A Systematic Guide to Algorithmic Patterns: Backtracking and Dynamic Programming

Introduction: A Pattern-Based Framework for Algorithmic Mastery

The landscape of algorithmic problem-solving, particularly in competitive programming and technical interviews, is vast. With platforms listing over 200 backtracking problems and more than 700 dynamic programming challenges, a strategy based on memorizing individual solutions is both impractical and ineffective.[1] The key to navigating this complexity lies not in volume, but in recognizing that this large set of problems is built upon a small, core set of recurring patterns. Mastering these foundational patterns provides a scalable framework for deconstructing and solving a majority of seemingly unique challenges.[3]

This report provides an expert-level analysis of 11 such patterns, focusing on the closely related paradigms of Backtracking and Dynamic Programming. These techniques exist on a logical continuum that begins with simple recursion and progresses toward highly optimized solutions.

At its base is **recursion**, the strategy of solving a problem by breaking it into smaller, self-similar subproblems.[6] When a problem requires exploring a multitude of possible solutions, a brute-force recursive approach can be employed. However, this is often inefficient.

Backtracking emerges as an intelligent optimization of this brute-force search. It is a recursive technique that incrementally builds candidates for a solution and abandons a path ("backtracks") as soon as it determines the current path cannot possibly lead to a valid outcome.[3] This "pruning" of the search space makes it far more efficient than blind exploration. Backtracking is typically employed when the goal is to generate

all possible valid solutions to a problem.[6]

Dynamic Programming (DP) represents a further level of optimization, applicable to a specific subset of problems that exhibit two key properties: optimal substructure and overlapping subproblems.[2]

- **Optimal Substructure:** The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.
- **Overlapping Subproblems:** The same subproblems are solved repeatedly during the recursive process.

DP avoids this redundant computation by storing the results of subproblems in a cache (memoization) or by iteratively building up solutions in a table (tabulation). Consequently, DP is the method of choice when the objective is to find a *single optimal solution* (e.g., the maximum value, minimum cost, or total number of ways) rather than enumerating all possible solutions.[9]

Understanding this progression—from general recursion to pruned searching with Backtracking, to optimized computation with Dynamic Programming—provides a powerful mental model. The primary heuristic for distinguishing between these techniques often boils down to a simple question derived from the problem statement: "Am I being asked to find *all* valid configurations, or the *single best* one?" The former points toward Backtracking, the latter toward Dynamic Programming.

Part I: The Art of State-Space Exploration — Backtracking Patterns

Backtracking is a methodical, depth-first search algorithm that explores the solution space of a problem by constructing a solution incrementally.[8] It is fundamentally a form of organized brute force, where the search is pruned to avoid exploring paths that violate the problem's constraints.[3]

The entire set of possible states (partial and complete solutions) can be visualized as a **state-space tree**.[13] The algorithm starts at the root and explores a path downwards, making a choice at each level. If a path leads to a state that violates a constraint or is a dead end, the algorithm "backtracks" to the parent node and explores an alternative choice.[14]

This process is elegantly captured by a universal recursive template known as the **"Choose, Explore, Unchoose"** pattern. This template forms the backbone of virtually all backtracking

solutions.[13]

C++

```
void backtrack(State& state, const Choices& choices, Results& results) {
    // Base case: check for a valid or complete solution
    if (is_a_solution(state)) {
        add_to_results(state, results);
        return;
    }

    // Iterate through all possible choices at the current state
    for (const auto& choice : choices) {
        if (is_valid(choice)) {
            make_choice(state, choice);      // CHOOSE
            backtrack(state, new_choices, results); // EXPLORE
            undo_choice(state, choice);      // UNCHOOSE (Backtrack)
        }
    }
}
```

The three primary archetypes of backtracking problems—Subsets/Combinations, Permutations, and Constraint Satisfaction—can all be implemented using variations of this core template.

Pattern Name	Core Task	Key Signal Words	Canonical Example	Typical Time Complexity
Subsets & Combinations	Generating all possible groupings where order does not matter.	"subsets", "combinations", "powerset", "choose k"	LeetCode 78. Subsets	O(N·2N)
Permutations	Generating all possible orderings or	"permutations", "arrangements	LeetCode 46. Permutations	O(N·N!)

	arrangements where order matters.	", "orderings", "sequence"		
Constraint Satisfaction	Finding valid configurations or placements that adhere to a set of rules.	"solve", "valid placement", "can place", "satisfy"	N-Queens Problem	O(N!) or O(kN) (heavily pruned)

1.1 Pattern 1: Subsets and Combinations (Enumeration)

This pattern is employed for problems that require the generation of all possible subsets (the powerset) or combinations from a given collection of elements. The defining characteristic is that the order of elements within a generated group is irrelevant; {1, 2} is the same as {2, 1}. The fundamental decision for each element in the input set is binary: it is either included in the current subset, or it is not.[3]

When to Use This Pattern

This pattern should be used when a problem statement explicitly asks for:

- "all possible subsets" or "the powerset".[19]
- "all combinations" of a certain size or that sum to a target.[3]
- Any scenario involving the selection of a group of items where the order of selection has no significance.

Canonical problems include LeetCode 78. Subsets, LeetCode 90. Subsets II (with duplicates), and LeetCode 39. Combination Sum.[3]

C++ Implementation and Explanation (LeetCode 78. Subsets)

The standard implementation uses a recursive helper function that builds the subsets. The key to avoiding duplicate combinations is to control the search space by passing the next starting index to subsequent recursive calls.

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

class Solution {
public:
    std::vector<std::vector<int>> subsets(std::vector<int>& nums) {
        std::vector<std::vector<int>> result;
        std::vector<int> current_subset;
        generate(0, nums, current_subset, result);
        return result;
    }

private:
    void generate(int index, std::vector<int>& nums, std::vector<int>& current_subset,
std::vector<std::vector<int>>& result) {
        // 1. Add the current subset to the result list.
        // Every path in the recursion tree represents a valid subset.
        result.push_back(current_subset);

        // 2. Iterate through the available choices and explore.
        for (int i = index; i < nums.size(); ++i) {
            // CHOOSE: Include the element nums[i] in the current subset.
            current_subset.push_back(nums[i]);

            // EXPLORE: Recurse to generate subsets starting from the next index.
            // Passing `i + 1` ensures that we don't reuse the same element
            // and prevents generating permutations.
            generate(i + 1, nums, current_subset, result);

            // UNCHOOSE (Backtrack): Remove the element to explore subsets that do not include it.
            current_subset.pop_back();
        }
    }
```

```
};
```

Explanation:

1. **subsets(nums):** This public method initializes the result list and a current_subset vector to build a path in the recursion. It initiates the backtracking process by calling the generate helper function, starting at index 0.
2. **generate(...):** This is the core recursive function.
3. **Add to Result:** The first action is result.push_back(current_subset). This is because every state reached during the recursion, including the initial empty set, is a valid subset.
4. **The Loop (Choose/Explore/Unchoose):** The for loop iterates from the index passed into the function. This is the crucial mechanism for generating combinations rather than permutations.
    - **Choose:** current_subset.push_back(nums[i]) adds the selected element to the current path.
    - **Explore:** generate(i + 1,...) makes the recursive call. By passing i + 1, the next level of recursion is constrained to only consider elements at or after the current one, thus preventing duplicates like {2, 1} when {1, 2} has already been formed.
    - **Unchoose:** current_subset.pop_back() is the backtrack step. After the recursive call returns (meaning all subsets starting with the current prefix have been explored), the element is removed. This allows the loop to proceed to the next element, i+1, to form new subsets (e.g., moving from exploring subsets starting with {1, 2} to those starting with {1, 3}).

An alternative mental model for this problem is the "Include/Exclude" decision tree. For each element, a binary choice is made: either include it in the subset and recurse, or exclude it and recurse. Both models produce the same result and are equally valid ways to conceptualize the problem.[17]

Complexity Analysis

- **Time Complexity:** O(N·2N). There are 2N possible subsets for a set of size N. For each of these subsets, creating a copy to add to the result list takes up to O(N) time.[18]
- **Space Complexity:** O(N). This is determined by the maximum depth of the recursion stack, which is N. The space for the output is not typically included in this analysis.

1.2 Pattern 2: Permutations (Arrangement)

This pattern addresses problems that require generating all possible orderings or arrangements of a given set of elements. Unlike subsets, in permutations, the order of elements is paramount; is distinct from. The core task is to place each element from the available set into each available position.[13]

When to Use This Pattern

This pattern is appropriate when a problem statement asks for:

- "all possible permutations" or "all arrangements".[22]
- Solutions to problems involving sequencing, scheduling, or ordering where the sequence itself is the solution.

Canonical problems include LeetCode 46. Permutations and LeetCode 47. Permutations II (with duplicates).[23]

C++ Implementation and Explanation (LeetCode 46. Permutations)

A common and efficient technique for generating permutations is to perform in-place swaps within the input array. This method avoids the need for an extra "used" array to track which elements have been placed.

C++

```
#include <iostream>
#include <vector>
#include <algorithm>

class Solution {
public:
    std::vector<std::vector<int>> permute(std::vector<int>& nums) {
        std::vector<std::vector<int>> result;
        generate(0, nums, result);
        return result;
    }
```

```
private:
    void generate(int index, std::vector<int>& nums, std::vector<std::vector<int>>& result) {
        // 1. Base Case: If the index reaches the end of the array,
        // a full permutation has been formed.
        if (index == nums.size()) {
            result.push_back(nums);
            return;
        }

        // 2. Iterate through the available choices and explore.
        // The choices for the `index`-th position are all elements from `index` to the end.
        for (int i = index; i < nums.size(); ++i) {
            // CHOOSE: Place the element nums[i] at the current position `index` by swapping.
            std::swap(nums[index], nums[i]);

            // EXPLORE: Recurse to fill the next position.
            generate(index + 1, nums, result);

            // UNCHOOSE (Backtrack): Swap back to restore the array to its original state
            // for the next iteration of the loop.
            std::swap(nums[index], nums[i]);
        }
    }
};
```

Explanation:

1. **permute(nums):** The entry point that initializes the result list and starts the recursive generation process at index = 0.
2. **generate(...):** The recursive workhorse. The index parameter represents the current position in the permutation that we are trying to fill.
3. **Base Case:** When index == nums.size(), it signifies that all positions from 0 to n-1 have been filled. The current state of the nums vector is a complete, valid permutation, so it is added to the result.
4. **The Loop (Choose/Explore/Unchoose):** The loop iterates from the current index to the end of the array. This loop considers every remaining element as a candidate for the index-th position.
    - **Choose:** std::swap(nums[index], nums[i]) is the selection mechanism. It takes the element at i and places it at the index position.
    - **Explore:** generate(index + 1,...) recursively calls the function to solve the subproblem of permuting the remaining elements for the positions from index + 1 onwards.
    - **Unchoose:** std::swap(nums[index], nums[i]) is the critical backtrack step. It undoes

the initial swap, restoring the array to its state before the choice was made. This ensures that in the next iteration of the loop, the original element at index is back in its place, ready to be swapped with nums[i+1].

The fundamental difference in state management between subsets and permutations is key. Subsets control the choice list by passing a modified starting index (i + 1) to the next recursive call. Permutations, by contrast, modify the entire collection of choices in-place (via swapping) and advance the position to be filled (index + 1), allowing all remaining elements to be considered for the next slot.

Complexity Analysis

- **Time Complexity:** O(N·N!). There are N! permutations for a set of N distinct elements. Each permutation is of length N, and copying it to the result list takes O(N) time.[6]
- **Space Complexity:** O(N). This is for the depth of the recursion stack. The modifications are done in-place, so no extra space proportional to the number of permutations is used besides the output list.

1.3 Pattern 3: Constraint Satisfaction Puzzles

This pattern is applied to problems where the objective is to find one or more configurations that satisfy a complex set of rules or constraints. Unlike the previous patterns, which focus on generating all possibilities, this pattern is about intelligently navigating the search space to find only *valid* solutions. The efficiency of the algorithm is heavily dependent on how effectively it can "prune" branches of the state-space tree that violate constraints.[7]

When to Use This Pattern

This pattern is the go-to for problems that can be modeled as puzzles:

- Placing items on a grid or board with rules about their positions (e.g., N-Queens, Sudoku).
- Finding a path through a maze or grid that adheres to certain conditions.[6]
- Any problem where a solution is built piece by piece, and each piece must be validated

against a set of constraints before proceeding.

Canonical problems include LeetCode 51. N-Queens and LeetCode 37. Sudoku Solver.[7]

C++ Implementation and Explanation (N-Queens Problem)

The N-Queens problem asks for all distinct configurations of placing N chess queens on an N×N board such that no two queens threaten each other. The solution explores placing queens one row at a time.

C++

```
#include <iostream>
#include <vector>
#include <string>

class Solution {
public:
    std::vector<std::vector<std::string>> solveNQueens(int n) {
        std::vector<std::vector<std::string>> result;
        std::vector<std::string> board(n, std::string(n, '.'));
        solve(0, board, result, n);
        return result;
    }

private:
    // Constraint checking function
    bool isSafe(int row, int col, std::vector<std::string>& board, int n) {
        // Check the column upwards
        for (int i = 0; i < row; ++i) {
            if (board[i][col] == 'Q') {
                return false;
            }
        }
        // Check upper-left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; --i, --j) {
            if (board[i][j] == 'Q') {
```

```
            return false;
        }
    }
    // Check upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < n; --i, ++j) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }
    return true;
}

void solve(int row, std::vector<std::string>& board, std::vector<std::vector<std::string>>& result, int n)
{
    // Base Case: If all queens are placed (i.e., we've filled rows 0 to n-1)
    if (row == n) {
        result.push_back(board);
        return;
    }

    // Iterate through all columns in the current row
    for (int col = 0; col < n; ++col) {
        // Check if placing a queen at (row, col) is valid
        if (isSafe(row, col, board, n)) {
            // CHOOSE: Place the queen
            board[row][col] = 'Q';

            // EXPLORE: Recurse to the next row
            solve(row + 1, board, result, n);

            // UNCHOOSE (Backtrack): Remove the queen to explore other possibilities
            board[row][col] = '.';
        }
    }
}
};
```

Explanation:

1. **solveNQueens(n):** Initializes an N×N board represented by a vector of strings and the result list. It starts the backtracking process from the first row (row = 0).
2. **solve(...):** The recursive function that attempts to place queens starting from the given row.

3. **Base Case:** If row == n, it means we have successfully placed a queen in every row from 0 to n-1. A valid solution has been found, so the current board configuration is added to the result.
4. **The Loop:** The function iterates through each column col for the current row, considering each as a potential placement for a queen.
5. **isSafe(...):** This is the crucial constraint-checking and pruning function. Before making a choice, it verifies that placing a queen at (row, col) is safe. It checks for other queens in the same column and on the two upper diagonals. Since we are placing queens row-by-row, we only need to check rows above the current one. If isSafe returns false, this entire branch of the search tree is pruned.
6. **Choose/Explore/Unchoose:**
    - **Choose:** If the position is safe, board[row][col] = 'Q' places the queen.
    - **Explore:** solve(row + 1,...) proceeds to the next row to place the next queen.
    - **Unchoose:** board[row][col] = '.' removes the queen after the recursive call returns. This backtracking step is essential to allow the loop to try placing the queen in the next column of the same row.

The performance of this pattern is dominated by the efficiency of the constraint check. The provided isSafe function takes O(N) time. This check can be optimized to O(1) by using auxiliary data structures (e.g., boolean arrays or hash sets) to keep track of occupied columns and diagonals, significantly improving performance.[26]

Complexity Analysis

- **Time Complexity:** The complexity is difficult to state precisely but is bounded by O(N!). While the state-space tree has NN leaf nodes in a naive search, the pruning from the isSafe function drastically reduces the number of nodes visited. Each valid placement requires an O(N) check in this implementation.
- **Space Complexity:** O(N2) to store the board, plus an additional O(N) for the depth of the recursion stack.

Part II: The Principle of Optimal Substructure — Dynamic Programming Patterns

Dynamic Programming is an algorithmic paradigm for solving optimization problems by breaking them down into simpler, overlapping subproblems. It is applicable only when a

problem exhibits two specific characteristics: optimal substructure and overlapping subproblems.[2] A problem has

optimal substructure if its overall optimal solution can be constructed from the optimal solutions of its subproblems. It has **overlapping subproblems** if a recursive algorithm solves the same subproblems multiple times.

DP tackles this inefficiency by storing the solutions to subproblems, ensuring that each is computed only once. There are two primary strategies for implementing DP:

1. **Top-Down with Memoization:** This approach maintains the natural recursive structure of the solution. A cache (typically an array or hash map) is used to store the results of function calls. Before computing a subproblem, the cache is checked; if the result is present, it is returned directly, avoiding re-computation. This method is often more intuitive to derive from a pure recursive solution.[28]
2. **Bottom-Up with Tabulation:** This approach is iterative and avoids recursion altogether. It builds a table (the dp array or matrix) and fills it "from the bottom up," starting with the smallest subproblems and progressively solving larger ones. The solution to a given subproblem is calculated using the already-computed solutions to smaller subproblems. This method can be more space-efficient and avoids recursion stack depth limits.[28]

The key to solving any DP problem is to correctly identify the **state** (what the dp table entries represent) and the **recurrence relation** (the formula that transitions from one state to another).

Pattern Name	Problem Type	State Definition dp[...]	General Recurrence Relation	Canonical Example
1D DP	Sequence decisions, counting ways	dp[i]: Optimal value/count up to index i	dp[i] = f(dp[i-1], dp[i-2],...)	Climbing Stairs
0/1 Knapsack	Constrained subset selection	dp[i][w]: Max value using first i items with capacity w	dp[i][w] = max(dp[i-1][w], v[i] + dp[i-1][w-wt[i]])	0/1 Knapsack
Unbounded Knapsack	Unconstrained subset selection	dp[i][w]: Max value using first i items	dp[i][w] = max(dp[i-1][w], v[i] +	Coin Change

		with capacity w	dp[i][w-wt[i]])	
Longest Common Subsequence	Comparison of two sequences	dp[i][j]: LCS length of s1[0..i-1] and s2[0..j-1]	dp[i][j] = 1+dp[i-1][j-1] or max(dp[i-1][j], dp[i][j-1])	Longest Common Subsequence
Longest Increasing Subsequence	Ordered subsequence in one sequence	dp[i]: LIS length ending at index i	dp[i] = 1 + max(dp[j]) for j<i and arr[j]<arr[i]	Longest Increasing Subsequence
DP on Grids	Pathfinding in a matrix	dp[i][j]: Optimal value/count to reach cell (i,j)	dp[i][j] = f(dp[i-1][j], dp[i][j-1])	Unique Paths
Kadane's Algorithm	Max sum of a contiguous subarray	max_ending_here: Max sum of subarray ending at i	max_ending_here = max(a[i], a[i] + max_ending_here)	Maximum Subarray
Matrix Chain Multiplication	Optimal partitioning (Interval DP)	dp[i][j]: Optimal value for interval [i, j]	dp[i][j] = min/max(dp[i][k] + dp[k+1][j] + cost)	Matrix Chain Multiplication

2.1 Pattern 1: 1D DP (Sequences and Decisions)

This is the most fundamental DP pattern. The state at any point i depends only on a fixed number of preceding states (e.g., dp[i-1], dp[i-2]). It's commonly applied to problems involving sequences where a decision at each step builds upon previous decisions.

When to Use This Pattern

This pattern is indicated when:

- The problem input is a 1D array or a single integer n.
- The recurrence relation is simple, like f(i)=g(f(i−1),f(i−2)).
- The problem involves counting ways, finding a minimum/maximum cost, or making a simple choice (e.g., rob/don't rob) at each step in a sequence.

Canonical problems include Fibonacci Numbers, LeetCode 70. Climbing Stairs, and LeetCode 198. House Robber.[27]

C++ Implementation and Explanation (LeetCode 70. Climbing Stairs)

The problem asks for the number of distinct ways to climb a staircase of n steps, taking either 1 or 2 steps at a time. The number of ways to reach step i is the sum of the ways to reach step i-1 (and taking one step) and the ways to reach step i-2 (and taking two steps). This gives the recurrence dp[i]=dp[i−1]+dp[i−2], which is the Fibonacci sequence.

Top-Down (Memoization)

C++

```
#include <vector>

class Solution {
public:
    int climbStairs(int n) {
        std::vector<int> memo(n + 1, -1);
        return climb(n, memo);
    }
```

```
private:
    int climb(int n, std::vector<int>& memo) {
        // Base cases
        if (n < 0) return 0;
        if (n == 0) return 1;

        // Check if already computed
        if (memo[n]!= -1) {
            return memo[n];
        }

        // Compute and store the result
        memo[n] = climb(n - 1, memo) + climb(n - 2, memo);
        return memo[n];
    }
};
```

Bottom-Up (Tabulation)

C++

```
#include <vector>

class Solution {
public:
    int climbStairs(int n) {
        if (n <= 1) return 1;

        std::vector<int> dp(n + 1);
        // Base cases
        dp = 1;
        dp = 1;

        // Fill the DP table iteratively
        for (int i = 2; i <= n; ++i) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
```

```
        return dp[n];
    }
};
```

A crucial observation for many 1D DP problems is the potential for space optimization. Since the calculation of dp[i] only depends on dp[i-1] and dp[i-2], the entire dp array is not necessary. We only need to store the last two values, reducing space complexity from O(N) to O(1).[32]

C++

```
// Space-Optimized Bottom-Up
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 1) return 1;
        int prev2 = 1; // Corresponds to dp[i-2]
        int prev1 = 1; // Corresponds to dp[i-1]
        for (int i = 2; i <= n; ++i) {
            int current = prev1 + prev2;
            prev2 = prev1;
            prev1 = current;
        }
        return prev1;
    }
};
```

Complexity Analysis

- **Time Complexity:** O(N) for all DP approaches.
- **Space Complexity:** O(N) for the standard Top-Down and Bottom-Up versions. O(1) for the space-optimized version.

2.2 Pattern 2: 0/1 Knapsack (Constrained Choice)

This pattern solves problems where one must select a subset of items, each with associated properties (like weight and value), to maximize a total value while adhering to a capacity constraint. Each item can be chosen at most once—it is either taken (1) or not taken (0).[33]

When to Use This Pattern

Apply the 0/1 Knapsack pattern when:

- The problem involves choosing a subset of items.
- Each item has two competing metrics (e.g., cost vs. value, weight vs. profit).
- The goal is to optimize one metric (maximize profit) subject to a constraint on the other (total weight ≤ capacity).
- Items cannot be selected more than once.

This pattern is versatile and can be adapted. For example, in LeetCode 416. Partition Equal Subset Sum, the "items" are numbers, "weight" is the number's value, and "capacity" is half the total sum. The goal is to see if a value of exactly the capacity can be achieved.[35]

C++ Implementation and Explanation

The state dp[i][w] represents the maximum value achievable using the first i items with a knapsack capacity of w.

Bottom-Up (Tabulation)

C++

```
#include <vector>
#include <algorithm>
```

```cpp
int knapsack(int W, const std::vector<int>& wt, const std::vector<int>& val, int n) {
    // dp[i][w] will be the maximum value that can be obtained
    // using first i items and capacity w.
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= W; ++w) {
            // If the current item's weight is more than the current capacity,
            // we cannot include it. So, the value is the same as without this item.
            if (wt[i - 1] > w) {
                dp[i][w] = dp[i - 1][w];
            } else {
                // We have two choices:
                // 1. Don't include the item: value is dp[i-1][w]
                // 2. Include the item: value is val[i-1] + value from remaining capacity
                dp[i][w] = std::max(dp[i - 1][w], val[i - 1] + dp[i - 1][w - wt[i - 1]]);
            }
        }
    }
    return dp[n];
}
```

Explanation:
The algorithm iterates through each item i and each possible capacity w. For each (i, w) pair, it decides whether including item i yields a better result than excluding it.

- **dp[i-1][w]:** Represents the choice to *exclude* the current item (i-1 in 0-indexed arrays). The max value is simply what could be achieved with the previous i-1 items at the same capacity w.
- val[i-1] + dp[i-1][w - wt[i-1]]: Represents the choice to include the current item. This is only possible if its weight wt[i-1] does not exceed the current capacity w. The value is the item's own value plus the max value achievable with the previous i-1 items and the remaining capacity.
  The std::max function makes the optimal choice for dp[i][w].

Complexity Analysis

- **Time Complexity:** O(N·W), where N is the number of items and W is the knapsack capacity.
- **Space Complexity:** O(N·W). This can be optimized to O(W) because computing row i only requires values from row i-1.

2.3 Pattern 3: Unbounded Knapsack (Unconstrained Choice)

This is a variation of the knapsack problem where each item is available in unlimited quantities. The objective remains the same: maximize total value within a given capacity.[36]

When to Use This Pattern

Use this pattern under the same conditions as 0/1 Knapsack, but when the problem statement indicates that items can be reused. Signal phrases include "unlimited supply," "repetition is allowed," or problems like Coin Change where you can use multiple coins of the same denomination.[35]

C++ Implementation and Explanation

The implementation is remarkably similar to the 0/1 version, with one subtle but critical change in the recurrence relation.

Bottom-Up (Tabulation)

C++

```cpp
#include <vector>
#include <algorithm>

int unboundedKnapsack(int W, const std::vector<int>& wt, const std::vector<int>& val, int n) {
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(W + 1, 0));
```

```
    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= W; ++w) {
            if (wt[i - 1] > w) {
                dp[i][w] = dp[i - 1][w];
            } else {
                // The only change is here: dp[i][w - wt[i-1]] instead of dp[i-1][...].
                // This allows the current item `i` to be considered again.
                dp[i][w] = std::max(dp[i - 1][w], val[i - 1] + dp[i][w - wt[i - 1]]);
            }
        }
    }
    return dp[n];
}
```

Explanation:
The key change is in the "include" case. Instead of dp[i-1][w - wt[i-1]], we use dp[i][w - wt[i-1]]. This means that when we decide to take item i, the subproblem we solve for the remaining capacity still has access to the full set of items up to i, including item i itself. This allows for the repeated selection of the same item.
This subtle difference has an important implication for the space-optimized O(W) solution. For 0/1 Knapsack, the inner capacity loop must run from W down to 0 to ensure that dp[w - wt[i-1]] refers to the value from the previous row (i-1). For Unbounded Knapsack, the loop must run from 0 up to W, which allows dp[w - wt[i-1]] to use a value that may have already been updated in the current row (i), correctly modeling the reuse of an item.[36]

Complexity Analysis

- **Time Complexity:** O(N·W).
- **Space Complexity:** O(N·W), optimizable to O(W).

2.4 Pattern 4: Longest Common Subsequence (LCS)

This pattern is used to find the length of the longest subsequence that is present in two given sequences (e.g., strings). The elements of a subsequence must maintain their relative order but do not need to be contiguous.[38]

When to Use This Pattern

This pattern is fundamental for problems involving the comparison of two sequences:

- Directly asking for the "longest common subsequence."
- Problems that can be framed in terms of LCS, such as finding the minimum number of insertions/deletions to transform one string into another (related to Edit Distance).
- Applications in bioinformatics for DNA sequence alignment and in version control systems (like diff).[39]

Canonical problems include LeetCode 1143. Longest Common Subsequence and LeetCode 72. Edit Distance.[35]

C++ Implementation and Explanation

The state dp[i][j] stores the length of the LCS between the prefixes text1[0...i-1] and text2[0...j-1].

Bottom-Up (Tabulation)

C++

```cpp
#include <string>
#include <vector>
#include <algorithm>

int longestCommonSubsequence(std::string text1, std::string text2) {
    int m = text1.length();
    int n = text2.length();
    std::vector<std::vector<int>> dp(m + 1, std::vector<int>(n + 1, 0));
```

```
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            // If characters match, the LCS is 1 + LCS of the prefixes before these characters.
            if (text1[i - 1] == text2[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            } else {
                // If they don't match, the LCS is the best of what we can get by
                // either ignoring the current char of text1 or the current char of text2.
                dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[m][n];
}
```

Explanation:
The algorithm builds a 2D table. For each pair of characters text1[i-1] and text2[j-1]:

- If they **match**, we know they can be part of the common subsequence. The length of the LCS is thus one greater than the LCS of the strings without these characters, which is found at dp[i-1][j-1].
- If they **do not match**, we cannot use both characters. The longest common subsequence must be the longer of two possibilities: the LCS of text1[0...i-2] and text2[0...j-1] (stored in dp[i-1][j]), or the LCS of text1[0...i-1] and text2[0...j-2] (stored in dp[i][j-1]).

Complexity Analysis

- **Time Complexity:** O(M·N), where M and N are the lengths of the two strings.
- **Space Complexity:** O(M·N). This can be optimized to O(min(M,N)).

2.5 Pattern 5: Longest Increasing Subsequence (LIS)

This pattern is used to find the length of the longest subsequence within a single given sequence where all elements of the subsequence are in strictly increasing order.[41]

When to Use This Pattern

This pattern is applicable when:

- The problem involves a single sequence (array).
- The goal is to find the longest subsequence that adheres to an ordering property (e.g., strictly increasing, non-decreasing).
- It forms the basis for more complex problems like finding the maximum sum increasing subsequence or the number of increasing subsequences.

Canonical problems include LeetCode 300. Longest Increasing Subsequence and LeetCode 354. Russian Doll Envelopes, which can be reduced to an LIS problem.[35]

C++ Implementation and Explanation

There are two well-known DP solutions for LIS with different time complexities.

O(N^2) Bottom-Up DP

This is the standard, more intuitive DP approach. The state dp[i] represents the length of the LIS that *ends* at index i.

C++

```
#include <vector>
#include <algorithm>

int lengthOfLIS_N2(std::vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    std::vector<int> dp(n, 1);
    int max_length = 1;

    for (int i = 1; i < n; ++i) {
```

```
        for (int j = 0; j < i; ++j) {
            // If nums[i] can extend the subsequence ending at j
            if (nums[i] > nums[j]) {
                dp[i] = std::max(dp[i], 1 + dp[j]);
            }
        }
        max_length = std::max(max_length, dp[i]);
    }
    return max_length;
}
```

Explanation:
The outer loop iterates through each element nums[i] as a potential end of an increasing subsequence. The inner loop checks all previous elements nums[j]. If nums[i] is greater than nums[j], it means nums[i] can be appended to the increasing subsequence that ends at j. We update dp[i] to be the maximum length achievable by extending any of the valid previous subsequences. The overall LIS is the maximum value found in the dp array.

O(N log N) Optimized Approach

This more advanced solution uses a greedy approach with binary search. It maintains a sorted list (often called tails or piles) which represents the smallest possible tail element for an increasing subsequence of a certain length.

C++

```
#include <vector>
#include <algorithm>

int lengthOfLIS_NLogN(std::vector<int>& nums) {
    std::vector<int> tails;
    for (int num : nums) {
        // Find the first element in tails that is not less than num
        auto it = std::lower_bound(tails.begin(), tails.end(), num);

        if (it == tails.end()) {
            // num is greater than all elements in tails, so it extends the LIS.
            tails.push_back(num);
```

```
        } else {
            // Replace the element at `it` with num. This creates a new
            // potential for a longer LIS with a smaller tail element.
            *it = num;
        }
    }
    return tails.size();
}
```

Explanation:
This algorithm builds the LIS in a different way. The tails vector does not store the LIS itself, but rather stores candidate tail elements. For each number num from the input, we find its correct position in the sorted tails vector. If num is larger than any element in tails, it extends the longest subsequence found so far, and we append it. If num is smaller, we find the smallest element in tails that is greater than or equal to num and replace it. This replacement is key: it doesn't change the length of the existing subsequences but lowers the tail value, creating more opportunities for subsequent elements to extend them. The final size of the tails vector is the length of the LIS.42

Complexity Analysis

- **Standard DP:** Time complexity is O(N2), space complexity is O(N).
- **Optimized Approach:** Time complexity is O(NlogN) due to the binary search (lower_bound) at each step. Space complexity is O(N) for the tails vector.

2.6 Pattern 6: DP on Grids (Matrix Pathfinding)

This pattern applies to problems set on a 2D grid where the goal is to find an optimal path from a starting cell (e.g., top-left) to a destination cell (e.g., bottom-right). The optimization can be counting the number of unique paths or finding a path with the minimum or maximum sum, given constraints on movement (typically only right and down).[43]

When to Use This Pattern

Use this pattern when:

- The problem is defined on a 2D matrix or grid.
- The task involves finding a path from one corner to another.
- Movement is restricted (e.g., only right and down moves are allowed).
- The goal is to count paths or find a path with an optimal (min/max) value.

Canonical problems include LeetCode 62. Unique Paths, LeetCode 63. Unique Paths II (with obstacles), and LeetCode 64. Minimum Path Sum.[45]

C++ Implementation and Explanation (LeetCode 62. Unique Paths)

The state dp[i][j] will store the number of unique paths to reach cell (i, j).

Bottom-Up (Tabulation)

C++

```cpp
#include <vector>

int uniquePaths(int m, int n) {
    // dp[i][j] will store the number of paths to reach cell (i, j)
    std::vector<std::vector<int>> dp(m, std::vector<int>(n, 0));

    // There is only one way to reach any cell in the first row (by moving right)
    for (int j = 0; j < n; ++j) {
        dp[j] = 1;
    }
    // There is only one way to reach any cell in the first column (by moving down)
    for (int i = 0; i < m; ++i) {
        dp[i] = 1;
    }

    // Fill the rest of the grid
    for (int i = 1; i < m; ++i) {
```

```
        for (int j = 1; j < n; ++j) {
            // Paths to (i,j) = paths from above + paths from left
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
```

Explanation:
The number of ways to reach any cell (i, j) is the sum of the number of ways to reach the cell directly above it, (i-1, j), and the number of ways to reach the cell directly to its left, (i, j-1). The base cases are the cells in the first row and first column, for which there is only one path from the start. The algorithm iteratively fills the dp table using this recurrence relation.

Complexity Analysis

- **Time Complexity:** O(M·N) for traversing the grid.
- **Space Complexity:** O(M·N). This can be optimized to O(N) since each row's calculation only depends on the previous row.

2.7 Pattern 7: Kadane's Algorithm (Maximum Subarray Sum)

Kadane's Algorithm is a highly efficient, specialized 1D DP algorithm. It solves the problem of finding the contiguous subarray within a one-dimensional array that has the largest possible sum. It is a classic example of how a DP state can be simplified to just a few variables instead of a full array.[47]

When to Use This Pattern

This algorithm should be the immediate choice when:

- A problem asks for the maximum (or minimum) sum of a *contiguous* subarray.
- The input is a 1D array containing positive and negative numbers.

Canonical problems are LeetCode 53. Maximum Subarray and its variations like LeetCode 152.

Maximum Product Subarray.[35]

C++ Implementation and Explanation

The algorithm's DP state is implicit. It maintains two variables: the maximum sum ending at the current position, and the global maximum sum found so far.

C++

```cpp
#include <vector>
#include <algorithm>

int maxSubArray(std::vector<int>& nums) {
    if (nums.empty()) return 0;

    int max_so_far = nums;
    int max_ending_here = nums;

    for (size_t i = 1; i < nums.size(); ++i) {
        // The max subarray ending at `i` is either the element `nums[i]` itself,
        // or `nums[i]` combined with the max subarray ending at `i-1`.
        max_ending_here = std::max(nums[i], max_ending_here + nums[i]);

        // Update the global maximum if the new `max_ending_here` is larger.
        max_so_far = std::max(max_so_far, max_ending_here);
    }
    return max_so_far;
}
```

Explanation:
The algorithm iterates through the array once. At each position i, it makes a decision:

1. Does the maximum subarray ending at the previous position (max_ending_here) have a positive sum? If so, adding the current element nums[i] will create a larger subarray sum.
2. Is the max_ending_here from the previous position negative? If so, it would detract from nums[i]. In this case, it is better to start a new subarray beginning with nums[i].
   The line max_ending_here = std::max(nums[i], max_ending_here + nums[i]); elegantly captures this logic. The max_so_far variable simply keeps track of the largest

max_ending_here value seen during the traversal.

Complexity Analysis

- **Time Complexity:** O(N), as it requires a single pass through the array.
- **Space Complexity:** O(1), as it only uses a few variables for storage.

2.8 Pattern 8: Matrix Chain Multiplication (Interval DP)

This is an advanced DP pattern used for problems that involve finding an optimal solution by breaking a sequence down into sub-intervals. The solution for an interval [i, j] is typically found by iterating through all possible split points k within that interval and combining the optimal solutions for the sub-intervals [i, k] and [k+1, j].[49]

When to Use This Pattern

This pattern is suited for problems where:

- The problem involves finding an optimal value over a sequence or interval (e.g., minimum cost, maximum coins).
- The solution requires making a series of decisions that can be modeled as parenthesizing or partitioning the sequence.
- The recurrence relation has the form dp[i][j]=mini≤k<j{dp[i][k]+dp[k+1][j]+cost(i,k,j)}.

Canonical problems include the classic Matrix Chain Multiplication, LeetCode 312. Burst Balloons, and LeetCode 1039. Minimum Score Triangulation of Polygon.[35]

C++ Implementation and Explanation

The state dp[i][j] represents the minimum cost to multiply the chain of matrices from index i to j. The input p is an array of dimensions, where matrix Ai has dimensions p[i−1]×p[i].

Bottom-Up (Tabulation)

C++

```cpp
#include <vector>
#include <climits>
#include <algorithm>

int matrixChainMultiplication(const std::vector<int>& p) {
    int n = p.size() - 1; // Number of matrices
    if (n <= 1) return 0;

    std::vector<std::vector<int>> dp(n, std::vector<int>(n, 0));

    // len is the length of the matrix chain being considered
    for (int len = 2; len <= n; ++len) {
        for (int i = 0; i <= n - len; ++i) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            // k is the split point
            for (int k = i; k < j; ++k) {
                // Cost to multiply sub-chains (i,k) and (k+1,j) plus the cost
                // of multiplying the two resulting matrices.
                // Result of (i,k) is p[i] x p[k+1]
                // Result of (k+1,j) is p[k+1] x p[j+1]
                int cost = dp[i][k] + dp[k + 1][j] + p[i] * p[k + 1] * p[j + 1];
                dp[i][j] = std::min(dp[i][j], cost);
            }
        }
    }
    return dp[n - 1];
}
```

Explanation:
The algorithm works by solving for progressively larger chain lengths.

1. The base cases (chain length 1, i.e., dp[i][i]) have a cost of 0, as no multiplication is needed.

2. The outer loop iterates over the chain len from 2 to N.
3. The middle loop iterates over the starting index i of the chain. The ending index j is determined by i and len.
4. The inner loop iterates through all possible split points k for the chain [i, j]. For each k, it calculates the cost of multiplying the sub-chain [i, k] (cost dp[i][k]), the sub-chain [k+1, j] (cost dp[k+1][j]), and the final multiplication of the two resulting matrices (cost p[i] * p[k+1] * p[j+1]).
5. It stores the minimum cost found among all possible splits k in dp[i][j]. The final answer is in dp[n-1].

Complexity Analysis

- **Time Complexity:** O(N3) due to the three nested loops (for len, i, and k).
- **Space Complexity:** O(N2) for the DP table.

Conclusion: Synthesizing and Applying Algorithmic Patterns

The transition from solving individual algorithmic problems to developing a systematic, pattern-based approach is a critical step toward mastery. The Backtracking and Dynamic Programming patterns detailed in this report represent foundational frameworks that govern the solutions to hundreds of problems.

The primary heuristic for initial problem classification remains the distinction in the objective:

- **Backtracking** is the tool for **enumeration problems**, where the goal is to generate *all* possible configurations that satisfy a given set of constraints. The core mechanism is a recursive "Choose, Explore, Unchoose" strategy that prunes invalid search paths.
- **Dynamic Programming** is the tool for **optimization problems**, where the goal is to find a *single best* solution (e.g., maximum, minimum, or total count). It is applicable only when the problem possesses optimal substructure and overlapping subproblems, allowing for efficient computation via memoization or tabulation.

Within these paradigms, recognizing the specific sub-pattern is the next crucial step:

- **1D Input:** If the problem involves decisions on a sequence, consider **1D DP**. If it specifically asks for the maximum sum of a *contiguous* block, **Kadane's Algorithm** is the optimal choice.

- **Subset Selection:** If the problem involves choosing a subset of items with constraints (e.g., weight/value), it is a **Knapsack** problem. The key differentiator is whether items can be reused (**Unbounded**) or not (**0/1**).
- **Sequence Comparison:** Problems involving two sequences often map to **Longest Common Subsequence**. Problems seeking an ordered subsequence within a single sequence point to **Longest Increasing Subsequence**.
- **Grid Traversal:** Pathfinding on a matrix with restricted moves is a classic case for **DP on Grids**.
- **Optimal Partitioning:** Problems that require finding an optimal value by trying all possible split points in a sequence suggest **Matrix Chain Multiplication (Interval DP)**.

Ultimately, these patterns are not rigid recipes but flexible mental models. True proficiency is achieved not by merely implementing these templates, but by understanding the principles that underpin them. This allows for the adaptation and combination of these patterns to deconstruct the novel and complex challenges that define the upper echelon of algorithmic problem-solving.

Works cited

1. C/C++ Backtracking Programs - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/c-cpp-backtracking-programs/
2. Dynamic Programming or DP - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/competitive-programming/dynamic-programming/
3. Leetcode Pattern 3 | Backtracking | by csgator | Leetcode Patterns ..., accessed on September 12, 2025, https://medium.com/leetcode-patterns/leetcode-pattern-3-backtracking-5d9e5a03dc26
4. 10 Top LeetCode Patterns to Crack FAANG Coding Interviews - Design Gurus, accessed on September 12, 2025, https://www.designgurus.io/blog/top-lc-patterns
5. “15 LeetCode Patterns That Changed My DSA Journey” | by Priyanka Daida | Medium, accessed on September 12, 2025, https://medium.com/@priyankadaida/15-leetcode-patterns-that-changed-my-dsa-journey-4a0cd98555f4
6. Introduction to Backtracking - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/introduction-to-backtracking-2/
7. Competitive Programming #4: Backtracking | by Mahedi Hasan ..., accessed on September 12, 2025, https://medium.com/@mahedihasanjisan/competitive-programming-3-backtracking-c9ccb5a1a6ea
8. A Comprehensive Guide on Backtracking Algorithm - Analytics Vidhya, accessed on September 12, 2025, https://www.analyticsvidhya.com/blog/2024/09/backtracking-algorithm/
9. How do I decide between Dynamic Programming vs Backtracking? : r/leetcode -

Reddit, accessed on September 12, 2025, https://www.reddit.com/r/leetcode/comments/ntuycc/how_do_i_decide_between_dynamic_programming_vs/
10. C/C++ Dynamic Programming Programs - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/c-cpp-dynamic-programming-programs/
11. Introduction to Backtracking Pattern - Design Gurus, accessed on September 12, 2025, https://www.designgurus.io/course-play/grokking-the-coding-interview/doc/introduction-to-backtracking-pattern
12. Backtracking, accessed on September 12, 2025, https://jeffe.cs.illinois.edu/teaching/algorithms/book/02-backtracking.pdf
13. Backtracking Algorithm Common Patterns and Code Template | Labuladong Algo Notes, accessed on September 12, 2025, https://labuladong.online/algo/en/essential-technique/backtrack-framework/
14. Your One-Stop Solution to Understand Backtracking Algorithm - Simplilearn.com, accessed on September 12, 2025, https://www.simplilearn.com/tutorials/data-structure-tutorial/backtracking-algorithm
15. Backtracking with C++ - Crack FAANG - Medium, accessed on September 12, 2025, https://crackfaang.medium.com/backtracking-with-c-91e3bfc56a21
16. Understanding Backtracking. An overview | by Aniruddha Karajgi ..., accessed on September 12, 2025, https://polaris000.medium.com/understanding-backtracking-26d512e34b26
17. Recursive Backtracking and Optimization, accessed on September 12, 2025, https://web.stanford.edu/class/cs106b-8/lectures/backtracking-optimization/Lecture13.pdf
18. Print all subsets of a given Set or Array - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/backtracking-to-find-all-subsets/
19. Subsets - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/subsets/
20. Subsets II - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/subsets-ii/
21. Print all Subset for set (Backtracking and Bitmasking approach) - Topcoder, accessed on September 12, 2025, https://www.topcoder.com/thrive/articles/print-all-subset-for-set-backtracking-and-bitmasking-approach
22. 20 Essential Coding Patterns to Ace Your Next Coding Interview - DEV Community, accessed on September 12, 2025, https://dev.to/arslan_ah/20-essential-coding-patterns-to-ace-your-next-coding-interview-32a3
23. Permutations - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/permutations/
24. Permutations II - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/permutations-ii/

25. Recursive Backtracking in c++ - Stack Overflow, accessed on September 12, 2025, https://stackoverflow.com/questions/42811988/recursive-backtracking-in-c
26. N-Queen Problem in C++ - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/n-queen-problem-in-cpp/
27. Dynamic Programming Study Guide : r/leetcode - Reddit, accessed on September 12, 2025, https://www.reddit.com/r/leetcode/comments/yntpp4/dynamic_programming_study_guide/
28. Dynamic programming in C++ - Educative.io, accessed on September 12, 2025, https://www.educative.io/answers/dynamic-programming-in-cpp
29. Mastering Dynamic Programming: Questions and Solutions in C++ | by Utkarsh Gupta, accessed on September 12, 2025, https://medium.com/@utkarsh.gupta0311/mastering-dynamic-programming-questions-and-solutions-in-c-da5672764826
30. Dynamic Programming Pattern, accessed on September 12, 2025, https://patterns.eecs.berkeley.edu/?page_id=416
31. Climbing Stairs - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/climbing-stairs/
32. 70. Climbing Stairs - Solution & Explanation - NeetCode, accessed on September 12, 2025, https://neetcode.io/solutions/climbing-stairs
33. C++ Program to Solve the 0-1 Knapsack Problem - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/cpp-program-to-solve-the-0-1-knapsack-problem/
34. 0/1 Knapsack Problem - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/0-1-knapsack-problem-dp-10/
35. Dynamic programming patterns - Discuss - LeetCode, accessed on September 12, 2025, https://leetcode.com/discuss/interview-question/3099280/Dynamic-programming-patterns
36. 14.5 Unbounded knapsack problem - Hello Algo, accessed on September 12, 2025, https://www.hello-algo.com/en/chapter_dynamic_programming/unbounded_knapsack_problem/
37. Unbounded Knapsack (Repetition of items allowed) - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/unbounded-knapsack-repetition-items-allowed/
38. Longest Common Subsequence - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/longest-common-subsequence/
39. Longest Common Subsequence (LCS) - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/longest-common-subsequence-dp-4/
40. C++ Program for Longest Common Subsequence - GeeksforGeeks, accessed on September 12, 2025,

https://www.geeksforgeeks.org/dsa/cpp-program-for-longest-common-subsequence/
41. Longest Increasing Subsequence - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/longest-increasing-subsequence/
42. Longest Increasing Subsequence (LIS) - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/longest-increasing-subsequence-dp-3/
43. Unique paths in a Grid with Obstacles - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/unique-paths-in-a-grid-with-obstacles/
44. Grid Unique Paths - Count Paths in matrix - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/count-possible-paths-top-left-bottom-right-nxm-matrix/
45. Unique Paths II - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/unique-paths-ii/
46. Unique Paths - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/unique-paths/
47. Kadane's Algorithm : Maximum Subarray Sum in an Array - Tutorial, accessed on September 12, 2025, https://takeuforward.org/data-structure/kadanes-algorithm-maximum-subarray-sum-in-an-array/
48. How to find the maximum sum subarray of a given array using DP - Quora, accessed on September 12, 2025, https://www.quora.com/How-do-I-find-the-maximum-sum-subarray-of-a-given-array-using-DP
49. Matrix Chain Multiplication Algorithm - Tutorials Point, accessed on September 12, 2025, https://www.tutorialspoint.com/data_structures_algorithms/matrix_chain_multiplication.htm
50. Matrix Chain Multiplication - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/matrix-chain-multiplication-dp-8/
51. Matrix Chain Multiplication | (DP-48) - Tutorial - takeUforward, accessed on September 12, 2025, https://takeuforward.org/dynamic-programming/matrix-chain-multiplication-dp-48/