**Introduction to Backtracking**

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

**About the pattern**

Imagine we’re planning an exciting road trip through a city, aiming to visit all the places we want to see while covering the shortest distance possible. However, there are some conditions we must follow: we can’t revisit the same place more than once, and we must end up back where we started. This problem, known as the city road trip problem, requires finding the optimal route that satisfies these conditions. It’s a classic example where the concept of backtracking comes into play, allowing us to explore different paths until we find the shortest one that fulfils all the conditions.

Let’s first see how this problem can be solved using a brute-force approach. We can do this by exploring routes in every single way we can visit the places. We have to write down every possible route, check how long each one is, and then pick the shortest one. But as our list of places grows, it makes this approach computationally impractical for a large number of routes.

Now, let’s look at a backtracking approach to solve the same problem. With backtracking, we can start by picking a place and choose the next place to visit that’s close and follows our conditions. We move back (backtrack) to the previous place if the current place has been visited before or if we cannot move forward to any place from here. We check these conditions on each of our choices because we do not want to break any of our road trip rules. We keep doing this, choosing, checking conditions, and backtracking until we’ve visited all the places according to the requirements. At every step, we choose the closest place, ensuring we have chosen the shortest path to visit all the places we want to see.

**Backtracking** is an algorithmic technique for solving problems by incrementally constructing choices to the solutions. We abandon choices as soon as it is determined that the choice cannot lead to a feasible solution. On the other side, brute-force approaches attempt to evaluate all possible solutions to select the required one. Backtracking avoids the computational cost of generating and testing all possible solutions. This makes backtracking a more efficient approach. Backtracking also offers efficiency improvements over brute-force methods by applying constraints at each step to prune non-viable paths.

As seen in the above example, backtracking works by exploring all potential routes toward a solution step-by-step. It can be visualized as traversing a state space tree, where each node represents a partial solution. Starting from the root (an empty solution), backtracking moves deeper into the tree, exploring branches (choices) until it finds a feasible solution or reaches a leaf node that cannot be extended into a complete solution. Upon reaching a dead end, the algorithm backtracks to the previous state and explores a different branch. This process is repeated, with constraints applied at each step to avoid exploring paths that cannot lead to a successful, feasible solution.

Let’s look at the visualization of the space state tree below to better understand the working:

A diagram of a business process

AI-generated content may be incorrect.In the visualization above, we start with an initial point, SS. From this point, we proceed to explore a potential solution, S1S1, via an intermediate choice, C1C1. After evaluating, we determine that S1S1 does not satisfactorily solve our problem. Therefore, we backtrack to SS and then shift our exploration towards another potential choice, C2C2. This process of exploration and backtracking continues until we identify a successful feasible solution.

In the scenario above, both S1S1 and C2C2 fail to provide feasible solutions and only S3S3 emerges as a successful solution to the problem. This illustrates that the backtracking approach examines all potential combinations until it discovers a successful feasible solution.

The backtracking algorithm can be implemented using recursion. We use recursive calls where each call attempts to move closer towards a feasible solution. This can be outlined as follows after starting from the initial point as the current point:

* **Step 1:** If the current point represents a feasible solution, declare success and terminate the search.
* **Step 2:** If all paths from the current point have been explored (i.e., the current point is a dead-end) without finding a feasible solution, backtrack to the previous point.
* **Step 3:** If the current point is not a dead-end, keep progressing towards the solution, and reiterate all the steps until a solution is found or all possibilities are exhausted.

If all the points have been explore without finding a feasible solution, declare failure; no solution exists.

Now, lets look at the following illustration to visualize the algorithm discussed above:

A screenshot of a diagram

AI-generated content may be incorrect.A screenshot of a diagram

AI-generated content may be incorrect.A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a diagram

AI-generated content may be incorrect.A screenshot of a diagram

AI-generated content may be incorrect.

**Examples**

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

**1. Path in binary matrix:** Find a path of 1s from top-left to bottom-right in an n×n binary maze. We are only allowed to move to the right or downward.

A screenshot of a game

AI-generated content may be incorrect.

## **Check knight tour configuration:** Check if a knight can cover all possible squares once in an n×n chessboard. The initial position of the knight is at the top-left square of the board.A screenshot of a puzzle AI-generated content may be incorrect. **Does your problem match this pattern?**

* Yes, if any of these conditions is fulfilled:
  + **Complete exploration is needed for any feasible solution:** The problem requires considering every possible choice to find any feasible solution.
  + **Selecting the best feasible solution:** When the goal is not just to find any feasible solution but to find the best one among all feasible solutions.
* No, if the following condition is fulfilled:
  + **Solution invalidity disqualifies other choices:** In problems where failing to meet specific conditions instantly rules out all other options, backtracking might not add value.

**Real-world problems**

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

* **Syntax analysis:** In compilers, we use recursive descent parsing. It is a form of backtracking, to analyze the syntax of the program. This analysis involves matching the sequence of tokens (basic symbols of programming language) against the grammar rules of the language. When a mismatch occurs during the analysis, the parser backtracks to a previous point to try a different rule of the grammar. This ensures that even complex nested structures can be accurately understood and compiled.
* **Game AI (Artificial Intelligence):** In games like chess or Go, AI algorithms use backtracking to try out different moves and see what happens. If a move doesn’t work out well, the AI goes back and tries something else. This helps the AI learn strategies that might be better than those used by humans because it can think about lots of different moves and figure out which ones are likely to work best.
* **Pathfinding algorithms:** In pathfinding problems like finding the way through a maze or routing in a network, backtracking is used. It tries out different paths to reach the destination. If it hits a dead end or a spot it can’t pass through, it goes back and tries another path. This keeps happening until it finds a path that works and leads to the destination

**Subarray, subset & subsequence**

Consider an array:

{1,2,3,4}

* Subarray: contiguous sequence in an array i.e. {1,2},{1,2,3}
* Subsequence: Need not to be contiguous, but maintains order i.e. {1,2,4}
* Subset: Same as subsequence except it has empty set and do not have to maintain any order i.e. {3,1},{},{4,3}
* Permutations: A **permutation** is essentially an ordered combination, except the total length of each permutation must equal the original input. Finding all permutations of a string is sort of the same as saying "find all anagrams of a string" (except our permutations might not all be real words). Eg: {1,2,3} has permutations {1,2,3},{1,3,2},{2,1,3},{2,3,1},{3,2,1},{3,1,2}
* Combination: A Combination is selecting items without considering order, while Permutation is arranging items considering order of selection from a certain group. In Comb. AB is same as BA, but in Perm. they are diff.

Given an array/sequence of size n, possible

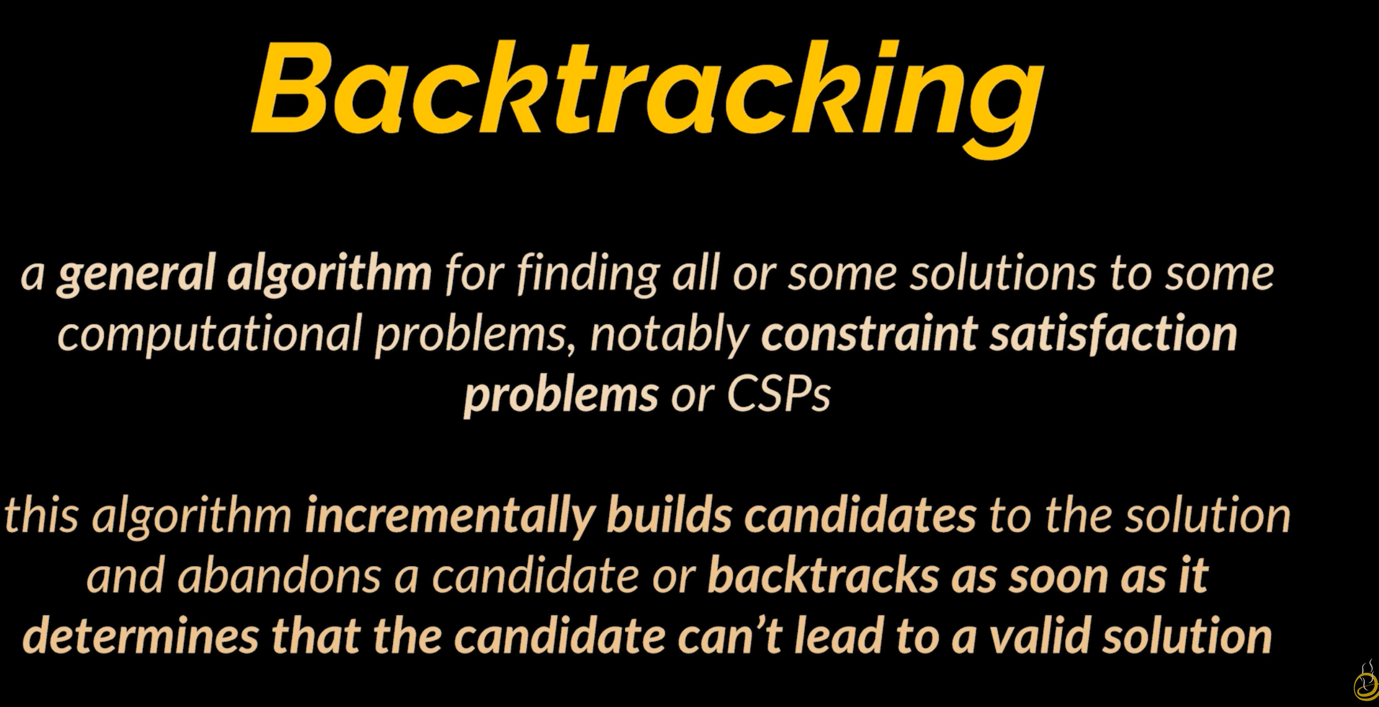
Subarray = n\*(n+1)/2

Subseqeunce = (2^n) -1 (non-empty subsequences)

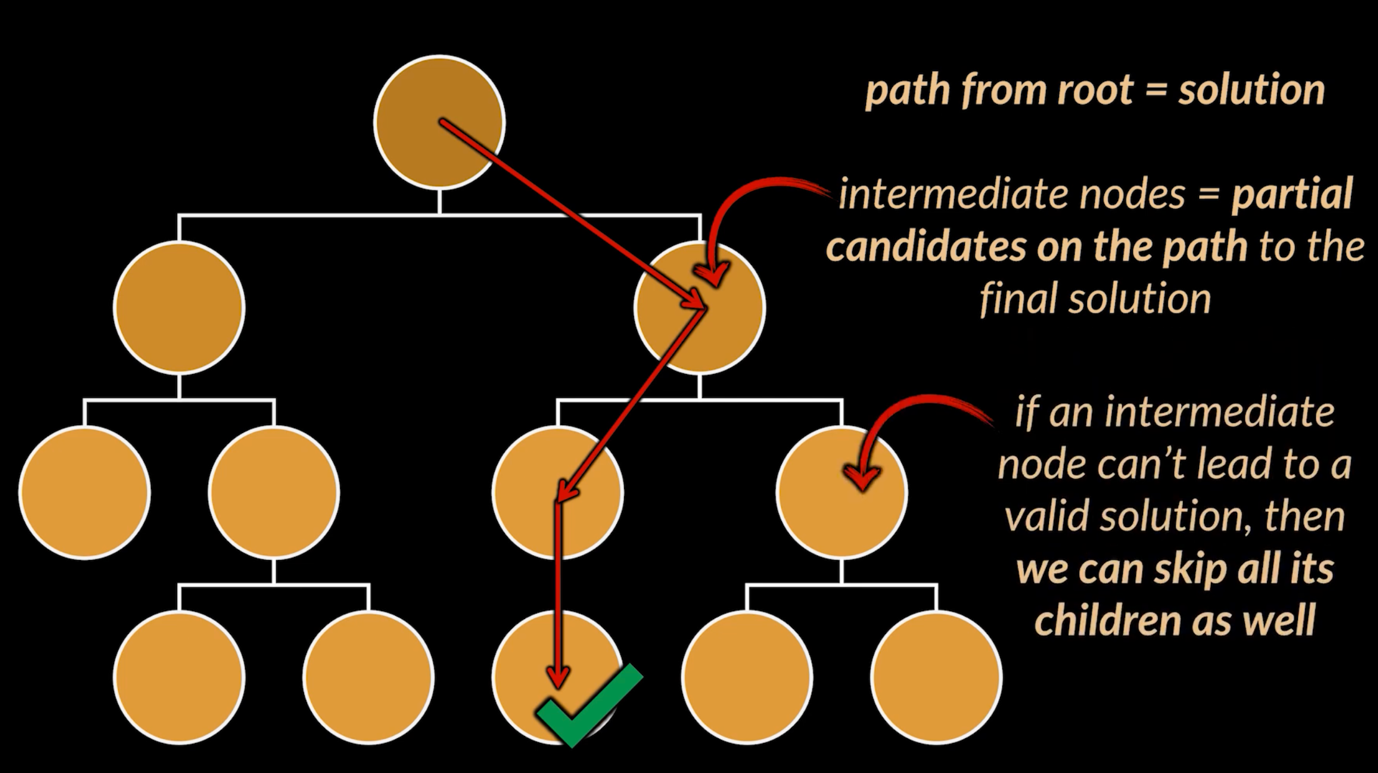
Subset = 2^n

n! for permutations

<https://www.youtube.com/watch?v=vqnZ9RhhkmY>



To think easier, we can think of backtracking as a tree traversal. We start from root and our solution ends at one of the leaf nodes.

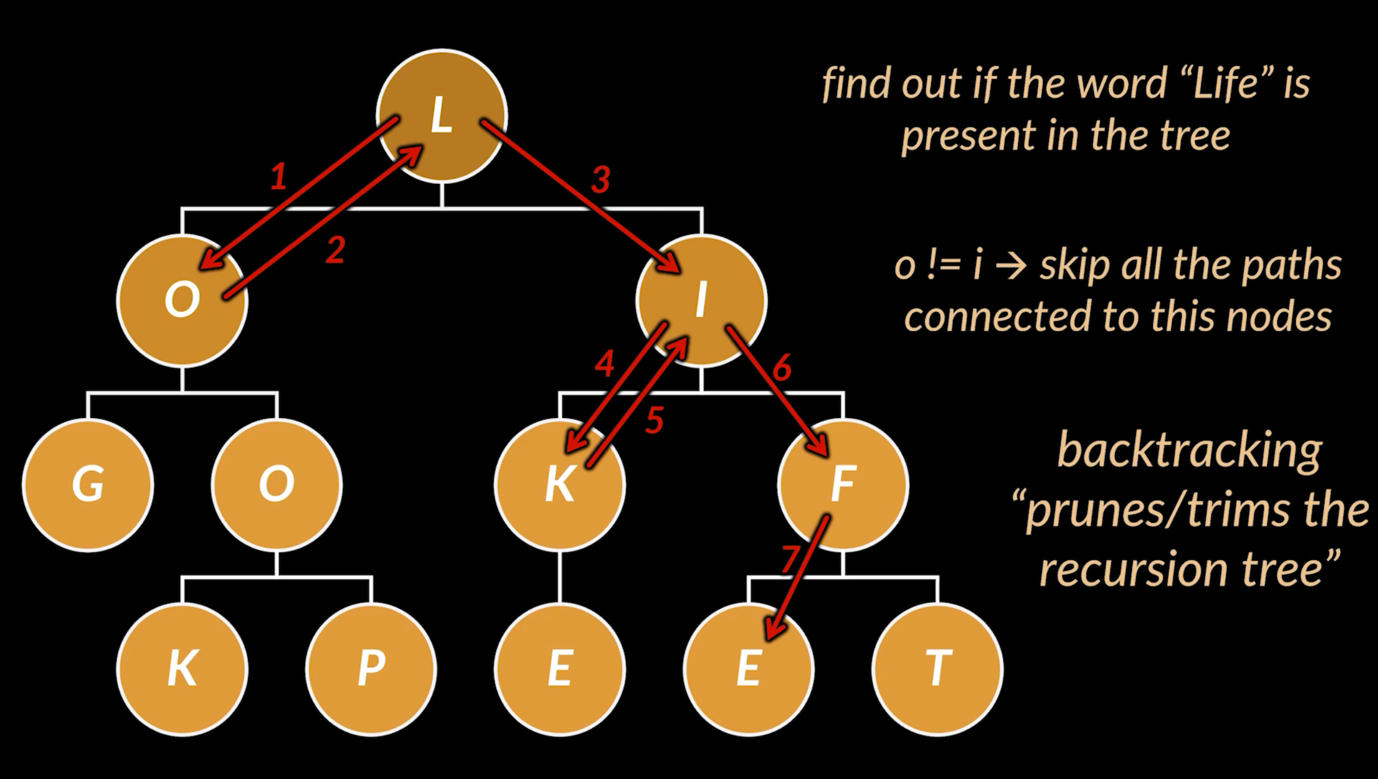


For ex, we want to find a word in a tree where each node is a character. Brute force way here would be to traverse each path and compare the word obtained from inp word (Life) in this case.

A screenshot of a computer

AI-generated content may be incorrect.

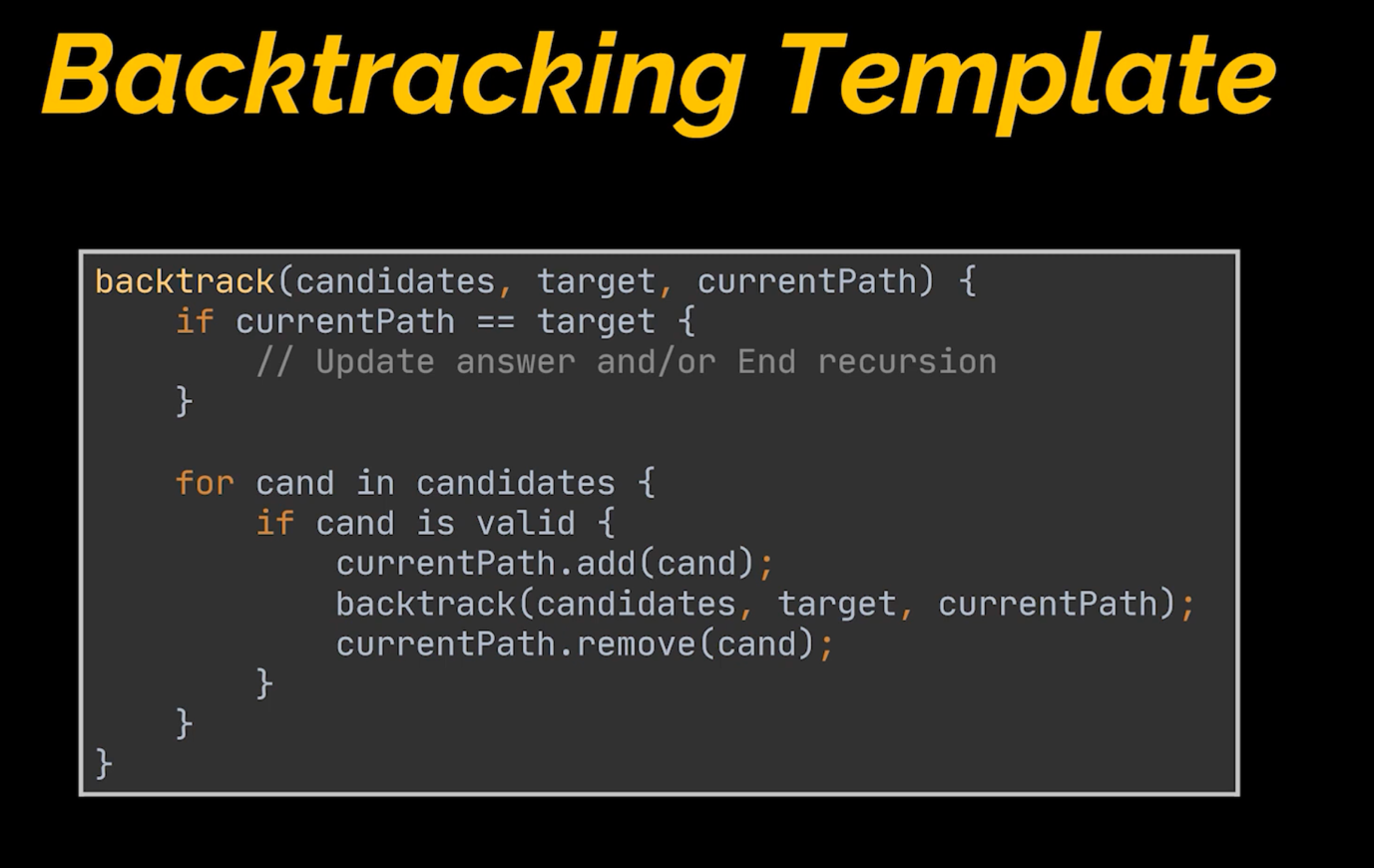
But in backtrack, we move down each path but stops as soon as we know that in this path sol is not possible.



Backtrack template:

In a backtrack problem, we have these components needed for sure:

* Input Candidates
* Target
* currentPath: Since BTrack is a recursion, we need some var which hold the state to keep track of where we are in a recursion or traversal. As soon as currentPath matches the target we can update our currentpath or end the recursion.
* After the breaking condition, we need to traverse through our input candidates, if the curr candidate is valid we can add it to currPath.
* Then we call the backtrack algo again
* And post that we clear the currPath of the last recursion path it took so that it can take the next path



## 🧠 **Rule of Thumb**

**Use recursion with a loop** when you have **multiple choices at each step** and need to **try all of them**.

**Use plain recursion** when you only have **one choice (or deterministic path)** at each step.

### 🔁 Recursion **with a loop** = Backtracking over a "decision space"

* You are **exploring all possibilities**, usually from a **list** or **range**.
* The loop iterates through choices, and recursion explores further from each.

### ➡️ Plain Recursion = One path, no branching

## 1. **Recursion** – "Try all paths"

A method that calls itself to solve smaller subproblems.

* **Core Idea**: Break a problem into smaller versions of itself.
* **No optimization or memory** — just brute-force exploration.
* Often leads to exponential time if overlapping subproblems are present.

### ✅ Example (Fibonacci)

📌 Recursion is the base of both backtracking and DP.

## 🔄 2. **Backtracking** – "Try all **choices** at each step, undo when done"

A special kind of recursion used when you need to **build and explore multiple paths** (e.g., combinations, permutations).

* **Key Feature**: You build a path, and then undo the choice (backtrack) to explore another path.
* Used in **combinatorics**, **constraint problems**, **puzzle solving**.
* Involves:
  + A loop to explore multiple options.
  + A way to track current state (like a path or visited list).
  + An "undo" step (path.pop() or remove()).

### ✅ Example (Generate all subsets)

📌 **Recursive + exploratory + undo** = backtracking.

## 🧠 3. **Dynamic Programming (DP)** – "Solve once, remember the result"

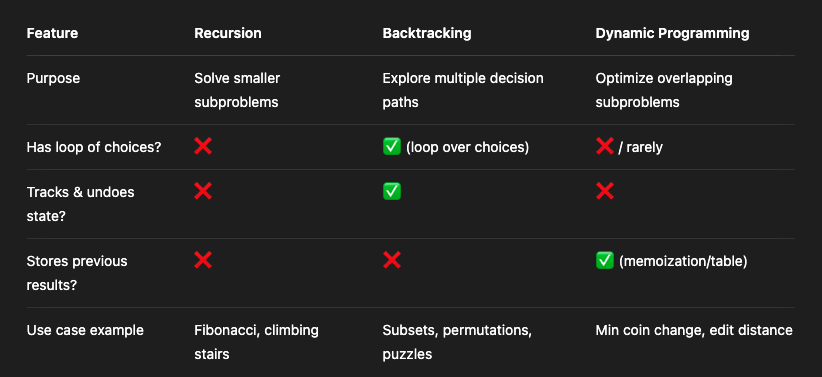
An optimization of recursion: **store results** of subproblems to avoid re-computation.

* Use when:
  + Problem has **overlapping subproblems** (same inputs repeat)
  + Problem has **optimal substructure** (you can build solution from sub-solutions)
* Two types:
  + **Top-Down (Memoization)**: Recursive + cache
  + **Bottom-Up (Tabulation)**: Iterative + table

### ✅ Example (Fibonacci with DP)

📌 DP avoids redundant work — unlike plain recursion.

🔍 Summary Table



### 🔹 Step 3: Are there **overlapping subproblems**? (i.e., same inputs recomputed multiple times)

* **Yes → Use Dynamic Programming (Memoization or Tabulation)**
  + ✅ Common in: Coin change, Knapsack, Fibonacci, Edit distance, Grid paths.
  + Use:
    - Memoization (Top-down) → recursion + cache
    - Tabulation (Bottom-up) → loop-based with DP array
* **No → Plain Recursion is fine**
  + ✅ Example: Binary search, tree traversal, tower of Hanoi

| **Problem Pattern** | **Use Technique** |
| --- | --- |
| Generate all possibilities | Backtracking |
| Make a choice, explore, undo | Backtracking |
| Solve once, reuse results | DP |
| Build solution from subparts | Recursion or DP |
| No overlapping calls | Recursion only |

| **Problem** | **Type** |
| --- | --- |
| Subsets of a list | Backtracking |
| Coin change (ways) | DP |
| Coin change (combinations) | Backtracking |
| Fibonacci (naive) | Recursion |
| Fibonacci (optimized) | DP |
| Sudoku solver | Backtracking |
| Maze path finder | Recursion / Backtracking |
| Climbing stairs | DP (with memo) |

Min coin change:

private static void minCoinsRequired\_whenAnyNumOfCoinofEachCanBeUsed(int target, List<Integer> coins, int index, List<Integer> coinsConsideredSofar, List<List<Integer>> opMain) {  
 if (target < 0 || index < 0)  
 return;  
 if (target == 0) {  
 opMain.add(coinsConsideredSofar);  
 return;  
 }  
 coinsConsideredSofar.add(coins.get(index));  
 *minCoinsRequired\_whenAnyNumOfCoinofEachCanBeUsed*(target - coins.get(index), coins, index, coinsConsideredSofar, opMain);  
 coinsConsideredSofar.remove(coinsConsideredSofar.size() - 1);  
 *minCoinsRequired\_whenAnyNumOfCoinofEachCanBeUsed*(target, coins, index - 1, coinsConsideredSofar, opMain);  
}

It finds all combination but doesnt use for loop. so is it a backtrack or recursion?

this code **does implement backtracking**, even though it **doesn’t use a for loop**.

Let’s break it down clearly:

## 🔍 What is Backtracking?

Backtracking is **recursive search + decision reversal**, regardless of whether a for loop is used.

The **core traits of backtracking** are:

1. **Making a choice** (here: include or exclude a coin),
2. **Recursing** to explore deeper,
3. **Undoing the choice** (here: remove(...) from list),
4. Exploring alternative paths (not just one).

* You're **trying one decision** (include current coin).
* Then **undoing it** (backtrack via remove()).
* Then **trying the next choice** (move to next coin by reducing index).

This is logically equivalent to:

for (int i = start; i < coins.length; i++) {

// try coin[i]

// recurse

// backtrack

}

🧠 Rule of Thumb

| **Feature** | **Backtracking** | **Plain Recursion** |
| --- | --- | --- |
| Multiple branches | ✅ | ✅ |
| State building (path) | ✅ | ❌ |
| Undoing state (remove) | ✅ | ❌ |
| All possible valid solutions | ✅ | ❌ (usually 1 path) |
| Uses loop over choices | Optional | Not needed |

Let’s **refactor your backtracking coin change function** to a cleaner and more idiomatic version using a for loop — this also makes it easier to generalize and more readable in interviews.

🔁 **Original (No Loop, Two Branches)**

private static void minCoinsRequired\_whenAnyNumOfCoinofEachCanBeUsed(

int target, List<Integer> coins, int index,

List<Integer> coinsConsideredSofar, List<List<Integer>> opMain) {

if (target < 0 || index < 0)

return;

if (target == 0) {

opMain.add(new ArrayList<>(coinsConsideredSofar));

return;

}

// Include coin at index

coinsConsideredSofar.add(coins.get(index));

minCoinsRequired\_whenAnyNumOfCoinofEachCanBeUsed(

target - coins.get(index), coins, index, coinsConsideredSofar, opMain);

// Backtrack

coinsConsideredSofar.remove(coinsConsideredSofar.size() - 1);

// Exclude coin and move to next

minCoinsRequired\_whenAnyNumOfCoinofEachCanBeUsed(

target, coins, index - 1, coinsConsideredSofar, opMain);

}

✅ **Refactored with For-Loop**

private static void coinChangeBacktrack(

int target, List<Integer> coins, int start,

List<Integer> path, List<List<Integer>> result) {

if (target == 0) {

result.add(new ArrayList<>(path));

return;

}

for (int i = start; i < coins.size(); i++) {

int coin = coins.get(i);

if (coin <= target) {

path.add(coin);

// Recurse: can reuse same coin, so i (not i + 1)

coinChangeBacktrack(target - coin, coins, i, path, result);

path.remove(path.size() - 1); // Backtrack

}

}

}