Theoretical Concepts and Definitions for Memorization

This document compiles the key theoretical concepts, definitions, and strategy explanations extracted from the provided lectures.

Lecture 1: Introduction and Asymptotic Analysis

Algorithm: An algorithm is defined as a step-by-step procedure designed for solving a specific problem within a finite amount of time and utilizing a finite amount of space.

Analysis of an Algorithm: This involves determining the quantity of time and space resources required for an algorithm to execute. It evaluates the algorithm's complexity to gauge its efficiency concerning time and space.

Time Complexity: This quantifies the number of elementary operations or time-steps an algorithm needs to complete its execution, typically expressed as a function of the input size.

Space Complexity: This measures the amount of memory space an algorithm requires to run to completion, also usually expressed as a function of the input size.

Why Complexity Analysis (Not Execution Time): Algorithm complexity is preferred over direct execution time measurement because execution time is heavily dependent on specific hardware capabilities and operating system environments, rather than solely on the algorithm's inherent efficiency. Complexity analysis provides a more abstract, machine-independent measure.

Dependence on Input Size (n): The number of operations or the memory allocated by an algorithm generally increases as the size of the input (n) increases. Complexity analysis focuses on how these resource requirements scale with the input size.

Asymptotic Notation: This provides a way to describe the limiting behavior or growth rate of a function (representing an algorithm's complexity) as the input size grows very large. It gives a rough estimation, focusing on the dominant terms.

- **Big-O Notation (O):** Represents the upper bound on the growth rate of an algorithm's complexity. It describes the worst-case scenario, guaranteeing that the algorithm's performance will not exceed this rate.

- **Omega Notation (Ω):** Represents the lower bound on the growth rate. It describes the best-case scenario, indicating the minimum resources the algorithm will require.
- **Theta Notation (Θ):** Represents a tight bound, indicating that the algorithm's growth rate is bounded both from above and below by the same function. It describes the average or typical-case behavior when the best and worst cases have the same growth rate.

Algorithm Cases: * **Worst Case:** The input scenario that causes the algorithm to perform the maximum number of operations. Analyzed using Big-O notation. * **Best Case:** The input scenario that causes the algorithm to perform the minimum number of operations. Analyzed using Omega notation. * **Average Case:** The expected performance over all possible inputs. Often analyzed using Theta notation if the upper and lower bounds match.

Growth Rates: Describes how the resource requirements (time or space) of an algorithm increase as the input size (n) grows. * **Constant (O(1)):** Runtime/space does not depend on the input size. * **Logarithmic (O(log n)):** Runtime/space increases very slowly with input size (e.g., halving the problem size at each step). * **Linear (O(n)):** Runtime/space increases proportionally to the input size. * **Polynomial (O(n^k)):** Runtime/space increases as a polynomial function of the input size (e.g., $O(n^2)$, $O(n^3)$). * **Exponential (O(k^n)):** Runtime/space increases extremely rapidly with input size. Generally considered inefficient for large inputs. * **Factorial (O(n!)):** Runtime/space grows even faster than exponential. Highly inefficient.

Lecture 2: Recursion Analysis

Recursion: A problem-solving technique where a function calls itself to solve smaller instances of the same problem. It involves breaking down a problem until a base case is reached.

Components of Recursion: * **Base Case(s):** The condition(s) under which the function stops calling itself. This is the stopping criterion that prevents infinite recursion. * **Recursive Case:** The part of the function where it calls itself, typically with modified arguments that move closer to the base case.

Recursion Analysis Methods: * **Backward Substitution:** A method to analyze the time complexity of recursive algorithms by repeatedly substituting the recursive relation into itself until a pattern emerges or the base case is reached. * **Master Theorem:** A "cookbook" method for determining the time complexity of recursive relations that fit the specific form: `T(n) = aT(n/b) + f(n)`, where `f(n)` is often `c*n^d`. The theorem provides three cases based on the comparison between `a` and `b^d`: 1. If `a < b^d`, then

`T(n) = O(n^d)`. 2. If `a = b^d`, then `T(n) = O(n^d * log n)`. 3. If `a > b^d`, then `T(n) = O(n^(log_b a))`. * Note: The theorem applies when `a >= 1`, `b > 1`, and `c` and `d` are constants (with `d >= 0` in the lecture's form `c*n^d`).

Recursive Relation: An equation that defines a sequence recursively, where a term is defined as a function of preceding terms. For algorithms, it expresses the time complexity `T(n)` in terms of the complexity of smaller inputs.

Space Complexity of Recursion: * **Call Stack:** A data structure used by the system to manage function calls. It operates in a Last-In-First-Out (LIFO) manner. * **Stack Frame:** When a function is called (including recursive calls), a new frame is pushed onto the call stack. This frame stores information like parameters, local variables, and the return address. * **Complexity:** The space complexity of a recursive algorithm is often determined by the maximum depth of the recursion call stack. Deep recursion can lead to significant space usage and potentially a **Stack Overflow** error if the stack limit is exceeded. * Example: Recursive factorial has O(n) space complexity due to n nested calls.

Why Use Recursion? * **Natural Structure:** Simplifies code for problems with inherent recursive definitions (e.g., factorial, Fibonacci). * **Problem Decomposition:** Aligns well with strategies like Divide and Conquer, breaking problems into smaller, self-similar subproblems. * **State Management:** Useful in algorithms like Backtracking, where the call stack implicitly manages the state for different exploration paths.

Algorithm Design Strategy: The general approach or methodology used to design an efficient and effective algorithm for a specific problem. Choosing the right strategy helps structure the solution and optimize performance.

Common Algorithm Design Strategies (Introduced/Listed): Brute Force, Greedy, Decrease and Conquer, Divide and Conquer, Transfer and Conquer, Backtracking, Dynamic Programming, Evolutionary Algorithms.

Decrease and Conquer Strategy: Reduces a problem instance to a smaller instance of the same problem, solves the smaller instance, and then extends the solution to the original problem. * **Decrease by a Constant Value:** Problem size reduced by a constant amount (e.g., `n-1` in factorial). * **Decrease by a Constant Factor:** Problem size reduced by a constant factor (e.g., `n/2` in binary search or recursive decimal-to-binary). * **Decrease by a Variable Amount:** Problem size reduced by a variable amount depending on the instance's properties (e.g., Euclid's algorithm for GCD where size reduces by `a % b` or `b % a`).

Lecture 3: Brute Force and Greedy Algorithms

Brute Force Strategy: A straightforward approach that systematically explores all possible solutions or candidates to find the correct or optimal one. It often involves an exhaustive search. * **Characteristics:** Simple to design but often inefficient, especially for large problems, frequently leading to exponential ($O(k^n)$) or factorial ($O(n!)$) time complexity. * **Optimal Solution:** The best possible solution among all feasible solutions for a given problem. * **Feasible Solution:** A solution that satisfies the problem's constraints.

Traveling Salesman Problem (TSP): Find the shortest possible route that visits each city (node) exactly once and returns to the starting city. * **Feasible Solution (TSP):** A route visiting each node exactly once (a Hamiltonian Circuit). * **Optimal Solution (TSP):** The Hamiltonian Circuit with the minimum total length/cost. * **Brute Force Approach (TSP):** Generate all possible permutations ($n!$) of cities, calculate the length of each, and select the shortest. Complexity: $O(n!)$.

Knapsack Problem (0/1): Given a set of items, each with a weight and a value, determine the subset of items to include in a collection so that the total weight is less than or equal to a given limit (knapsack capacity) and the total value is maximized. * **Feasible Solution (Knapsack):** A subset of items whose total weight does not exceed the capacity W. * **Optimal Solution (Knapsack):** The feasible subset with the maximum total value. * **Brute Force Approach (Knapsack):** Generate all possible subsets (2^n) of items, check feasibility (weight <= W), and select the feasible subset with the maximum value. Complexity: $O(2^n)$.

Greedy Strategy: An approach that makes the locally optimal choice at each stage with the hope of finding a global optimum. It does not explore all possibilities but makes the choice that seems best at the moment. * **Characteristics:** Often faster than brute force (e.g., polynomial time), but does not always guarantee the globally optimal solution. It aims for efficient, nearly-optimal solutions. * **Greedy Approach (TSP - Nearest Neighbor):** Start at a city, repeatedly visit the nearest unvisited city until all cities are visited. Complexity: $O(n^2)$. Note: Not guaranteed to be optimal.

Minimum Spanning Tree (MST): Given a connected, undirected graph with weighted edges, find a subgraph that connects all vertices together (a spanning tree) without any cycles and with the minimum possible total edge weight. * **Spanning Tree:** A subset of edges in a graph that connects all vertices without forming a cycle. * **Greedy Approaches (MST):** Both Kruskal's and Prim's algorithms use a greedy strategy to find the optimal MST. * **Kruskal's Algorithm:** Sorts all edges by weight and iteratively adds the next smallest edge that does not form a cycle. * **Prim's Algorithm:** Starts with a

single vertex and iteratively adds the cheapest edge that connects a vertex in the growing tree to a vertex outside the tree.

Lecture 4: Divide and Conquer

Divide and Conquer Strategy: A strategy that breaks down a problem into smaller, independent subproblems of the same type, solves these subproblems recursively, and then combines their solutions to solve the original problem. * **Steps:** 1. **Divide:** Break the problem into smaller subproblems. 2. **Conquer:** Solve the subproblems recursively. If the subproblems are small enough, solve them directly (base case). 3. **Combine:** Combine the solutions of the subproblems to get the solution for the original problem. * **Examples:** Merge Sort, Quick Sort.

Merge Sort: A sorting algorithm following the Divide and Conquer strategy. * **Process:** 1. **Divide:** Split the n-element sequence into two subsequences of n/2 elements each. 2. **Conquer:** Sort the two subsequences recursively using Merge Sort. 3. **Combine:** Merge the two sorted subsequences to produce the sorted answer. * **Merge Function:** Takes two sorted subarrays and combines them into a single sorted array. This step typically requires O(n) time and O(n) auxiliary space.

Quick Sort: A sorting algorithm following the Divide and Conquer strategy. * **Process:** 1. **Divide:** Partition the array `S[q..r]` into two (possibly empty) subarrays `S[q..p-1]` and `S[p+1..r]` such that each element of `S[q..p-1]` is less than or equal to `S[p]` (the pivot), and each element of `S[p+1..r]` is greater than or equal to `S[p]`. The index `p` of the pivot is computed. 2. **Conquer:** Sort the two subarrays `S[q..p-1]` and `S[p+1..r]` by recursive calls to Quick Sort. 3. **Combine:** Because the subarrays are sorted in place, no work is needed to combine them; the entire array `S[q..r]` is now sorted. * **Partition Function:** Rearranges the subarray in-place around a chosen pivot element. The choice of pivot and the resulting balance of the partition significantly impact performance. * **Worst Case:** Occurs when the partitioning routine produces one subproblem with n-1 elements and one with 0 elements (e.g., when the array is already sorted and the first or last element is chosen as the pivot). * **Best Case:** Occurs when the partitioning routine produces two subproblems of size roughly n/2.

Lecture 5: Non-Comparative Sorting and Transfer & Conquer

Comparative Sorting: Sorting algorithms that determine the sorted order by comparing pairs of elements (e.g., Merge Sort, Quick Sort, Selection Sort). Lower bound for comparison sorts is Ω(n log n).

Non-Comparative Sorting: Sorting algorithms that do not rely on comparing elements. They often use properties of the data (like digit values or counts) to sort. Can achieve better than O(n log n) time complexity (e.g., O(n)) under certain conditions (like a limited range of input values). * **Examples:** Counting Sort, Radix Sort, Bucket Sort.

Counting Sort: A non-comparative sorting algorithm suitable for integers within a small range (k). * **Process:** Counts the occurrences of each distinct element value and uses these counts to determine the positions of each element in the sorted output sequence. * **In-place Sorting:** An algorithm that sorts the input data without needing significant additional memory space beyond the input array itself (typically O(1) extra space). Counting Sort is not in-place.

Stable Sort: A sorting algorithm that preserves the relative order of records with equal keys (values). If two items have the same value, their order in the output sequence is the same as their order in the input sequence. * **Stable Examples:** Merge Sort, Counting Sort, Bubble Sort, Insertion Sort. * **Unstable Examples:** Quick Sort, Selection Sort, Heap Sort.

Radix Sort: A non-comparative sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value (working from least significant digit to most significant, or vice versa). * **Process:** Uses a stable sorting algorithm (like Counting Sort) repeatedly on each digit position. * **Radix (r):** The number of unique digits or characters in a specific positional numeral system (e.g., 10 for decimal).

Bucket Sort: A non-comparative sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sort algorithm. * **Process:** Divides the range of input values into `k` buckets, distributes the `n` elements into these buckets, sorts each bucket, and then concatenates the sorted buckets. * **Performance:** Efficiency depends heavily on the uniform distribution of elements into buckets.

Transfer and Conquer Strategy: A strategy that transforms the problem instance into another, potentially simpler or more structured, instance that is easier to solve. The solution to the transformed instance then provides the solution to the original problem. * **Goal:** Often used for problems involving frequent operations (like searching) where an initial transformation cost (like sorting) pays off over many subsequent operations. * **Example:** Sorting an array first (transfer) to enable fast searching using binary search (conquer).

Binary Search: An efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half.

Uniqueness Problem: Determine if all elements in a given array are distinct. * **Transfer & Conquer Approach:** Sort the array first, then check adjacent elements for equality. Complexity dominated by the sort.

Mode Finding Problem: Find the value that appears most frequently in a data set. * **Transfer & Conquer Approach:** Sort the array first, then scan the sorted array to count consecutive occurrences of each element and find the maximum count. Complexity dominated by the sort.

Lecture 6: Backtracking and Complexity Classes

Backtracking Strategy: A recursive problem-solving technique that incrementally builds a solution, abandoning paths (backtracking) as soon as it determines they cannot lead to a valid or optimal solution based on problem constraints. * **Characteristics:** Explores the solution space more efficiently than Brute Force by pruning invalid branches early. Often used for constraint satisfaction problems, puzzles, and games. * **General Steps:** 1. **Choose:** Select a potential next step/component for the solution. 2. **Constraint Check:** Verify if the choice is valid according to problem rules. 3. **Goal Check:** Determine if the current state constitutes a complete solution. 4. **Recurse:** If valid and not yet a solution, recursively call to continue building. 5. **Undo (Backtrack):** If a choice leads to a dead end or an invalid state, undo the choice and try the next alternative. * **Application (TSP):** Can be used to find the shortest path, pruning paths that exceed the current best known length or violate constraints (visiting a city twice). * **Application (Sudoku):** Places numbers in empty cells one by one, checking row, column, and sub-grid constraints. If a placement violates constraints or leads to a dead end, it backtracks and tries a different number.

Computational Complexity: The study of the resources (primarily time and space) required to solve computational problems.

Complexity Class: A set of problems grouped together based on their shared resource-based complexity characteristics (e.g., solvable in polynomial time).

- **P (Polynomial Time):** The class of decision problems that can be solved by a deterministic algorithm in polynomial time (e.g., O(n^k) for some constant k).
- **EXP (Exponential Time):** The class of decision problems that can be solved by a deterministic algorithm in exponential time (e.g., O(2^(n^k))).
    - Relationship: P is a subset of EXP ($P \subseteq EXP$).
- **Decision Problem:** A problem that requires a 'yes' or 'no' answer.

- **NP (Nondeterministic Polynomial Time):** The class of decision problems for which a given solution (or certificate) can be verified in polynomial time by a deterministic algorithm. It does not necessarily mean the problem can be solved in polynomial time.
    - Relationship: P is a subset of NP ($P \subseteq NP$), and NP is a subset of EXP ($NP \subseteq EXP$). The question of whether P = NP is a major unsolved problem in computer science.
- **NP-Complete:** The class of problems within NP that are the "hardest" problems in NP. A problem is NP-complete if:
    1. It is in NP (verifiable in polynomial time).
    2. Every other problem in NP can be reduced to it in polynomial time (meaning it's at least as hard as any other NP problem).
    3. Examples: Sudoku (decision version), Traveling Salesman Problem (decision version), Knapsack (decision version).
- **NP-Hard:** The class of problems that are at least as hard as the hardest problems in NP. An NP-hard problem does not have to be a decision problem and does not have to be in NP itself (i.e., it might not be verifiable in polynomial time). If an NP-hard problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time (implying P=NP).
    - Optimization versions of NP-complete problems (like finding the shortest TSP tour or the optimal Knapsack solution) are often NP-hard.

Lecture 7: Dynamic Programming

Dynamic Programming (DP) Strategy: An optimization technique that solves complex problems by breaking them down into simpler, overlapping subproblems. It solves each subproblem only once and stores its solution, typically in a table (memoization or tabulation), to avoid redundant computations. * **Characteristics:** 1. **Overlapping Subproblems:** Solutions to subproblems are reused multiple times. 2. **Optimal Substructure:** The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems. * **Process:** 1. Divide the problem into smaller overlapping subproblems. 2. Solve each subproblem (often starting from the smallest) and store the result. 3. Combine the stored results to solve the original problem. * **Trade-off:** Often improves time complexity (compared to naive recursion or brute force) by avoiding re-computation, but usually at the cost of increased space complexity due to storing subproblem solutions. * **Comparison with Divide and Conquer:** Both break

problems down. However, DP handles overlapping subproblems (storing results is key), while Divide and Conquer typically deals with independent subproblems.

Applications of Dynamic Programming: * **Longest Common Subsequence (LCS):** Finding the longest subsequence common to two sequences. Used in bioinformatics (gene sequences), text comparison (diff utilities). * **Knapsack Problem (0/1):** Selecting items with maximum total value within a weight constraint. Used in resource allocation, investment portfolio optimization. * **Shortest Path Problems (e.g., Dijkstra's):** Finding the path with the minimum cost/distance between nodes in a graph. Used in navigation systems, network routing, robotics.

Dijkstra's Algorithm: An algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It works for graphs with non-negative edge weights. * **Approach:** Maintains a set of unexplored nodes and iteratively selects the unexplored node with the shortest known distance from the source. It updates the distances of neighbors of the selected node. Uses a greedy approach within a DP-like framework of building up the shortest path tree. * **Routing Table:** Typically maintains the current shortest distance found from the source to each node and the predecessor node on that path.

Lecture 8: Evolutionary Algorithms

Evolutionary Algorithms (EA) Strategy: A class of population-based metaheuristic optimization algorithms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. * **Goal:** To find near-optimal solutions to complex problems, particularly when traditional methods are too slow or fail. * **Core Idea:** Mimics Darwin's theory of "survival of the fittest." A population of candidate solutions evolves over generations. * **Fitness Function:** A function used to evaluate the quality or "goodness" of a candidate solution in the context of the problem being solved. * **Process:** Iteratively improves a population of solutions based on their fitness. Better solutions are more likely to be selected to produce the next generation.

Genetic Algorithms (GA): A specific type of Evolutionary Algorithm that uses techniques inspired by natural genetics. * **Representation:** Solutions are often represented as "chromosomes," typically strings (often binary strings). * **Chromosome:** Represents a candidate solution. * **Gene:** A part of the chromosome, often representing a component or parameter of the solution (e.g., a single bit in a binary string). * **Population:** A set of candidate solutions (chromosomes). * **Reproduction Operators:** Used to create new solutions (offspring) for the next generation: * **Selection:** Choosing parent solutions from the current population based on their fitness (fitter individuals have a higher chance of being selected). * **Crossover:** Combining genetic material from two parent

chromosomes to create one or more offspring chromosomes. Mimics biological recombination. * **Mutation:** Randomly altering one or more genes in a chromosome. Introduces diversity into the population and helps avoid getting stuck in local optima. * **General GA Steps:** 1. Initialize a population of random solutions. 2. Evaluate the fitness of each individual in the population. 3. Repeat until a termination criterion is met (e.g., number of generations, fitness threshold reached, no improvement): a. Select individuals to be parents. b. Apply crossover and mutation operators to create offspring. c. Form a new population (e.g., replacing the old one or combining parents and offspring). d. Evaluate the fitness of the new individuals. 4. Return the best solution found. * **Application (Knapsack):** Chromosomes can represent subsets of items (e.g., a binary string where 1 means item included, 0 means excluded). The fitness function evaluates the total value, possibly penalized if the weight constraint is violated.