1. Fibonacci

The Fibonacci series is a fundamental concept in computer science and mathematics, where each number is the sum of the two preceding ones. The sequence starts from 0 and 1 and continues infinitely. It finds applications in various fields like algorithm analysis, nature's patterns, and financial modeling due to its recursive mathematical structure.

The recursive approach to generating Fibonacci numbers uses the concept of function calling itself repeatedly until the base condition is reached. Though it is simple to implement and closer to the mathematical definition, it suffers from redundant calculations and high time complexity. Every function call generates two more calls, leading to an exponential growth of computation.

The iterative approach, on the other hand, solves the same problem using loops and variable updates. This method eliminates redundant calculations and makes the algorithm efficient in terms of both time and memory. It runs in linear time and uses constant space. Comparing both methods helps in understanding the importance of algorithmic optimization in problem-solving.

The Fibonacci series also demonstrates how recursion can be replaced with iteration for better efficiency. It forms a foundation for advanced concepts like dynamic programming and memoization, which optimize recursive solutions by storing intermediate results. Understanding both methods is essential for analyzing computational performance and memory management.

2. 0/1 Knapsack (DP & BB)

The 0/1 Knapsack problem is an essential optimization problem in computer science, particularly in operations research and combinatorial optimization. The objective is to maximize the total profit by selecting items without exceeding a fixed capacity. Each item has a specific value and weight, and we must decide whether to include or exclude it entirely.

The dynamic programming approach divides the main problem into smaller subproblems. It stores results in a two-dimensional table, preventing redundant computations. This approach ensures that the solution is computed optimally in polynomial time. It is especially useful when dealing with medium-sized input sets where brute force becomes impractical.

The branch and bound method represents the problem as a decision tree where each node corresponds to a partial solution. By calculating upper and lower bounds, the algorithm eliminates paths that cannot yield better results. This pruning technique significantly reduces the number of computations required to find the optimal solution.

Both dynamic programming and branch & bound provide different perspectives on problem-solving. While DP guarantees optimality through bottom-up computation, branch and bound introduces an intelligent search technique with pruning. Together, they highlight efficiency trade-offs between memory usage and search optimization.

3. N-Queens (Backtracking)

The N-Queens problem is a classic example of the backtracking technique. It involves placing N queens on an N×N chessboard such that no two queens attack each other horizontally, vertically, or diagonally. The challenge lies in systematically exploring all possible configurations until a valid one is found.

Backtracking is a refined brute-force approach that builds the solution step by step. If placing a queen violates the constraints, the algorithm "backtracks" to the previous step and tries another position. This recursive search ensures that all possible configurations are explored without redundancy. It efficiently prunes infeasible paths.

The algorithm uses recursion to place one queen in each row and checks for safety before proceeding. When a conflict is detected, the queen is removed, and the program attempts the next column. This strategy allows the algorithm to efficiently navigate the search space with minimal repeated checks.

The N-Queens problem demonstrates how backtracking provides elegant solutions to constraint satisfaction problems. It is also a good model for solving other puzzles like Sudoku, graph coloring, and maze-solving, emphasizing logic-based problem-solving techniques in artificial intelligence.

4. Huffman Coding

Huffman Encoding is an optimal lossless data compression algorithm based on frequency analysis of symbols. It assigns shorter binary codes to more frequent characters and longer codes to less frequent ones. This technique reduces the total number of bits required to represent the same data, achieving efficient storage.

The algorithm starts by counting the frequency of each character in the input string. These frequencies are used to build a binary tree called the Huffman Tree. The two least frequent nodes are repeatedly merged until a single root node remains, representing the complete encoding structure.

By traversing the tree from the root to each leaf, unique prefix-free binary codes are generated. This ensures that no code is a prefix of another, enabling unambiguous decoding. The result is a highly compact bitstream representing the original text with minimal space usage.

Huffman encoding has numerous applications in compression formats like ZIP, JPEG, and MP3. It exemplifies the greedy algorithm paradigm, as it makes locally optimal choices that lead to a globally optimal solution. Understanding its working mechanism provides insight into data communication and efficient encoding methods.

5. Fractional Knapsack

The Fractional Knapsack problem is one of the most popular examples of the Greedy algorithmic paradigm, widely used in optimization and resource allocation problems. The main objective is to maximize the total profit that can be accommodated in a knapsack of limited capacity. Unlike the 0/1 Knapsack, where items must be taken entirely or left out, this version allows fractional quantities of items to be included for achieving the optimal result.

The algorithm works by calculating the value-to-weight ratio (profit per unit weight) for each item. The items are then sorted in descending order of this ratio, ensuring that the most valuable items per unit weight are selected first. This step-by-step greedy selection guarantees that each decision locally maximizes the value, eventually leading to the globally optimal solution for the problem.

The implementation begins by taking input for the total number of items, their individual weights and values, and the capacity of the knapsack. The program then continuously selects items starting from the one with the highest ratio. If an entire item cannot fit due to capacity limits, only a fraction of it is included. This ensures the knapsack is completely utilized while maximizing the total accumulated value.

The Fractional Knapsack problem demonstrates the power of greedy algorithms, which make decisions based on immediate benefits without considering future consequences. Despite its simplicity, it achieves an optimal solution efficiently in *O(n log n)* time due to sorting. This approach is highly useful in fields like resource allocation, load balancing, and financial portfolio optimization, where partial inclusion of elements is allowed to achieve maximum utility.

Practical	**Algorithm Type**	**Time Complexity**	**Space Complexity**
Fibonacci (Recursive)	**Recursion**	**O(2^n)**	**O(n)**
Fibonacci (Iterative)	**Iterative**	**O(n)**	**O(1)**

0/1 Knapsack (DP)	**Dynamic Programming**	**O(n * W)**	**O(n * W)**
0/1 Knapsack (Branch & Bound)	**Branch & Bound**	**O(2^n) (worst)**	**O(n)**
Fractional Knapsack	**Greedy**	**O(n log n)**	**O(n)**
Huffman Coding	**Greedy + Tree**	**O(n log n)**	**O(n)**
N-Queens	**Backtracking**	**O(N!)**	**O(N^2)**