Backtracking: Introduction

Let’s understand the Backtracking pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following

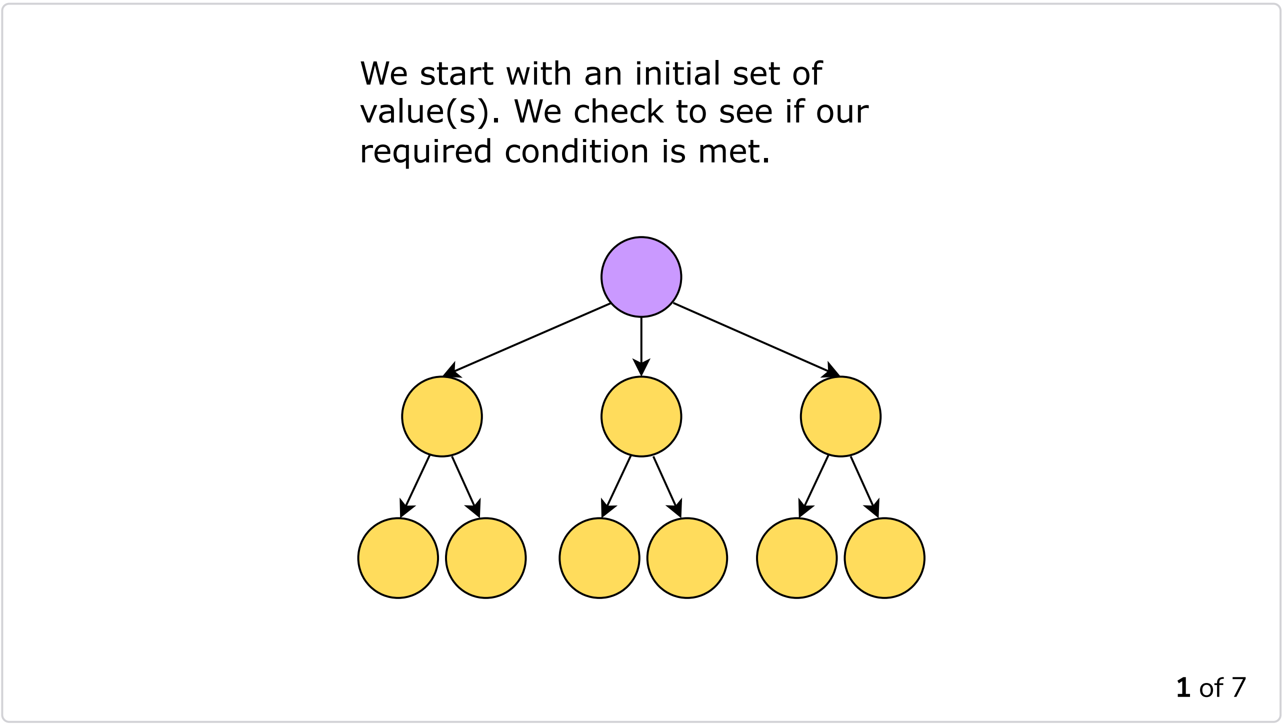
Overview

Backtracking is a technique that explores multiple paths to find the solution. It builds the solution step by step by increasing values with time and removes the choices that don’t contribute to the problem’s solution based on some constraints. Backtracking is different from recursion because, in recursion, the function calls itself until it reaches a base case whereas backtracking tries to explore all possible paths to a solution.

The way backtracking works is that it first explores one possible option. If the required criteria have been met with that option, we choose a path that stems from that option and keep on exploring that path. If a solution is reached from this path, we return this solution. Otherwise, if a condition is violated from the required set of conditions, we backtrack and explore another path.

The backtracking approach is better than brute-force since we don’t hav to generate all possible solutions and choose our required solution from among these. It provides us with the option to check our required condition at each possible recursive call. If the condition is met, we continue exploring that path. If it isn't, we take a step back and explore another path. In this way, we avoid generating redundant solutions.

The following illustration shows how backtracking explores different paths until the base condition is met:


We start with an initial set of value(s). We check to see if our required condition is met.
1 of 7

Examples

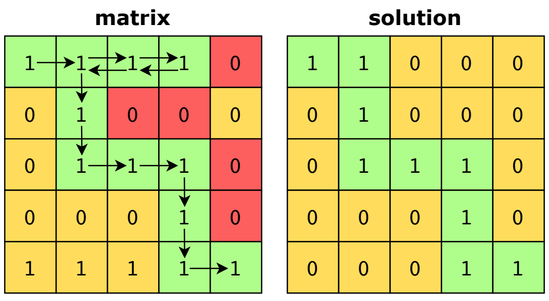
The following examples illustrate some problems that can be solved with this approach:

Find a path of 1s from top-left to bottom-right in an nxn binary maze

Note: We are only allowed to move to the right or downward.

We declare a **solution** matrix initialized to **0** to store the path we find.

1. We start from the top-left cell of the matrix and check if the position is valid, i.e., whether the cell contains **1**.
2. If the position is valid, we recursively keep moving to the right and mark the corresponding cell in the solution matrix to **1**.
3. If we encounter a **0**, we can't move to the right anymore, so we try to move down from there.
4. If we can't go in either direction, we backtrack to a **1** from which we can still move downward and again try to recursively find a valid path from there. When backtracking, we need to unmark the corresponding position in the **solution** matrix to **0**, since that cell will not be part of the path leading to the destination.



matrix

1	1	1	1	0
0	1	0	0	0
0	1	1	1	0
0	0	0	1	0
1	1	1	1	1

solution

1	1	0	0	0
0	1	0	0	0
0	1	1	1	0
0	0	0	1	0
0	0	0	1	1

1 of 2

Does my problem match this pattern?

- Yes, if any of these conditions is fulfilled:
    - While constructing any single candidate solution, all paths must be explored. This means that if exploring a certain path results in a dead end, we need to move back one level and explore all the other paths in the solution space.
      Example: Determine whether an undirected graph can be colored using no more than n colors in such a way that no two adjacent vertices share the same color.
    - The problem requires us to consider all feasible solutions in order to select the best one. While solving such a problem, not a single feasible solution may be ignored. In certain problems, even if some feasible solutions are eventually discarded, we stil need to find and evaluate them.

Example: Select elements from an array of strings and concatenate them, such that the concatenated string is of the maximum possible length and contains unique characters.

    - The problem requires us to compile a list of all feasible solutions.
      Example: Partition a given string into substrings so that each substring is a palindrome. We need to determine all possible ways in which the partitioning can be done to obtain palindromic substrings.

- No, if the problem structure is such that, while constructing a candidate solution, failing to meet the acceptability criteria of the solution eliminates all other possibilities within that solution. So, we don't need to check those remaining possibilities.
  Example: Given a set of integers and a target sum, determine if any subset of the given set of integers can sum up to the target sum. The nature of the problem is such that, once we find a subset that sums up to the target, there is no need to continue exploring other subsets. Similarly, if we reach a point where adding any additional elements to the current subset would result in the sum exceeding the target, there is no need to explore any more possibilities for that subset.

Real-world problems

Many problems in the real world share the backtracking pattern. Let's look at some examples.

Constraint satisfaction problems: Backtracking is used to solve puzzles

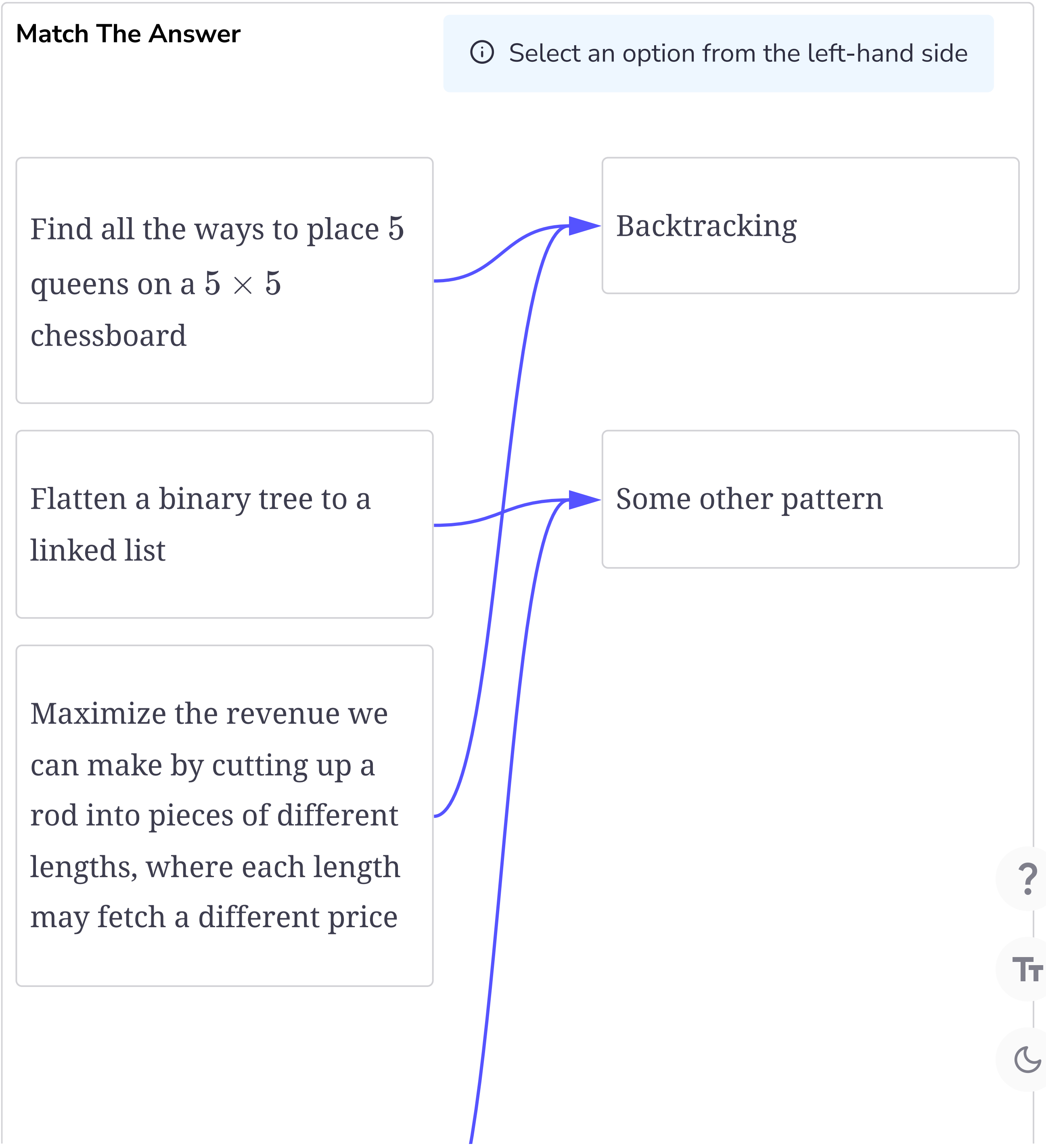
violates the rules, and try again to find a valid path that satisfies the conditions.

Recursive descent parsers: The compiler takes a path through the grammar and reaches a point where the incoming tokens no longer mat that part of the grammar. Therefore, the compiler backtracks to a point where there is another path through the grammar and follows that. This can occur several times until the compiler finds a grammar path that fits the incoming tokens, in which case it accepts the construct. If it finds no matching path, a syntax error is shown.

Strategy time!

Match the problems that can be solved using the backtracking pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.



Match The Answer

Select an option from the left-hand side

Find all the ways to place 5 queens on a 5×5 chessboard	Backtracking
Flatten a binary tree to a linked list	Some other pattern
Maximize the revenue we can make by cutting up a rod into pieces of different lengths, where each length may fetch a different price	

Calculate the median from a data stream

Reset Show Solution Submit

Back

Next