-Hard**	**NP-Complete**
Definition	Problems that are at least as hard as NP problems.	Problems that are both in NP and NP-Hard.
Belongs to NP?	**No** (NP-Hard problems may not be in NP).	**Yes** (NP-Complete problems are decision problems in NP).

Property	NP-Hard	NP-Complete
Verification	**May not be verifiable in polynomial time.**	**Verifiable in polynomial time**.
Polynomial-time Solution	If an NP-Hard problem is solved in polynomial time, then P = NP.	If any NP-Complete problem is solved in polynomial time, then P = NP.
Examples	**Halting Problem, Optimization problems of NP-Complete problems**	**3-SAT, TSP (Decision version), Knapsack (Decision version)**

Polynomial time reduction:

Polynomial Time Reduction is the process of converting an instance of one problem **A** into an instance of another problem **B**.

🎯 **Purpose of Reduction:**

- To **compare complexity** of two problems
- To **prove hardness** (like NP-Completeness)
- To **reuse** solutions of known problems to solve new ones

🧠 **Why It's Important:** If we can reduce **A → B**, and we already know how to solve B efficiently, we can also solve A efficiently.

The input of a problem is called instance

There are two types of reduction. Turing reduction and Karp reduction.

Karp Reduction

Karp reduction is a method used to transform one **decision problem** into another in **polynomial time**. It is a **many-to-one reduction**, meaning that **a single instance of the original problem A** is converted into **a single instance of another problem B**.

Let **A** and **B** be decision problems.

We say that **A is Karp-reducible to B** (written as $A \leq_m^P B$) if there exists a **polynomial-time computable function** f such that:

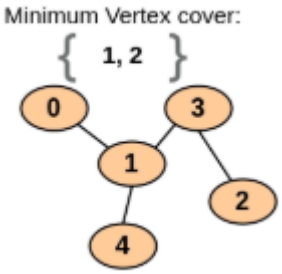
$$\forall\ w: w \in A \Leftrightarrow f(w) \in B$$

- w is an instance of problem **A**
- f(w) is the transformed instance of problem **B**
- f must be computable in **polynomial time**

Turing Reduction
Turing reduction is when we solve problem **A** by repeatedly asking for help from another problem **B** and still complete the solution in **polynomial time**.

Vertex cover (NP-Complete)

The **Vertex Cover Problem** is a fundamental problem in graph theory where the goal is to find the smallest set of vertices such that every edge in the graph is incident to at least one of the selected vertices.



- The vertex cover for the above graph can be: {0, 1, 2}, {1, 3, 4} as all of the edges available in the graph are adjacent to these vertices.
- But we have to find the minimum size vertex cover that is why for the above graph the minimum vertex cover is **{1, 2}**.

The vertex cover problem is an NP-Complete problem and it has been proven that the NP-Complete problem **cannot be solved in polynomial time.** So, we try to find the **near optimal** solution that finds a vertex cover.

Approach to solve the problem

1. Initialize an empty set **C** to store the result
2. Consider a collection **E** of all the edges in a graph.
3. While **E** is not empty:
    1. Pick an edge from collection **E**.
    2. Add **u** and **v** of the picked edge and store it in result set **C**.
    3. Remove all of the edges that incident on either **u** or **v**.
4. Print **result**.

Pseudo Code

```
SOLVE() {
    bool visited[total_vertices] = {false};

    for u = 0 to total_vertices - 1:
        if visited[u] is false:

           for each v in neighbors of u:
                if visited[v] is false:
                    visited[v] = true
                    visited[u] = true
                    break;

    for i = 0 to total_vertices - 1:
        if visited[i] is true:
            print i
}
```

Time Complexity: O(V + E)

Space Complexity: O(V)

Hamiltonian Cycle

In an undirected graph, the Hamiltonian path is a path, that visits each vertex exactly once.

Hamiltonian cycle or circuit is a Hamiltonian path, that there is an edge from the last vertex to the first vertex.

The naive way to solve Hamiltonian cycle problem is by generating all possible configurations of vertices and checking if any configuration satisfies the given constraints. However, this approach is not suitable for large graphs as its time complexity will be (O(N!)).

The following steps explain the working of backtracking approach −

- First, create an empty path array and add a starting vertex 0 to it.
- Next, start with vertex 1 and then add other vertices one by one.
- While adding vertices, check whether a given vertex is adjacent to the previously added vertex and hasn't been added already.
- If any such vertex is found, add it to the path as part of the solution, otherwise, return false.

Tractable Problems:

Tractable problems, also known as polynomial-time problems, are those for which efficient algorithms exist that can solve them in a reasonable amount of time. These algorithms have a time complexity that grows polynomially with the size of the input.

Characteristics:

1. Efficient Algorithms: Tractable problems can be solved efficiently using algorithms with polynomial time complexity.
2. Polynomial Time: The time required to solve tractable problems grows polynomially with the size of the input.
3. Examples: Examples of tractable problems include computing the sum of all elements in an array, sorting a list of numbers, and finding the shortest path in a graph using Dijkstra’s algorithm.

Intractable Problems:

Intractable problems, also known as non-polynomial-time problems, are those for which no efficient algorithms are known that can solve them in a reasonable amount of time. These problems often require exponential time to solve, making them impractical for large inputs.

Characteristics:

1. Lack of Efficient Algorithms: Intractable problems lack efficient algorithms that can solve them in polynomial time.
2. Exponential Time: The time required to solve intractable problems grows exponentially with the size of the input.
3. Examples: Examples of intractable problems include the Traveling Salesman Problem (TSP), the Knapsack Problem, and the Boolean Satisfiability Problem (SAT).

Computability

In Design and Analysis of Algorithms (DAA), computability is the ability to determine if an algorithm can solve a problem. It is a key topic in computability theory, a branch of mathematical logic and computer science.

How is computability determined?

- Prove a procedure exists

To show a problem is computable, you can describe a procedure and prove it always terminates and produces the correct answer.

- Show no algorithm exists

To show a problem is not computable, you can show that no algorithm exists that solves the problem.

Complexity

Complexity is the resources required to solve a problem, such as time and space.

Analysis of algorithms

The process of evaluating and understanding the performance characteristics of algorithms.

Complexity Classes

In computer science, problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity.

Types of Complexity Classes:

1. P Class.
2. NP Class.
3. NP-Complete.
4. NP-Hard.

1. Prove that Vertex Cover Problem is NP Complete

Vertex Cover:

The **Vertex Cover** problem is defined as follows:

- Given an undirected graph G=(V,E) and a number k, determine whether there exists a set of at most k vertices such that each edge in the graph has at least one endpoint in this set.

Proving Vertex Cover is NP-Complete:

1. **Vertex Cover is in NP**:
    - To verify a solution for the **Vertex Cover** problem, we can check if a given set of vertices has size **k** or less and if every edge in the graph is incident to at least one vertex in this set.

- Checking this can be done in polynomial time by simply iterating over the edges and verifying their coverage.
- Hence, **Vertex Cover** is in NP.

2. **Vertex Cover is NP-Hard**:
    - To prove that **Vertex Cover** is NP-Hard, we can reduce another known NP-Complete problem to it. A standard reduction comes from the **3-SAT** problem.

Reduction from 3-SAT or Independent Set to vertex cover:

- Given an instance of the **3-SAT** problem, we construct an instance of **Vertex Cover**:
- It can be shown that a **3-SAT** formula is satisfiable if and only if there exists a vertex cover of size at most k.
- This reduction from **3-SAT** to **Vertex Cover** can be done in polynomial time, proving that **Vertex Cover** is NP-Hard.

Since **Vertex Cover** is both in NP and NP-Hard, it is **NP-Complete**.

2. Prove that Traveling Salesman Problem (TSP) is NP Complete

Traveling Salesman Problem (TSP):

The **TSP** (Decision Version) problem is defined as follows:

- Given a set of cities and the distances between each pair of cities, is there a tour (a cycle) that visits every city exactly once and returns to the starting city such that the total length of the tour is less than or equal to a given number k?

Proving TSP is NP-Complete:

1. **TSP is in NP**:
    - To verify a solution for the **TSP** problem, we can check if a given tour visits every city exactly once and returns to the starting city, and whether the total distance of the tour is less than or equal to k.

- This can be done in polynomial time by verifying the cities visited and calculating the total distance of the tour.
- Hence, **TSP** is in NP.

2. **TSP is NP-Hard**:
   - To prove that **TSP** is NP-Hard, we can reduce another NP-Complete problem to it. A standard reduction comes from the **Hamiltonian Cycle** problem.

Reduction from Hamiltonian Cycle to TSP:

- The **Hamiltonian Cycle** problem is defined as: Given a graph, is there a cycle that visits every vertex exactly once?
- We can reduce **Hamiltonian Cycle** to **TSP** as follows:
  - Given an instance of the **Hamiltonian Cycle** problem (graph G), construct an instance of **TSP** by setting the distance between two vertices to:
    - 1 if there is an edge between them in G,
    - Infinite otherwise.
  - Set k to the number of vertices in the graph.
- If a **Hamiltonian Cycle** exists, there will be a tour in **TSP** with total length K, and vice versa.
- This reduction can be done in polynomial time, proving that **TSP** is NP-Hard.

Since **TSP** is both in NP and NP-Hard, it is **NP-Complete**.

What is the 3-SAT problem?

Before we talk about **3-SAT**, let's first understand the general **SAT (Boolean Satisfiability) problem**:

- **SAT Problem**: You are given a **Boolean formula**, which is made up of **variables** and **logical operations** (AND, OR, NOT), and your task is to

determine if there is a way to assign **True** or **False** values to the variables such that the entire formula becomes **True**.

For example, if we have a simple formula like:

- (x1∨x2)

Here, x1 and x2 are variables, and **∨** represents the **OR** operation. You need to check if there is a way to assign **True** or **False** to x1 and x2 such that the whole formula is **True**.

What is the 3-SAT problem?

The **3-SAT problem** is a **special case** of the **SAT problem** where:

- The formula is in **Conjunctive Normal Form (CNF)**, meaning it's a **conjunction (AND)** of multiple **clauses**.
- Each **clause** in the formula consists of **exactly 3 literals**.
- A **literal** is either a **variable** (like x1) or its **negation** (like ¬x1).

In simpler terms, a 3-SAT problem is a **SAT problem** where:

- The formula has several **ANDs** connecting the clauses.
- Each clause has exactly **3 terms** connected by **ORs**.

Example of a 3-SAT formula:

(x1∨¬x2∨x3)∧(x1∨x2∨¬x3)∧(¬x1∨x2∨x3)

In this formula:

- Each **clause** has 3 literals (variables or their negations).
- **ANDs** connect the clauses together.

1. Recursive Algorithms

A **recursive** algorithm is one that solves a problem by calling itself with a smaller subset of the original problem until it reaches a base case. The recursion stack is used to hold intermediate results.

Example: Factorial Function

The recursive factorial function for an integer **n** is defined as:

$$Factorial(n) = 1 \; if \; n = 0$$

$$n \times Factorial(n-1) otherwise$$

Analysis of Recursive Algorithms

1. **Time Complexity**:
    - In general, recursive algorithms can have a **higher time complexity** compared to iterative algorithms. This is due to the overhead introduced by repeated function calls and the potential for overlapping subproblems.
    - Example: The recursive Fibonacci algorithm has **O(2^n)** time complexity due to redundant calls.
2. **Space Complexity**:
    - Recursive algorithms use the **call stack** to keep track of intermediate values, which results in **O(n)** space complexity in many cases.
    - Example: For the factorial function, the depth of recursion will be **O(n)**, where each recursive call adds a new frame to the stack.
3. **Use Cases**:
    - Recursive algorithms are often used in problems that naturally follow a divide-and-conquer strategy, such as:
    - Tree and graph traversal (e.g., DFS)
    - Divide-and-conquer algorithms (e.g., Merge Sort, Quick Sort)
    - Backtracking problems (e.g., N-Queens, Sudoku solving)

2. Iterative Algorithms

An iterative algorithm solves a problem through repeated execution of a set of operations, typically using loops. The algorithm executes a fixed number of steps in each iteration and terminates once the problem is solved.

Example: Factorial Function (Iterative)

```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Analysis of Iterative Algorithms

1. **Time Complexity**:
    - Iterative algorithms generally have **lower time complexity** than recursive algorithms. They typically run in **O(n)** time when processing each element once in a loop.
    - Example: The iterative factorial function runs in **O(n)** time, where **n** is the input number.
2. **Space Complexity**:
    - Iterative algorithms tend to have **O(1)** space complexity because they do not require additional memory to store intermediate function calls. The computation happens in a single loop with only a few variables.
    - Example: The iterative factorial function only needs a constant amount of space (one variable for the result).
3. **Use Cases**:
    - Iterative algorithms are used for problems that can be solved in a sequential manner:
    - Simple loops (e.g., finding the sum of an array).
    - Sorting algorithms (e.g., Bubble Sort, Insertion Sort).
    - Linear traversals of data structures (e.g., linked lists, arrays).