**Fundamental Data Structures:**

**1. Lists:**

- Ordered collections of elements, enclosed in square brackets (`[]`).
- Can contain elements of various data types.
- Support indexing, slicing, and various methods for manipulation.
- Mutable, allowing elements to be added, removed, or modified.

**2. Tuples:**

- Ordered collections of elements, enclosed in parentheses (`()`).
- Similar to lists, but **immutable**, meaning elements cannot be changed once created.
- Often used for storing fixed data.

**3. Dictionaries:**

- Unordered collections of key-value pairs, enclosed in curly braces (`{}`).
- Keys must be unique and immutable (often strings or numbers).
- Values can be any data type.
- Efficient for accessing elements by their keys.

**4. Sets:**

- Unordered collections of unique elements, enclosed in curly braces (`{}`) or using the `set()` constructor.
- Elements must be immutable.
- Useful for membership testing, removing duplicates, and performing set operations (union, intersection, difference).

**Advanced Data Structures:**

**1. Defaultdict:**

- A subclass of `dict` that automatically initializes missing keys with a default value.
- Useful for building collections where keys might not exist initially.

**2. OrderedDict:**

- A subclass of `dict` that remembers the order in which elements were added.
- Maintains insertion order, making it useful for iteration and preserving the order of keys.

**3. Counter:**

- A subclass of `dict` specifically designed for counting elements in a sequence.
- Stores elements as keys and their counts as values.

**4. NamedTuple:**

- A subclass of `tuple` that provides named fields for its elements.
- Enhances readability and makes accessing elements more intuitive.

**5. Deque (Double-Ended Queue):**

- A list-like container that supports efficient operations at both ends.
- Useful for implementing stacks, queues, and other data structures that require fast appending and popping from both sides.

**Choosing the Right Data Structure:**

The best data structure for a particular task depends on factors such as:

- **Order:** Do you need elements to be in a specific order?
- **Mutability:** Do you need to modify the elements?
- **Access:** How do you need to access elements (by index, key, or membership testing)?
- **Performance:** What are the time and space complexity requirements?

**Example:**

```
# Lists
fruits = ["apple", "banana", "orange"]
# Tuples
coordinates = (3, 4)
# Dictionaries
person = {"name": "Alice", "age": 30}
# Sets
unique_numbers = {1, 2, 3, 3, 4}
# Defaultdict
word_counts = defaultdict(int)
# OrderedDict
ordered_dict = OrderedDict([("a", 1), ("b", 2), ("c", 3)])
# Counter
word_frequencies = Counter("hello world")
# NamedTuple
Point = namedtuple("Point", ["x", "y"])
p = Point(2, 5)
```

Stacks are a fundamental data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed.

## Operations on Stacks

1. Push: Adds an element to the top of the stack.
2. Pop: Removes and returns the top element from the stack.
3. Peek: Returns the top element without removing it.
4. IsEmpty: Checks if the stack is empty.

## Implementation using a List

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("Stack is empty")
```

```
def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("Stack is empty")
```

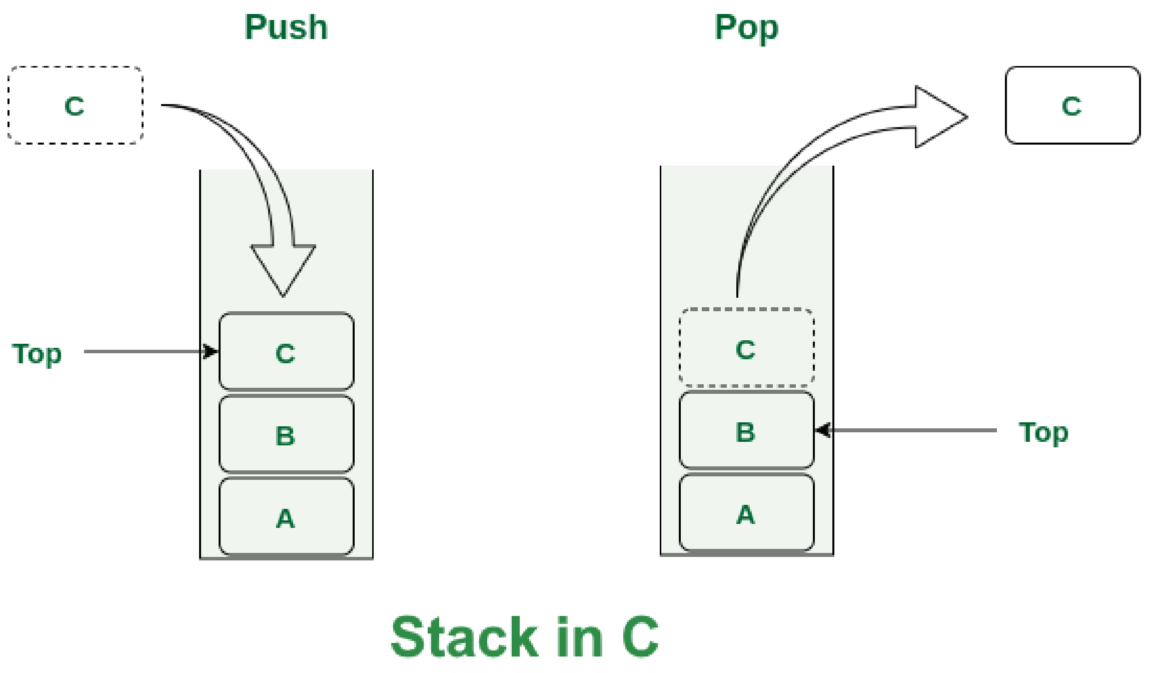
Example Usage

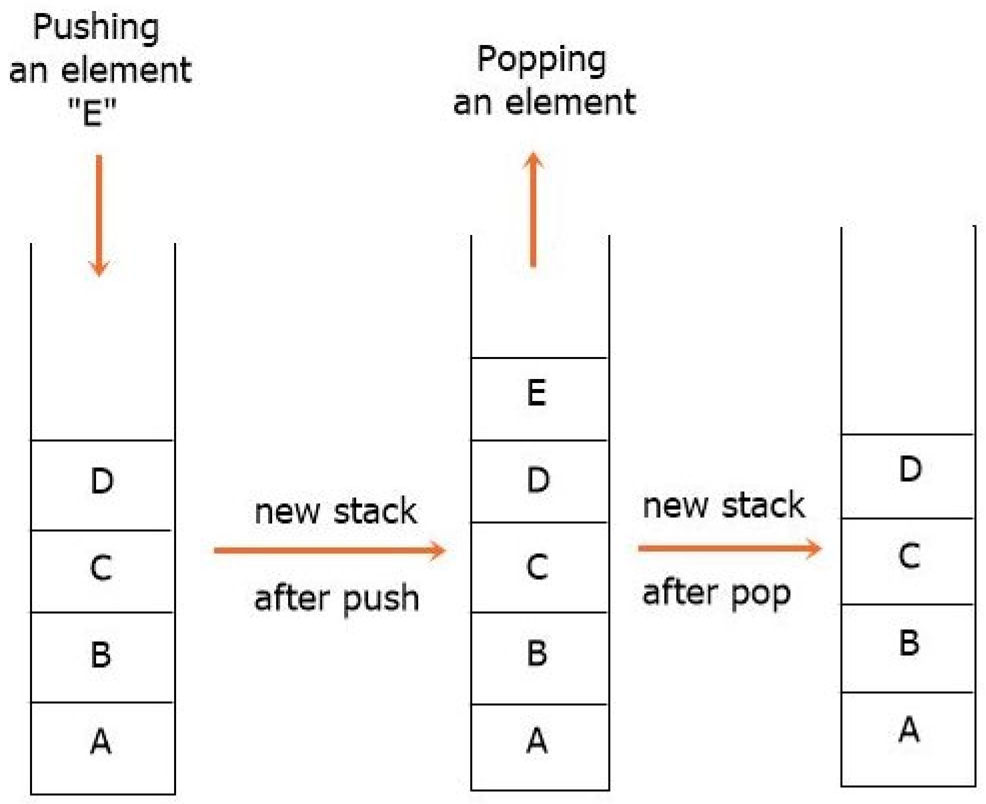
```
stack = Stack()

stack.push(1)
stack.push(2)
stack.push(3)

print(stack.peek())  # Output: 3
print(stack.pop())   # Output: 3
print(stack.is_empty())  # Output: False
```

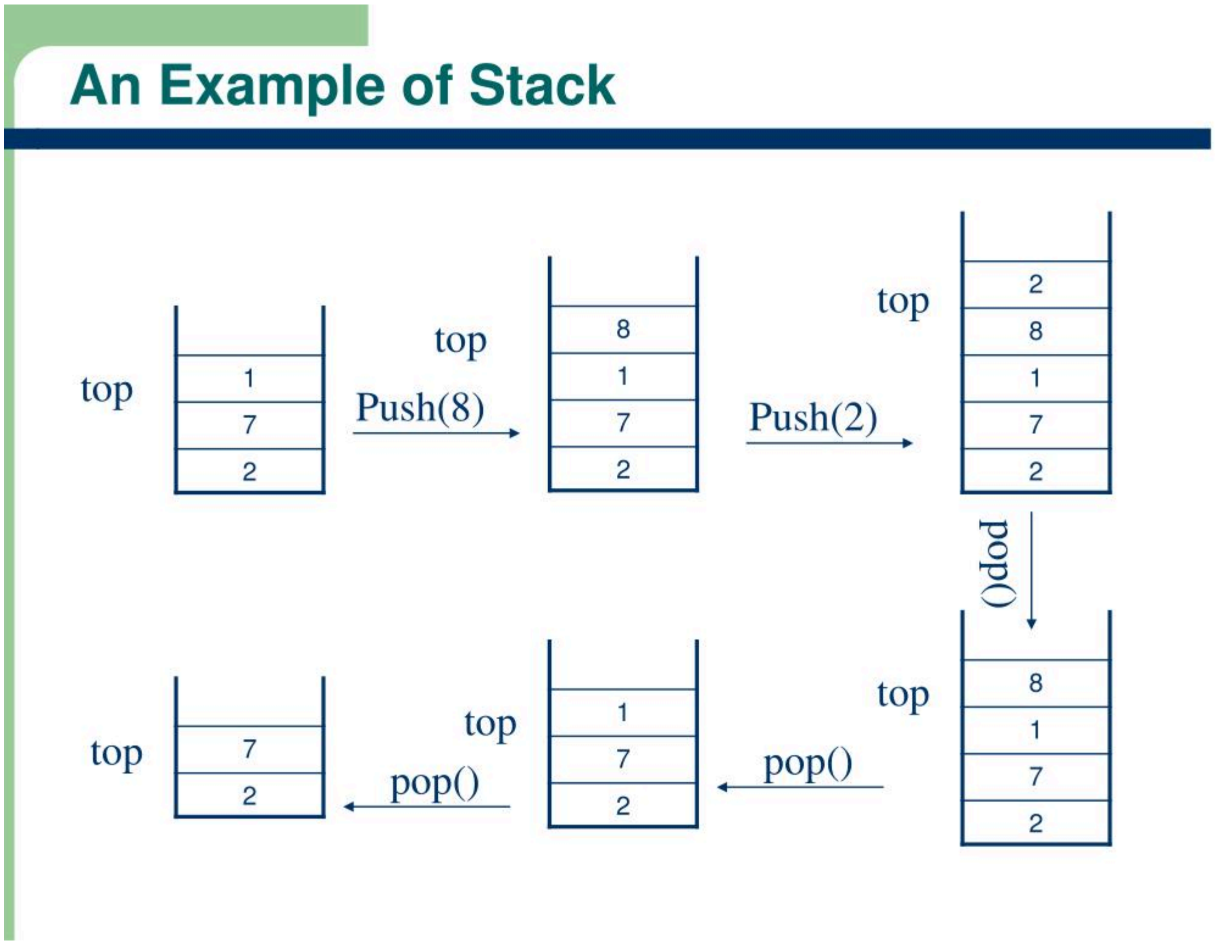
## Applications of Stacks

- **Function calls:** Stacks are used to store the return addresses and local variables of functions during recursive calls.
- **Expression evaluation:** Stacks are used to evaluate arithmetic expressions following the order of operations (PEMDAS).
- **Backtracking algorithms:** Stacks are used to store the states of a problem to allow backtracking to previous states if a solution is not found.
- **Undo/redo functionality:** Stacks can be used to store the previous states of a document or application, allowing users to undo or redo changes.

**Would you like to explore specific use cases or dive deeper into stack algorithms?**


Push
C
Top
C
B
A
Pop
C
C
B
Top
A
Stack in C


Pushing an element "E"
D
C
B
A
new stack
after push
Popping an element
E
D
C
B
A
new stack
after pop
D
C
B
A



## An Example of Stack

**Recursion** is a programming technique where a function calls itself directly or indirectly. This can be a powerful tool for solving certain problems, but it's essential to ensure the function has a base case to prevent infinite recursion.

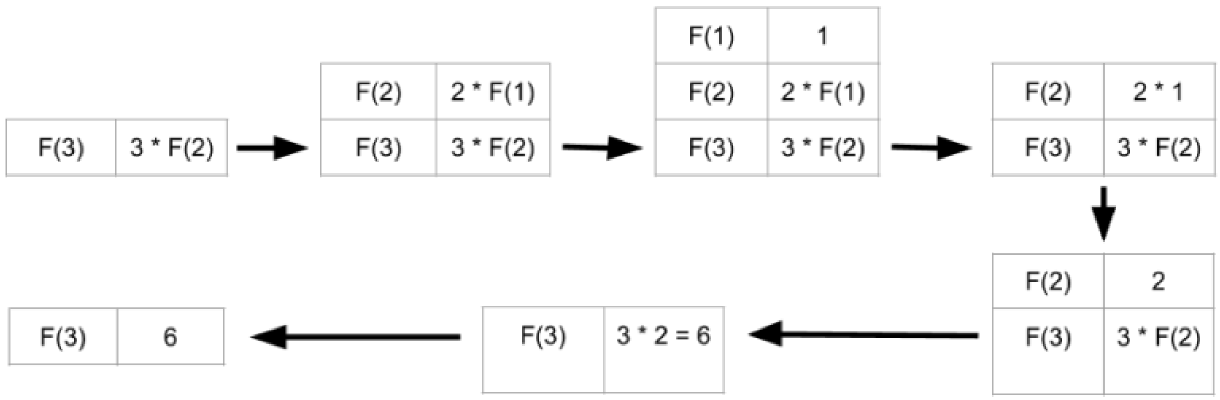
### Example: Factorial Calculation

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

**Explanation:**

1. **Base case:** If `n` is `0`, the function returns `1`.
2. **Recursive case:** If `n` is not `0`, the function calls itself with `n - 1` and multiplies the result by `n`.

`factorial(3)`



**Steps:**

1. **Initial call:** `factorial(5)` is called.
2. **Recursive calls:**
   - `factorial(4)` is called.
   - `factorial(3)` is called.
   - `factorial(2)` is called.
   - `factorial(1)` is called.
   - `factorial(0)` is called.
3. **Base case:** `factorial(0)` returns 1.
4. **Return values:**
   - `factorial(1)` returns 1 * 1 = 1.
   - `factorial(2)` returns 2 * 1 = 2.
   - `factorial(3)` returns 3 * 2 = 6.
   - `factorial(4)` returns 4 * 6 = 24.
   - `factorial(5)` returns 5 * 24 = 120.
   -

## Key Points

- **Base case:** Ensure the function has a base case to prevent infinite recursion.

- **Recursive case:** The function should call itself with a smaller input towards the base case.
- **Stack:** Recursive calls create a stack to store the return addresses and local variables of each function call.

## Evaluating Expressions:

**Expression evaluation** is the process of determining the value of a mathematical or logical expression. In programming, this often involves parsing the expression and applying the appropriate operators and operands.

### Example: Arithmetic Expression

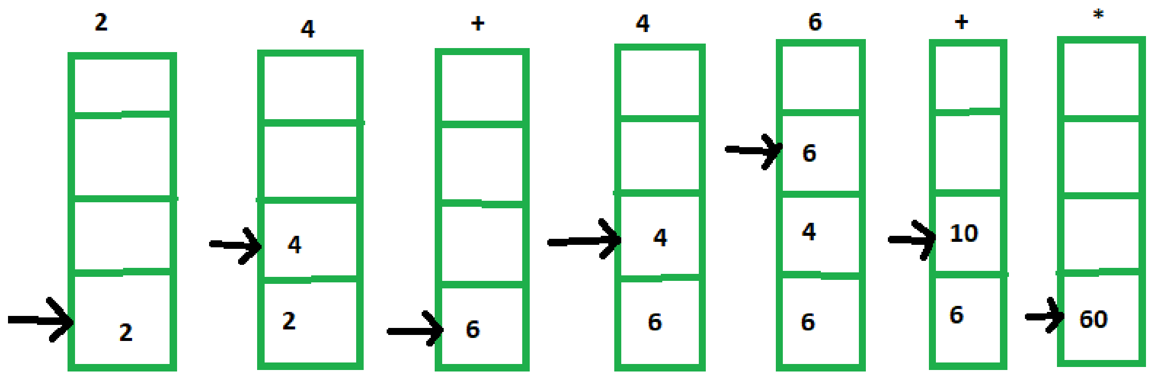
Consider the expression: `2 + 3 * 4`

**Order of Operations (PEMDAS):**

1. **Parentheses:** Evaluate expressions within parentheses first.
2. **Exponents:** Evaluate exponents next.
3. **Multiplication and Division:** Perform multiplication and division from left to right.
4. **Addition and Subtraction:** Perform addition and subtraction from left to right.
5.

**Evaluation:**

1. **Multiplication:** `3 * 4 = 12`
2. **Addition:** `2 + 12 = 14`



Stack operations to evaluate (2+4)*(4+6)

Infix notation: (2+4) * (4+6)

Post-fix notation: 2 4 + 4 6 + *

Result: 60

**Steps:**

1. **Tokenize the expression:** Break the expression into individual tokens (numbers and operators).
2. **Create a stack:** Use a stack to store operands and operators.
3. **Iterate through the tokens:**
    - If the token is an operand, push it onto the stack.
    - If the token is an operator:
        - Pop the top two operands from the stack.
        - Apply the operator to the operands.
        - 
        - Push the result back onto the stack.
4. **The final result:** The remaining element on the stack is the evaluated value of the expression.

## Code Example (Python)

```python
def evaluate_expression(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token.isdigit():
            stack.append(int(token))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            result = apply_operator(token, operand1, operand2)
            stack.append(result)
    return stack.pop()

def apply_operator(operator, operand1, operand2):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        return operand1 / operand2
    else:
        raise ValueError("Invalid operator")
```

**Note:** This is a simplified example. Real-world expression evaluation might involve more complex expressions, operator precedence rules, and error handling.

**Postfix Expression Example:**

Let's evaluate the postfix expression `5 6 + 2 * 3 -`, which corresponds to the infix expression `(5 + 6) * 2 - 3`.

**Steps:**

1. **Push operands** onto the stack until you encounter an operator.
2. **Pop** the required number of operands for the operator, perform the operation, and push the result back onto the stack.

```
def evaluate_postfix(expression):
    stack = []
    # Traverse through each token in the postfix expression
    for token in expression.split():
        if token.isdigit():
            # Push operand (number) to stack
            stack.append(int(token))
        else:
            # Pop two operands for an operator
            operand2 = stack.pop()
            operand1 = stack.pop()
            # Perform the operation based on the token (operator)
            if token == '+':
                result = operand1 + operand2
            elif token == '-':
                result = operand1 - operand2
            elif token == '*':
                result = operand1 * operand2
            elif token == '/':
                result = operand1 / operand2

            # Push the result back to the stack
            stack.append(result)

    # The final result is at the top of the stack
    return stack.pop()

# Postfix expression: 5 6 + 2 * 3 -
expression = "5 6 + 2 * 3 -"
print(evaluate_postfix(expression))  # Output: 19
```

## Stacks and Backtracking Algorithms

**Backtracking algorithms** are a problem-solving technique that explores possible solutions to a problem by trying different options and discarding those that don't lead to a solution. Stacks are often used to implement backtracking algorithms because they provide a convenient way to store and retrieve the current state of the exploration.

### How Stacks are Used in Backtracking

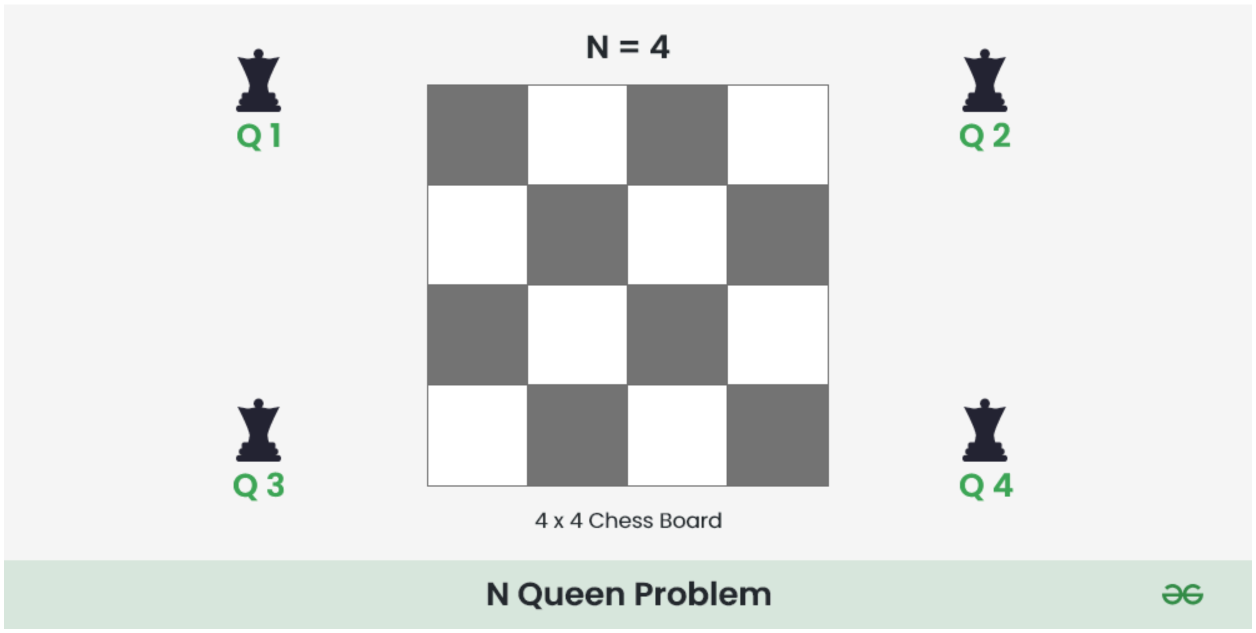
1. **Storing states:** Each state of the exploration is pushed onto the stack. This state might include the current solution, the remaining options, or other relevant information.
2. **Exploring options:** The algorithm explores one of the available options from the current state.
3. **Checking for solution:** If the current state leads to a solution, the algorithm returns the solution.
4. **Backtracking:** If the current state doesn't lead to a solution, the algorithm pops the current state from the stack and tries another option. This process continues until a solution is found or all options have been exhausted.

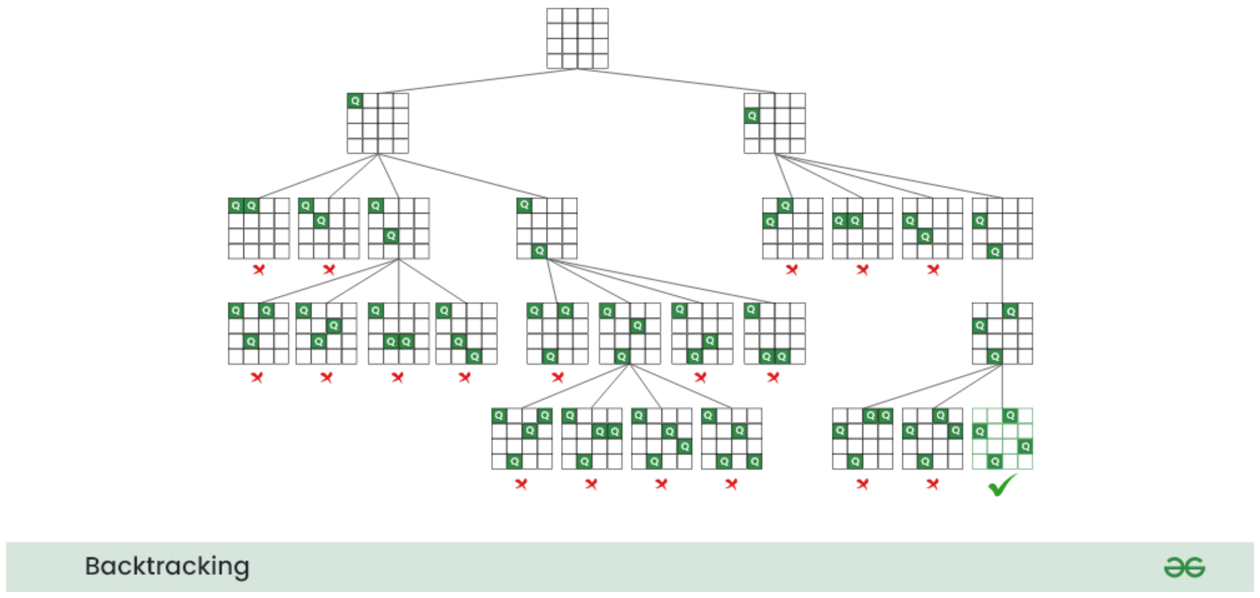
# A Visual Guide to Backtracking with Stacks: The N-Queens Problem

**Backtracking** is a problem-solving technique that explores possible solutions by trying different options and discarding those that don't lead to a solution. **Stacks** are often used to implement backtracking algorithms to store and retrieve the current state of the exploration.

## The N-Queens Problem

The N-Queens problem is a classic example of a problem that can be solved using backtracking. The goal is to place N queens on an N×N chessboard such that no queen attacks another.





**Steps:**

1. **Initialize the board:** Create an empty N×N chessboard.
2. **Place a queen:** Start at the first row and try to place a queen in each column.
3. **Check for conflicts:** If placing a queen doesn't create any conflicts with other queens, proceed to the next row.
4. **Backtrack:** If placing a queen creates a conflict, backtrack to the previous row and try a different column.
5. **Repeat:** Continue this process until all rows have been filled or no valid solution is found.

## Code Example (Python)

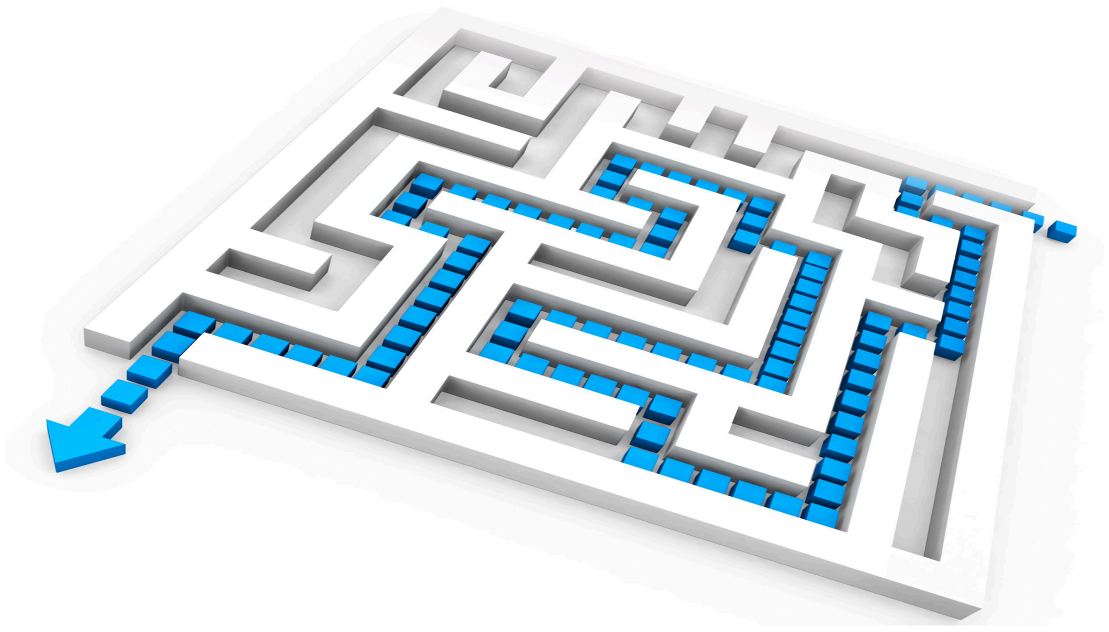
```
def solve_n_queens(n):
    board = [[-1] * n for _ in range(n)]
    col = [False] * n
    diag1 = [False] * (2 * n - 1)
    diag2 = [False] * (2 * n - 1)
    return solve_n_queens_util(board, col, diag1, diag2, 0)

def solve_n_queens_util(board, col, diag1, diag2, row):
    if row >= len(board):
        return True

    for col_idx in range(len(board)):
        if is_safe(board, col, diag1, diag2, row, col_idx):
            board[row][col_idx] = 1
            col[col_idx] = True
            diag1[row + col_idx] = True
            diag2[row - col_idx + len(board) - 1] = True

            if solve_n_queens_util(board, col, diag1, diag2, row + 1):
                return True

            board[row][col_idx] = -1
            col[col_idx] = False
            diag1[row + col_idx] = False
            diag2[row - col_idx + len(board) - 1] = False

    return False

def is_safe(board, col, diag1, diag2, row, col_idx):
    return (not col[col_idx] and
            not diag1[row + col_idx] and
            not diag2[row - col_idx + len(board) - 1])
```

In this example, the `solve_n_queens_util` function uses a stack implicitly to store the current state of the board, column, diagonal1, and diagonal2 arrays. When a solution is found or all options have been explored, the function backtracks by popping the current state from the implicit stack and trying another option.

**Key points:**

- Stacks provide a natural way to store and retrieve states in backtracking algorithms.
- Backtracking algorithms can be implemented recursively or iteratively using a stack.
- The choice of data structure for the stack depends on the specific requirements of the problem.

Maze solving is another problem that can be solved using backtracking. The goal is to find a path from the start point to the end point in a maze.



The backtracking algorithm would explore different paths in the maze, backtracking when a dead-end is reached. A stack can be used to store the current path and the previous positions visited.

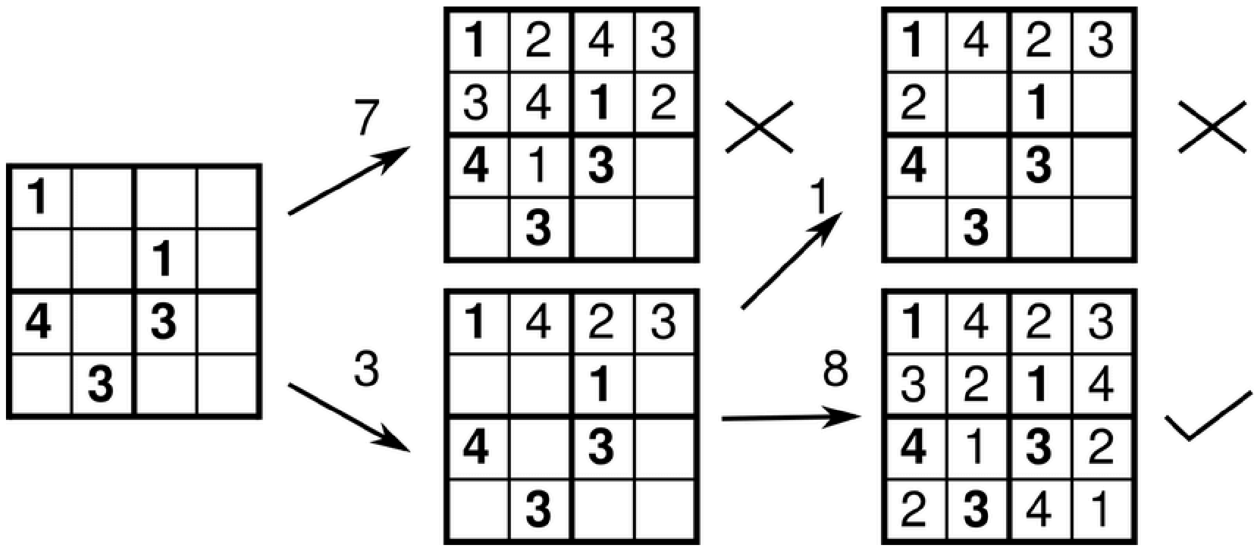
3)

## Sudoku Solver

**Sudoku** is a number-placement puzzle where the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids contains all the digits from 1 to 9.

## Backtracking Approach

1. **Start with an empty board:** Create a 9×9 grid filled with zeros.
2. **Find an empty cell:** Find the next empty cell (a cell with a value of 0).
3. **Try numbers:** For each number from 1 to 9, check if it's valid for the current cell. If it's valid, place the number in the cell and recursively try to solve the rest of the board.
4. **Backtrack:** If no number from 1 to 9 can be placed in the current cell without violating the rules, backtrack to the previous cell and try a different number.



```
def solve_sudoku(board):
    if not find_empty_cell(board):
        return True

    row, col = find_empty_cell(board)

    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num
            if solve_sudoku(board):
                return True
            board[row][col] = 0

    return False

def find_empty_cell(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None

def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num:
            return False
        if board[i][col] == num:
            return False

    start_row = (row // 3) * 3
    start_col = (col // 3) * 3

    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            if board[i][j] == num:
                return False

    return True
```