Study Guide 4: Identifying Problem Types and Advanced Strategies

Andrés Oswaldo Calderón Romero, PhD.

November 18, 2025

Introduction

Algorithm design is a cornerstone of computer science, enabling the systematic development of efficient and scalable solutions to computational problems. As the complexity and size of real-world data continue to grow, mastering advanced algorithmic strategies becomes increasingly important. This study guide is intended to support students in recognizing the types of problems that arise in practice and in selecting the most suitable techniques to solve them.

The material focuses on classifying problems by solution strategy, applying advanced methods such as divide and conquer, dynamic programming, and greedy algorithms, and evaluating the complexity of algorithms under different scenarios. By understanding when and how to use each approach, students will develop the analytical skills required to design optimal algorithms and interpret their performance rigorously.

Objectives

- Recognize different types of problems and solution strategies.
- Apply advanced strategies such as dynamic programming and divide and conquer.
- Develop skills to analyze complex problems.

Topics to Cover

Classification of Problems by Solution Strategy

- Differences between brute force, divide and conquer, dynamic programming, and greedy algorithms.
- Classic problem examples for each strategy.

Application of Advanced Techniques

- **Divide and Conquer:** Analysis using the Master Theorem.
- **Dynamic Programming:** Examples such as Longest Common Subsequence (LCS) and Knapsack.
- **Greedy Algorithms:** Basic concepts.

Complexity Analysis for Advanced Strategies

- Evaluation of average case, worst case, and best case.
- Identification of patterns in algorithmic solutions.

1 Classification of Problems by Solution Strategy

Understanding how to classify problems by their optimal solution strategy is essential in algorithm design. Four major strategies include **Brute Force**, **Divide and Conquer**, **Dynamic Programming**, and **Greedy Algorithms**. Each has its strengths and is suited for specific types of problems.

1.1 Brute Force

1.1.1 Definition

Tries all possible combinations or solutions without any optimization.

1.1.2 Characteristics

- Simple to implement.
- Often inefficient for large input sizes.
- Guarantees correctness (if exhaustive).

1.1.3 When to Use

- Small input sizes.
- When correctness is more important than speed.
- As a baseline to compare optimized algorithms.

1.1.4 Examples

- Exhaustive search in puzzles.
- Traveling Salesman Problem (TSP): trying all permutations.
- Subset Sum Problem (without optimization).

1.2 Divide and Conquer

1.2.1 Definition

Divides the problem into smaller subproblems, solves each independently, and combines the results.

1.2.2 Characteristics

- Recursive in nature.
- Often leads to logarithmic or polynomial time.
- Performance can be analyzed using the Master Theorem.

1.2.3 When to Use

- Problems that can be broken into independent subproblems.
- Large input sizes where recursion is manageable.

1.2.4 Examples

- Merge Sort and Quick Sort
- Binary Search
- Closest Pair of Points (Computational Geometry)

1.3 Dynamic Programming

1.3.1 Definition

Solves complex problems by breaking them into overlapping subproblems and storing intermediate results (memoization or tabulation).

1.3.2 Characteristics

- Avoids redundant calculations.
- Efficient for optimization problems.
- Requires optimal substructure and overlapping subproblems.

1.3.3 When to Use

- Problems with repeated subproblems.
- Optimization problems (maximize/minimize cost/value).

1.3.4 Examples

- Longest Common Subsequence (LCS)
- Knapsack Problem
- Fibonacci Sequence
- Edit Distance

1.4 Greedy Algorithms

1.4.1 Definition

Makes a locally optimal choice at each step with the hope of finding a global optimum.

1.4.2 Characteristics

- Fast and simple.
- Doesn't always produce an optimal solution.
- Requires the greedy choice property and optimal substructure.

1.4.3 When to Use

- Problems where local decisions lead to a global optimum.
- When speed is more critical than exactness.

1.4.4 Examples

- Activity Selection
- Huffman Encoding
- Minimum Spanning Tree (Kruskal's and Prim's)
- Fractional Knapsack

1.5 Summary Table

Table 1 provides a comparative overview of common algorithmic strategies, highlighting their optimal use cases and representative example problems.

Strategy	Key Features	Best For	Example Problem
Brute Force	Exhaustive search	Small input size, correctness	Subset Sum
Divide and Conquer	Divide → Solve → Combine	Recursive structures	Merge Sort, Binary Search
Dynamic Programming	Memoization & tabulation	Overlapping subproblems optimization	Knapsack[a], LCS
Greedy	Local optimal decisions	Problems with greedy structure	Huffman[b], Activity Selection[b]

[a] Refer to section 2.2.2.
[b] Refer to section 2.3.3.

Table 1: Comparison of Algorithmic Strategies

2 Application of Advanced Techniques

2.1 Divide and Conquer

2.1.1 Master Theorem

Used to analyze time complexity of divide and conquer algorithms. For recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Compare $f(n)$ with $n^{\log_b a}$ to determine the time complexity.

2.1.2 Examples

- **Merge Sort**: $T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$
- **Binary Search**: $T(n) = T(n/2) + O(1) \Rightarrow O(\log n)$

2.2 Dynamic Programming (DP)

2.2.1 Key Approaches

- **Top-down**: Memoization.
- **Bottom-up**: Tabulation.

2.2.2 Examples

- **0/1 Knapsack**: Maximizes value under weight constraint using tabulated decisions. Refer to *"0/1 Knapsack Problem"* by W3Schools [W3Schools(2024)] for an exhaustive analysis.

2.3 Greedy Algorithms

2.3.1 Concept

Make the best local (greedy) choice at each step, aiming for a globally optimal solution.

2.3.2 Requirements

- **Greedy-choice property**: A global solution can be reached by local choices.
- **Optimal substructure**: A problem's solution contains optimal solutions to subproblems.

2.3.3 Examples

- **Activity Selection** [GeeksforGeeks(2025)]: Select maximum number of non-overlapping activities.
- **Huffman Coding** [GeeksforGeeks(2020)]: Builds optimal prefix codes for data compression.

3 Complexity Analysis for Advanced Strategies

3.1 Complexity Evaluation

3.1.1 Best Case

- **Definition:** The minimum number of steps an algorithm takes on the most favorable input.
- **Notation:** $\Omega(f(n))$
- **Example:** In Binary Search, the best case is when the target is the middle element: $O(1)$.

3.1.2 Average Case

- **Definition:** The expected number of steps over all possible inputs of size n.
- **Notation:** $\Theta(f(n))$
- **Example:** In Linear Search, the average case is finding the element halfway through the array: $\Theta(n/2) = \Theta(n)$.

3.1.3 Worst Case

- **Definition:** The maximum number of steps an algorithm takes on the least favorable input.
- **Notation:** $O(f(n))$
- **Example:** Quick Sort can degrade to $O(n^2)$ if the pivot is always the smallest or largest element.

3.2 Recognizing Patterns in Algorithms

3.2.1 Divide and Conquer

- Recurrences of the form $T(n) = aT(n/b) + f(n)$.
- Solved using the **Master Theorem**.
- Appears in Merge Sort, Binary Search.

3.2.2 Dynamic Programming

- Involves overlapping subproblems and optimal substructure.
- Time complexity often $O(n^2)$ or $O(n \cdot W)$, as in LCS or Knapsack.

3.2.3 Greedy Algorithms

- Typically linear or $O(n \log n)$, especially when sorting is required.
- Complexity depends on the nature of local decisions.

3.3 Summary Table

Case Type	Notation	Description	Example
Best Case	$\Omega(f(n))$	Fastest possible scenario	Binary Search $\rightarrow O(1)$
Average Case	$\Theta(f(n))$	Expected behavior	Linear Search $\rightarrow \Theta(n)$
Worst Case	$O(f(n))$	Slowest possible scenario	Quick Sort $\rightarrow O(n^2)$

In the following link, you can download and listen to a Gemini podcast that discusses the main topics covered in this study guide. Please note that the podcast is in Spanish.

Conclusion

Understanding the complexity of an algorithm is just as crucial as implementing it correctly. Through best-case, average-case, and worst-case analyses, one can gain a comprehensive view of an algorithm's behavior across various inputs. Recognizing structural patterns in problems—such as optimal substructure or overlapping subproblems—enables the informed use of divide and conquer, dynamic programming, or greedy approaches.

Ultimately, the ability to match problems with effective algorithmic strategies fosters both problem-solving efficiency and deeper computational thinking. This guide serves not only as a review for the upcoming exam but also as a reference for future courses and professional scenarios where algorithmic decisions must be made under constraints of time and performance.

References

[GeeksforGeeks(2020)] GeeksforGeeks. Huffman coding/greedy algorithm, 2020. URL https://www.youtube.com/watch?v=0kNXhFIEd_w. Accessed: 2025-05-29.

[GeeksforGeeks(2025)] GeeksforGeeks. Activity selection problem/greedy algo-1, 2025. URL https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/. Accessed: 2025-05-29.

[W3Schools(2024)] W3Schools. 0/1 knapsack problem, 2024. URL https://www.w3schools.com/dsa/dsa_ref_knapsack.php. Accessed: 2025-05-29.