GUVI | HCL
Skill Up. Level Up

Recursion and Backtracking

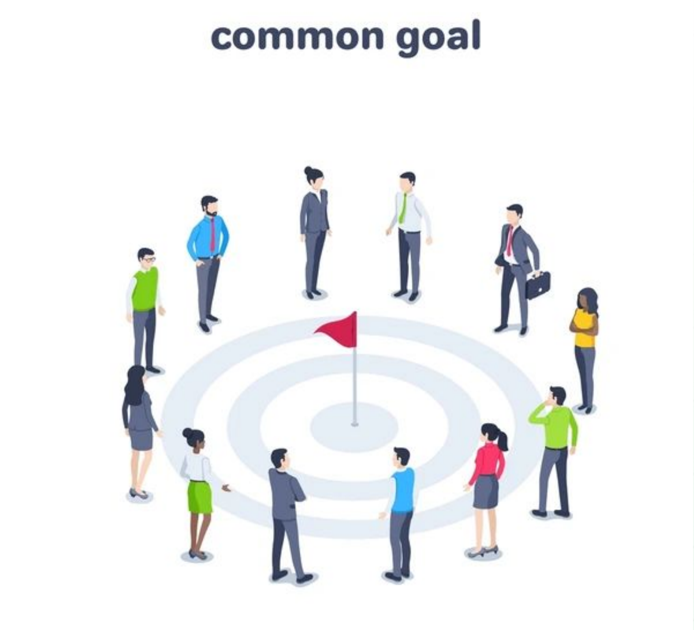
DSA

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

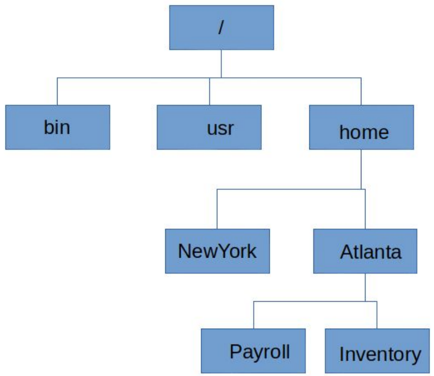
- What you will learn today
  - Introduction to Recursion
  - Recursive Tree Patterns
  - Backtracking Problems (e.g., N-Queens, Sudoku Solver)
  - Memoization Basics



Introduction to Recursion

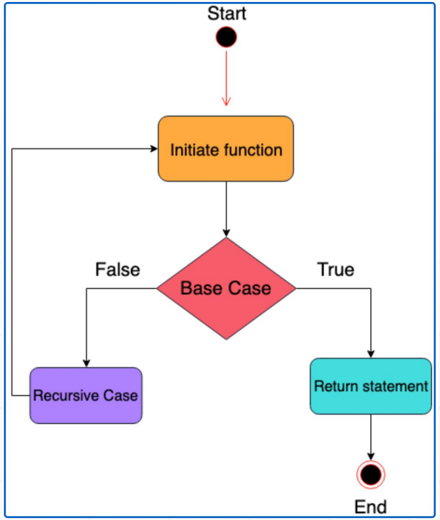
A New Problem-Solving Model

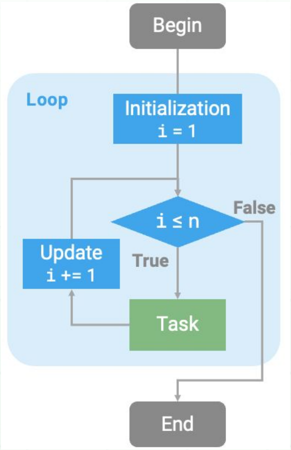
- An alternative to iteration for problems that have self-similarity
- The core strategy: Decompose a complex problem into a simpler instance of the same problem
- This repeats until the problem becomes trivial to solve directly
- Analogy: A file system. The size of a directory is the sum of the sizes of the items inside it



Anatomy of a Recursive Function

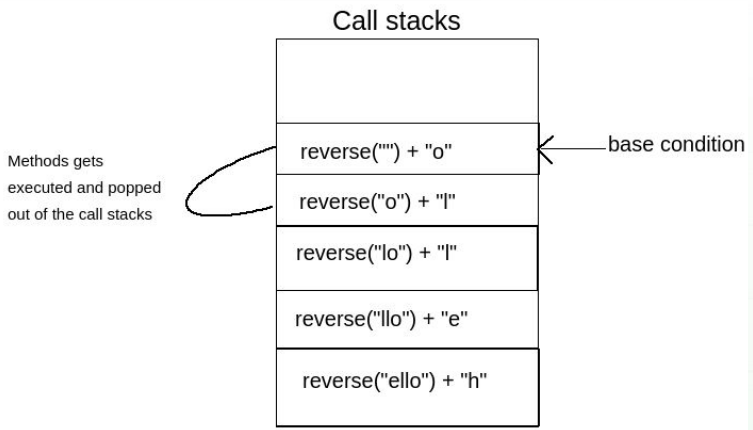
- Rule 1: The Base Case (Termination Condition)
  - A simple, non-recursive branch that returns a known, final value
  - This is the only thing that stops the calls from repeating forever
- Rule 2: The Recursive Step (Reduction Step)
  - The branch that calls the same method but with a modified argument
  - This new argument must progressively converge toward the base case





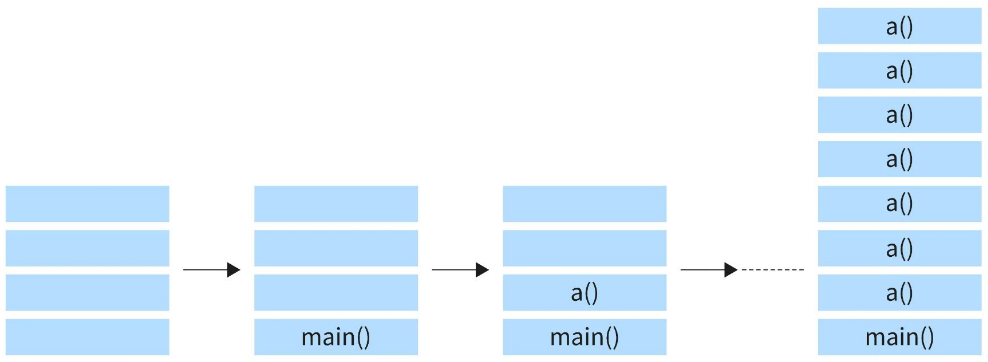
How Recursion Uses the Call Stack

- Every recursive call pushes a new stack frame onto the Call Stack
- Each frame isolates its own state (its unique local variables and parameters)
- The stack grows with each recursive step, consuming memory
- The stack unwinds as each frame returns its result to the caller below it



The Risk: Uncontrolled Recursion

- What if the base case is wrong, or the step fails to converge?
- Each recursive call unconditionally pushes a new stack frame
- The Java Call Stack is a finite memory region
- Exceeding this limit causes a java.lang.StackOverflowError
- This is a fatal Error, not an Exception: it signals an unrecoverable program state



Code Example: Calculating a Factorial

- **Purpose**: Implement n! using the two recursive rules.

  Input: (e.g., 5! = 5 * 4 * 3 * 2 * 1)
- The if (n <= 1) is the Base Case; it prevents infinite calls.
- The return n * factorial(n - 1) is the Recursive Step.
- Trace these:
    - The standard case: factorial(4)
    - The boundary case: factorial(0)
    - The error case: (running with no base case)

```java
public static int factorial(int n) {
  // 1. The Base Case (Termination Condition)
  if (n <= 1) {
    return 1;
  }

  // 2. The Recursive Step (Reduction Step)
  return n * factorial(n - 1);
}
```

Tracing the Stack: factorial(3)

- Phase 1: The "Push" (Stacking Calls)
  - main calls factorial(3). **Stack**: [main, f(3)]. f(3) waits.
  - f(3) calls factorial(2). **Stack**: [main, f(3), f(2)]. f(2) waits.
  - f(2) calls factorial(1). **Stack**: [main, f(3), f(2), f(1)].
  - f(1) hits the Base Case.
- Phase 2: The "Pop" (Unwinding & Calculating)
  - f(1) returns 1. f(1) is popped.
  - f(2) gets 1, returns 2 * 1 = 2. f(2) is popped.
  - f(3) gets 2, returns 3 * 2 = 6. f(3) is popped.
- The final value is assembled as the stack unwinds
- Consider: what's the time and space complexity?

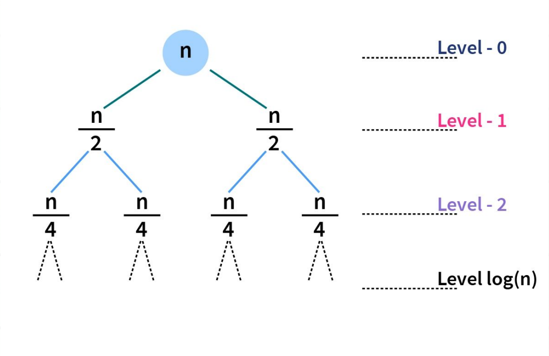
Activity: Trace the Stack

- Trace the execution of factorial(4)
- Use the two-phase "Push" and "Pop" model we just discussed
- Questions to Answer
    - What is the deepest the stack gets? (i.e., how many factorial frames?)
    - Which function call returns 1?
    - What value is returned by the factorial(3) frame to its caller?
- Share your solution in [Lecture 19 Discussion]!

Recursive Tree Patterns

What is a Recursive Tree?

- *factorial* was linear recursion, meaning one recursive call creates a single path down the stack
- What happens when a recursive step makes multiple calls to itself?
    - The calls branch out, forming a structure that looks like a tree
- This "tree recursion" is essential for problems that explore multiple distinct possibilities
- Consider: where in the code does this branching happen?



Code: Calculating Fibonacci

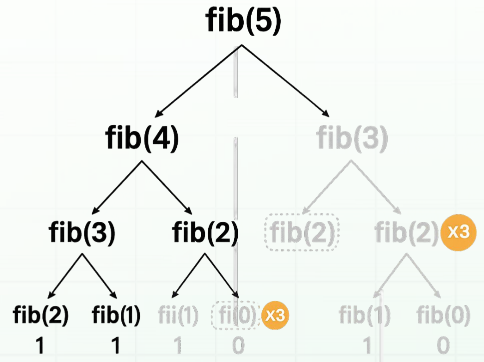
- **Purpose**: Implement the Fibonacci sequence (0, 1, 1, 2, 3, 5...) where fib(n) = fib(n-1) + fib(n-2).
- Note the two base cases, n=0 and n=1, handled by one if.
- Note the two recursive calls in the return statement.
- Let's attempt to run fibonacci(45) to observe its real-world performance

```java
public static long fibonacci(int n) {
    // Base Cases (handles 0 and 1)
    if (n <= 1) {
        return n;
    }

    // Recursive Step (two calls, creating a tree)
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

The Problem: Redundant Calculations

- Observe the call tree for fibonacci(5):
- Observations from this tree:
  - We are re-calculating the exact same values multiple times.
  - fib(3) is calculated 2 times.
  - fib(2) is calculated 3 times.
- This problem is called "Overlapping Subproblems".
- This is why the performance is O(2^n) (Exponential Time).



Call tree for fibonacci(5) – duplicates. fib(3) appears 2x, fib(2) appears 3x

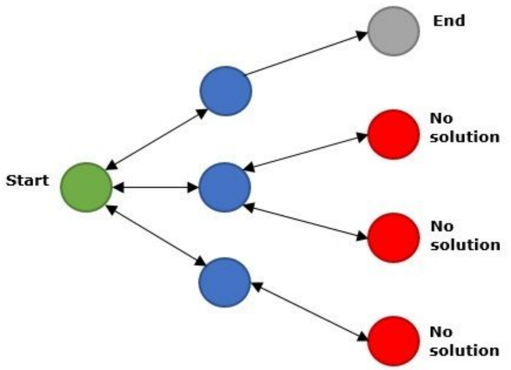
Activity: Draw the Call Tree

- Draw the full recursive call tree for fibonacci(4)
- Label every single call (e.g., f(4) calls f(3) and f(2))
- Questions to Answer
    - How many total recursive calls are made (e.g., f(3), f(2), etc.)?
    - How many times is fibonacci(2) calculated?
    - How many times is fibonacci(1) calculated?
- Share your solution in Lecture 19 Discussion!

Backtracking

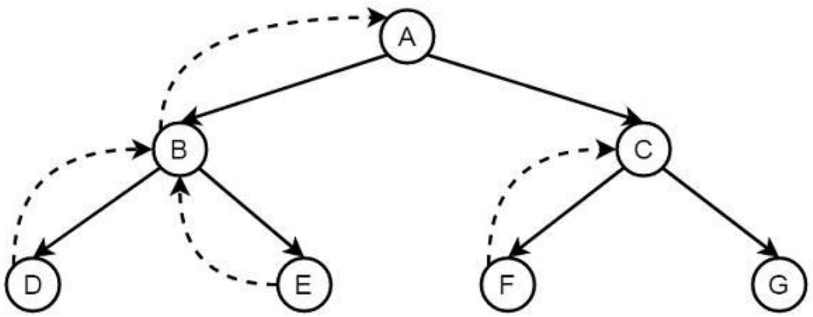
What is Backtracking?

- Backtracking is an algorithmic strategy that uses tree recursion
- It's a methodical way to find solutions by exploring all possible candidates
- It incrementally builds a solution, and abandons a path as soon as it fails
- **Analogy**: You are in a maze. You take a path. If you hit a dead end, you "backtrack" to the last junction and try a different path



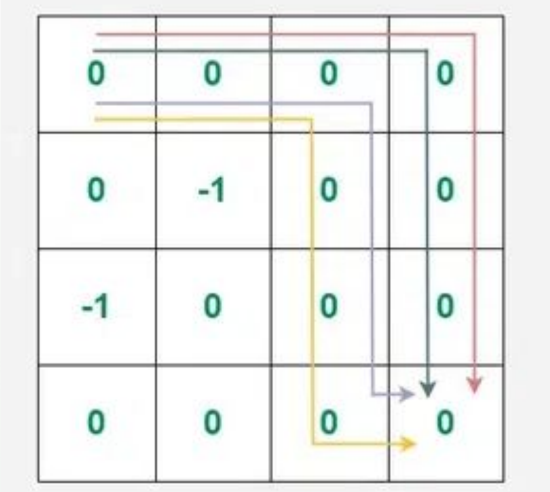
The Backtracking Algorithm Pattern

- All backtracking solutions follow this three-step pattern
    - **Choose**: Make a choice (e.g: move 'Right' in the maze).
    - **Explore**: Make a recursive call to see if this choice leads to a valid solution
        - If the recursive call returns true, a solution was found. Pass true up.
    - **Un-choose**: If the call returns false (dead end), undo your choice (backtrack)
        - This "un-choose" step is the most critical part in eventually finding the solution



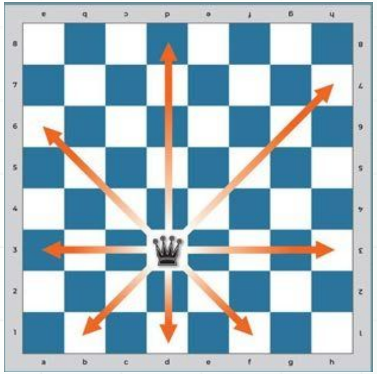
Visualizing Backtracking: A Maze

- Let's apply the "Choose, Explore, Un-choose" pattern to go from top left to bottom right of the maze
- Path: [0,0] (Our starting point)
  - Choose: Try the first option, Move Down to [1,0].
  - Explore: Call a method solve([1,0]).
- Path: [1,0] (The recursive call)
  - This call now tries its options (e.g., Move Right, Move Down).
  - Every path from [1,0] leads to a wall or a dead end
  - The solve([1,0]) call eventually tries all its options and fails
  - It returns false to its caller (solve([0,0])).
- Path: [0,0] (We are back in the first call)
  - The solve([1,0]) call returned false. That path was a failure.
  - Un-choose: We abandon the Down path.
  - Choose: We try the next option, Move Right.
  - Explore: Call solve([0, 1]). The process repeats



Visualizing Backtracking: The N-Queens Problem

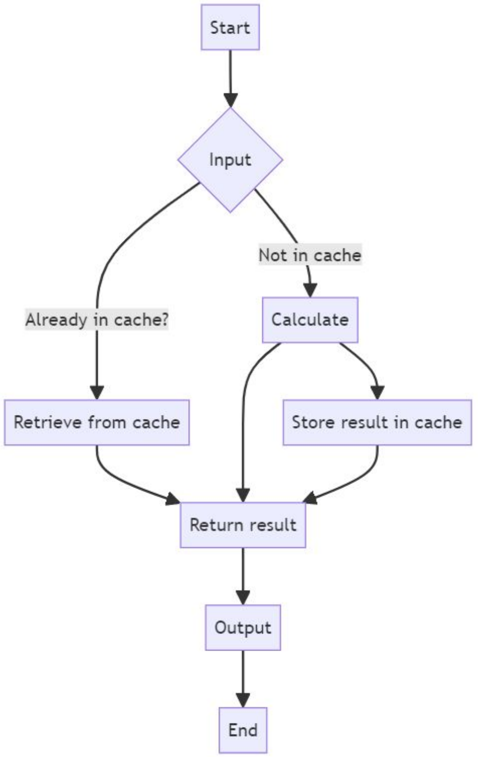
- **The Problem**: Place N queens on an N×N board so no two queens attack each other
- A queen attacks horizontally, vertically, and diagonally
- The Backtracking Approach
  - **Choose**: Place a queen in the first available column of the current row.
  - **Explore**: Check if this placement is safe. If yes, recursively call solve(row + 1).
  - **Un-choose**: If solve(row + 1) returns false (no safe spot in the next row), remove the queen from the current row and try the next column



Memoization

What is Memoization?

- Memoization is a specific optimization technique for recursion
- It's not the same as "memorization." It comes from "memo," as in writing a note
- The Pattern
  - Create a "cache" (an array or Map) to store results.
  - At the start of the function, check the cache. If the answer is there, return it immediately.
  - If the answer is not in the cache, compute it recursively.
  - Before you return the new answer, save it in the cache



Fibonacci with Memoization

- **Purpose**: Refactor our fibonacci function to use a memoization "cache" (an array)
- Time Complexity: O(n). We only compute each fib(i) once.
- Space Complexity: O(n) for the cache array + O(n) for the stack.

```java
// We use an array as our "cache"
static long[] fibCache; // Will be initialized with -1

public static long fibonacciMemoized(int n) {
    // Base Cases
    if (n <= 1) return n;

    // 1. Check the cache. -1 means "not yet computed"
    if (fibCache[n] != -1) {
        return fibCache[n]; // Return the stored value!
    }

    // 2. Not in cache. Compute it...
    long result = fibonacciMemoized(n - 1) + fibonacciMemoized(n - 2);

    // 3. ...store it, then return it.
    fibCache[n] = result;
    return result;
}
```

Try it yourself: Refactor with Memoization

- The given method calculates the number of unique paths in a grid
- It has the exact same overlapping subproblems as Fibonacci
- Refactor this code to use memoization
- Hint: you will need a 2D array (int[][] memo) as your cache
- Share your solution on [Github Discussions]!

```java
/*
 * Problem: Count all paths from (0,0) to (m,n).
 * You can only move Right or Down.
 */
public static int countPathsRecursive(int m, int n) {
    // Base Case: If we are on the first row or first col, only 1 path
    if (m == 0 || n == 0) {
        return 1;
    }

    // Recursive Step: Paths = (paths from above) + (paths from left)
    return countPathsRecursive(m - 1, n) +
           countPathsRecursive(m, n - 1);
}
```

That’s for today!

Any questions?