Midterm 1 Review:

➔ Covers Chapters 1

Chapter 1 Outline: Introduction and Mathematical Background

1.1 Algorithms and Programs

- Overview of the distinction between algorithms and programs.

1.1.1 Analysis of Algorithms

- Delving into analyzing algorithms, likely focusing on time and space complexity.

1.1.2 Design of Algorithms

- Discussing the design principles and methodologies involved in creating effective algorithms.

1.2 Definitions and Terminology

- Clarification of key terms and concepts related to algorithms.

1.2.1 Problems

- Defining what constitutes a problem in the context of algorithms.

1.2.2 Running Time and Complexity

- Exploring the notions of running time and algorithmic complexity.

1.3 Order Notation

- Introduction to order notation for expressing the growth rate of functions.

1.3.1 Techniques for Order Notation

- Methods for applying order notation to analyze algorithmic efficiency.

1.3.2 Relationships between Order Notations

- Examining the relationships between different order notations.

1.3.3 Algebra of Order Notations

- Exploring how order notations can be manipulated algebraically.

1.3.4 Common Growth Rates

- Listing and explaining typical growth rates encountered in algorithmic analysis.

1.4 Mathematical Formulae

- Providing essential mathematical formulae used in algorithmic analysis.

1.4.1 Sequences

- Defining sequences relevant to algorithmic analysis.

1.4.2 Logarithm Formulae

- Presenting logarithmic formulae is important for algorithmic complexity.

1.4.3 Miscellaneous Formulae

- Covering additional mathematical formulae useful in algorithms.

1.4.4 Equivalence Relations

- Discussing the concept of equivalence relations in the context of algorithms.

1.5 Probability Theory and Random Variables

- Introducing probability theory and its application to random variables in the context of algorithms.

Chapter 1: Introduction and Mathematical Background

1.1 Algorithms and Programs

➔ **Algorithms and Programs**
- Algorithms are defined as step-by-step processes for problem-solving in a finite number of steps.
- Emphasis on studying algorithms over specific computer programs.
- Distinction: Algorithm as an abstraction, computer program as its implementation.
- Introduction of structured pseudocode for algorithm presentation.
- Use of assignment notation (←) for denoting variable assignments.

➔ **Techniques for Designing and Analyzing Algorithms**
- **Structured Pseudocode**
  - Presentation of algorithms using loop structures and conditional constructs.
- **Assignment Notation**
  - Introduction of the left arrow (←) as notation for variable assignment.
- **Problem-Solving**
  - Definition: An algorithm solves a problem if it finds a valid solution for every instance in finite time.

1.1.1 Analysis of Algorithms

Definition of Correctness and Types of Correctness Proofs:
- **Correctness Defined:** An algorithm is correct if it consistently provides a correct answer for every problem instance in a finite amount of time.
- **Types of Correctness Proofs:** Proofs can be formal or informal, with the level of detail varying based on the algorithm's kind and complexity.

Importance of Proving Correctness:
- **Critical Aspect:** The validity of an algorithm relies on its correctness; hence, proving correctness is a fundamental aspect of algorithmic analysis.

Efficiency Analysis - Time and Space Complexity:
- **Time Complexity:** Predicting the time an algorithm consumes without implementing it, often expressed using asymptotic notation.
- **Space Complexity:** Predicting the memory (space) requirements of an algorithm.

Criteria for Analyzing Algorithms:
- **Asymptotic Complexity (Order Notation):**
  - **O-notation (Big O):** This represents an upper bound on an algorithm's growth rate. For example, O(n^2) denotes quadratic time complexity, indicating the worst-case scenario where the running time does not grow faster than the square of the input size.
  - **Ω-notation (Big Omega):** This represents a lower bound on the growth rate, providing a lower limit for the algorithm's efficiency.
  - **Θ-notation (Theta):** A tight bound indicates upper and lower growth rate limits. It offers a precise characterization of the algorithm's behaviour.

    - **o-notation (Small o):** Describes an upper bound that is not asymptotically tight.
    - **ω-notation (Small Omega):** Indicates a lower bound that is not asymptotically tight.
- **Exact Number of Computations:**
    - Differentiates algorithms with the same asymptotic complexity by determining the exact number of operations performed.
- **Average-Case vs. Worst-Case Complexity:**
    - Considers how an algorithm performs on average inputs compared to its worst-case scenario.
    - Important for understanding the algorithm's robustness.
- **Efficiency Comparison:**
    - Evaluates multiple algorithms to identify the most efficient solution.
    - Aims to select the algorithm that optimally balances time and space considerations.
- **Lower Bounds:**
    - Briefly discuss the theoretical limits of algorithmic performance.
    - Provides insights into the inherent complexity of certain problems.
- **Memory (Space Complexity):**
    - Acknowledges the importance of memory usage as a critical resource in algorithmic efficiency.
    - Space complexity is a key consideration for practical implementations.

1.1.2 Design of Algorithms

Definition of "Design" in the Context of Algorithms:

- **Design Defined:** Refers to creating algorithms using general strategies and methodologies. Good design leads to the development of correct and efficient algorithms while avoiding ad hoc approaches that are hard to analyze.

Introduction to General Strategies for Creating New Algorithms:

- **Strategic Approach:** Emphasizes the importance of having effective design strategies to facilitate the creation of algorithms that are not only correct but also efficient.

Design Techniques:

- **Reductions:**
    - *Strategy:* Utilizing an algorithm designed for one problem as a subroutine to solve a different problem.
    - *Application:* Applied in scenarios where the solution to one problem can be used to solve another.
- **Divide-and-Conquer:**
    - *Strategy:* Breaking down a problem into smaller instances, solving them recursively, and combining their solutions to solve the original problem.
    - *Application:* Commonly used in algorithms that exhibit recursive structures.
- **Greedy Algorithms:**
    - *Strategy:* Building solutions incrementally by making locally optimal choices at each step.

    - *Application:* Frequently employed in optimization problems where making the best choice at each step leads to a globally optimal solution.
- **Dynamic Programming:**
    - *Strategy:* Solve smaller instances of a problem and store their solutions in a table for efficient retrieval.
    - *Application:* Effective for problems with overlapping subproblems, avoiding redundant computations.
- **Search Algorithms (Depth-First and Breadth-First):**
    - *Strategy:* Systematically traversing graphs, visiting vertices and edges in a structured manner.
    - *Application:* Often used in graph-related problems such as network modelling.
- **Backtracking:**
    - *Strategy:* A method of exhaustive search, similar to depth-first search, used to enumerate all possible solutions to a problem or find the optimal solution.
    - *Application:* Useful for solving difficult problems, particularly those classified as NP-hard.

Additional Algorithm Categorizations:

- *Serial vs. Parallel Algorithms:* The book focuses on serial algorithms executed by a single processor.
- *Deterministic vs. Randomized Algorithms:* Primarily explores deterministic algorithms but briefly mentions randomized algorithms in specific sections.

Brief Mention of Nondeterministic Algorithms in the Context of NP-Completeness:

- *Nondeterministic Algorithms:* Explored concerning NP-completeness, where they are viewed as methods for verifying the correctness of a given solution to a problem instance.

1.2: Definitions and Terminology

1.2.1 Problems

- **Problem Definition (Π):** A problem is essentially a well-defined computational task with a set of rules and specifications. For instance, sorting a list of numbers or finding the shortest path in a graph can be considered as problems.
- **Problem Instance (I):** When you have a specific set of input values for a given problem, it becomes a problem instance. For example, if the problem is sorting, an instance might be a particular list of numbers.
- **Example:**
    - Problem: Sorting
    - Instance: Given an array [3, 1, 4, 1, 5, 9, 2], find the sorted order.
- **Problem Solution:** The solution to a problem instance is the desired output or correct answer, often obtained by applying an algorithm to the given input.
- **Example:**
    - For the instance [3, 1, 4, 1, 5, 9, 2], the solution is [1, 1, 2, 3, 4, 5, 9] after applying a sorting algorithm.

- **Size of a Problem Instance (size(I)):** This is a critical measure, representing the amount of information needed to describe a particular instance. For instance, in the case of an array, the size would be related to the number of elements it contains.
- **Example:**
    - For the array [3, 1, 4, 1, 5, 9, 2], the size could be measured in terms of the number of elements (7 in this case).

1.2.2 Running Time and Complexity

- **Running Time of a Program (TM(I)):** This denotes the time taken by a specific program to execute on a given problem instance. Real-world conditions such as hardware and environment influence this.
- **Worst-Case Running Time (TM(n)):** By considering the maximum running time over all instances of a particular size n, we establish a measure for worst-case scenarios.
- **Best-Case Running Time (T_best_M(n)):** The minimum running time over all instances of size n, providing a lower bound on execution time for specific inputs.
- **Example:**
    - For a sorting algorithm, the best-case running time is achieved when the input is already sorted. This would be the lower bound for the execution time.
- **Average-Case Running Time (T_avg_M(n)):** The expected running time over all instances of size n, offering insights into how the algorithm behaves on typical inputs.
- **Example:**
    - For a search algorithm, the average-case running time considers the likelihood of finding an element in different positions within a sorted array.

Observations:

- **Running Time vs. Complexity:** Running time is empirical and depends on specific conditions, while complexity is a high-level, asymptotic measure, abstracting away from hardware and environment specifics.
- **Relationship between Complexity and Running Time:** The complexity of an algorithm provides insights into its behavior on inputs of different sizes. It helps predict how the running time will grow as the size of the input increases.

Relationship between Complexity and Running Time

- **Worst-Case Complexity of an Algorithm:** Denoted as f(n), it signifies that there exists a program with running time in the order of f(n). This measure helps understand how the algorithm performs in the worst-case scenario.
- **Best-Case Complexity of an Algorithm:** Similar to worst-case complexity, this measure indicates the best running time over all instances of a particular size. It provides insights into the algorithm's optimal behavior.
- **Average-Case Complexity of an Algorithm:** The expected complexity over all instances of a particular size, offering a realistic expectation of how the algorithm performs on typical inputs.

Key Points:

- **Crossover Point Example:** The example of $T_1(n) = 75n + 500$ and $T_2(n) = 5n^2$ illustrates the importance of considering not only asymptotic complexity but also the behavior on small inputs.
- **Running Time Scenarios:** Understanding the worst, best, and average-case running times helps in making informed decisions about algorithm selection based on specific application requirements.

1.3: Order Notation

In Chapter 1.3, the focus is on order notation, a crucial tool in algorithm analysis. The primary goal is to simplify expressions representing the complexity of algorithms by ignoring lower-order terms and constant factors. This allows for a more abstract and generalized understanding of algorithmic efficiency.

Key Concepts

1.3.1 Types of Order Notation

1. **O-notation:**
   - Defined as $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
   - Indicates that the complexity of f is not higher than the complexity of g.
2. **Ω-notation:**
   - Defined as $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
   - Signifies that the complexity of f is not lower than the complexity of g.
3. **Θ-notation:**
   - Defined as $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
   - Indicates that the functions f and g have the same complexity.
4. **o-notation:**
   - Defined as $f(n) \in o(g(n))$ if, for every constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
   - Implies that f has a lower complexity than g.
5. **ω-notation:**
   - Defined as $f(n) \in \omega(g(n))$ if, for every constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
   - Implies that f has a higher complexity than g.

1.3.2 Relationships between Order Notations

- **Theorem 1.4 (Comparison of Growth Rates):**
  - Compares the growth rates of functions f and g based on their limits.
- **Theorem 1.5 (Comparison of Polynomials):**
  - Extends the comparison to polynomials, stating that any polynomial $f(n)$ is in $\Theta(x^t)$.
- **Theorem 1.6 (Relationships Between Notations):**
  - Establishes relationships between Θ, O, and Ω notations.

1.3.3 Algebra of Order Notations

- **Theorem 1.7 (Maximum Rules):**
  - Simplifies expressions involving the maximum of two functions under O, Θ, and Ω notations.
- **Theorem 1.8 (Summation Rules):**
  - Extends rules for summation of functions under O, Θ, and Ω notations.
- **Theorem 1.9 (Product Rules):**
  - Describes the rules for products of functions under O, Θ, and Ω notations.
- **Theorem 1.10 (Quotient Rules):**
  - Establishes rules for quotients of functions under O, Θ, and Ω notations.

1.3.4 Common Growth Rates

- Provides a list of common growth rates, ranging from constant to exponential.

Examples and Applications

1. **Example 1.1 (Polynomial Comparison):**
   - Compares the growth rate of $f(n) = n^2 - 7n - 30$ with $g(n) = n^2$ using first principles.
2. **Example 1.4 (Comparison of Growth Rates):**
   - Compares the growth rates of $(\ln n)^2$ and $n^{1/2}$ using l'Hôpital's rule.
3. **Example 1.5 (Comparing n^2 and $n^2 - 7n - 30$):**
   - Compares the growth rates of n^2 and $n^2 - 7n - 30$ using limits.
4. **Example 1.6 (Comparing $3 + (-1)^n$ and n):**
   - Compares the growth rates of $3 + (-1)^n$ and n to show the relationship between their complexities.
5. **Example 1.7 (Comparing $n^2 + \sin(n\pi/2)$ and n):**
   - Proves that $n^2 + \sin(n\pi/2)$ is in $\Theta(n)$ despite the limit non-existence.
6. **Example 1.8 (Comparing $n \sin(n\pi/2) + 1$ and $\sqrt{n}$):**
   - Establishes that there is no relation between the growth rates of $n \sin(n\pi/2) + 1$ and $\sqrt{n}$.

1.4 Mathematical Formulae

1.4.1 Sequences

- Arithmetic Sequence:
    - Form: Start with a number, then add a fixed amount each time.
    - Sum Formula: The total is the number of terms times the average of the first and last term.
- Geometric Sequence:
    - Form: Start with a number, then multiply by a fixed amount each time.
    - Sum Formula: Different cases for whether the multiplying factor is more than 1, exactly 1, or between 0 and 1.
- Arithmetic-Geometric Sequence:
    - Form: A mix of arithmetic and geometric sequences.
    - Sum Formula: Total sum calculation, with a different formula when the multiplying factor is not 1.
- Harmonic Sequence:
    - Form: Reciprocal of natural numbers.
    - Sum: The sum of these reciprocals grows at a logarithmic rate.

1.4.2 Logarithm Formulae

- Basic Logarithm Properties:
    - Rules for combining and manipulating logarithms.

1.4.3 Miscellaneous Formulae

- Factorial:
    - Stirling's Formula: An approximation for large factorials.
- Logarithm of Factorial:
    - Logarithmic growth rate of factorials.
- Sum of Logarithms:
    - The sum of logarithms grows at a rate proportional to the input size.
- Sum of Reciprocals:
    - A specific infinite sum of reciprocals converges to a constant.
- Sum of Powers:
    - The sum of consecutive powers has a predictable growth rate.
- Binomial Theorem:
    - A formula for expanding expressions like (x + y)^n.

1.4.4 Equivalence Relations

- Equivalence Relation Properties:
    - Basic properties for relations that group things together.
- Examples:
    - Examples where things are considered equivalent based on specific rules.

1.5 Probability Theory and Random Variables

1. Basic Probability Concepts

- Random Variable X:
    - Represents outcomes on a set X with assigned probabilities.
    - Denoted as Pr[X = x], indicating the chance of X taking a specific value x.

    - Probabilities in X sum up to 1.
- Example - Coin Toss:
    - Imagine a coin toss as a random variable with outcomes {heads, tails}.
    - Probability distribution: Pr[heads] = Pr[tails] = 1/2.

1. Conditional Probability and Events

- Conditional Probability:
    - Pr[X = x | Y = y] signifies the likelihood of X given Y.
    - Bayes' Theorem links joint and conditional probabilities.
- Events and Subset E:
    - If E is a subset of X, Pr[X ∈ E] is the chance X falls within E.
- Example - Dice Throw:
    - Create a random variable Z for the sum of two dice.
    - Compute probabilities for specific sums using the joint distribution.

1. Expectation and Linearity

- Expectation of X (E[X]):
    - Represents the average value of a random variable X.
    - Computed as the sum of (Pr[x] × x) for all values in X.
- Linearity of Expectation:
    - E[X + Y] = E[X] + E[Y], applicable to any random variables X and Y.
- Example - Sum of Dice:
    - Determine the expected value of a single die throw.
    - Extend to the sum of two dice using linearity of expectation.

1. Theorems and Applications

- Theorem 1.12:
    - The expected number of trials until the first successful outcome is 1/p.
- Theorem 1.13:
    - Expectation of a random variable X is the sum of the probabilities of X being at least i.
- Theorem 1.16 (Linearity):
    - Generalizes linearity of expectation for the sum of any r random variables.
- Example 1.13 - Book Redistribution:
    - Utilize indicator random variables to find the expected number of correctly returned books.

Exercise Chapter 1

Exercises

1.1 Give a proof from first principles (not using limits) that $n^3 - 100n + 1000 \in \Theta(n^3)$.

1. **For a Lower Value of n:**
   - Let's choose $n = 1$.
   - $n^3 - 100n + 1000$ becomes $1^3 - 100 \times 1 + 1000 = 901$.
   - n^3 becomes $1^3 = 1$.
   - Here, $n^3 - 100n + 1000$ is greater than n^3, but the difference is significant because n is small.
2. **For an In-Between Value of n:**
   - Let's choose $n = 50$.
   - $n^3 - 100n + 1000$ becomes $50^3 - 100 \times 50 + 1000 = 124900$.
   - n^3 becomes $50^3 = 125000$.
   - Here, $n^3 - 100n + 1000$ is slightly less than n^3, and the additional terms have a smaller impact.
3. **For an Upper Value of n:**
   - Let's choose $n = 1000$.
   - $n^3 - 100n + 1000$ becomes $1000^3 - 100 \times 1000 + 1000 = 999001000$.
   - n^3 becomes $1000^3 = 1000000000$.
   - Here, $n^3 - 100n + 1000$ is still growing, but the difference from n^3 is becoming negligible as n gets larger.

The question is essentially asking you to analyze the growth behavior of the expression $n^3 - 100n + 1000$ as n becomes very large. In other words, when n gets bigger and bigger, how does the expression change?

Here's what each part means:

n^3**:** This term is like the boss in the room. It's the most significant part of the expression and has the highest impact on the growth. It represents a cubic growth, meaning it gets larger as the cube of n.

$-100n + 1000$**:** These are like employees in the room. They have an impact on the growth, but as n becomes very large, their influence becomes less significant compared to the boss (n^3).

The question is asking you to classify this expression based on its growth:

- **Big O (O):** Does the expression grow at most as fast as a certain function (in this case, n^3)?
- **Big Omega (Ω):** Does the expression grow at least as fast as a certain function (again, n^3 in this case)?
- **Theta (Θ):** Does the expression grow at the same rate as a certain function?

In summary, the question is about understanding the growth rate behavior of the given expression and expressing it using mathematical notations like O, Ω, and Θ.

1.2 Compare the growth rates of the functions

$$n^{\log_2 \log_2 n} \quad \text{and} \quad (\log_2 n)^{\log_2 n}.$$

1.3 Compare the growth rates of the following three functions:

$$n!, \quad n^{n+1} \quad \text{and} \quad n^{n+1/2}.$$