Unit 5

Okay, let's dive into Unit V, focusing on Backtracking algorithms.

Introduction to Backtracking

- **Concept:** Backtracking is a general algorithmic technique for solving problems, typically constraint satisfaction problems (CSPs) or optimization problems, by incrementally building candidates for the solutions and abandoning ("backtracking") a candidate as soon as it's determined that it cannot possibly lead to a valid or optimal solution. It explores the set of all possible solutions (the "state-space tree") using a depth-first search approach. If a partial solution violates the problem's constraints, the algorithm backtracks to the previous step and tries a different option.
- **Core Idea:**
  1. **Explore:** Start from a root/initial state and explore potential solutions path by path.
  2. **Check Constraints:** At each step, check if the current partial solution is valid and potentially leads to a full solution.
  3. **Prune:** If the current path is invalid or cannot lead to a solution, abandon this path (prune the search tree).
  4. **Backtrack:** Return to the previous decision point and try the next available option.
  5. **Solution Found:** If a full, valid solution is reached, record it. Continue searching if all solutions are needed, or stop if only one is required.
- **Analogy:** Imagine trying to solve a maze. You follow a path (explore). If you hit a dead end (constraint violated/cannot lead to solution), you retrace your steps back to the last junction where you had a choice (backtrack) and try a different path. You continue until you find the exit (solution) or explore all possible paths.
- **Use-Cases:** Solving puzzles (Sudoku, n-Queens), parsing languages, combinatorial optimization problems (Hamiltonian cycle, subset sum), generating permutations/combinations, artificial intelligence (game playing).
- **State-Space Tree:** Backtracking implicitly builds and explores a tree where:
  - The root represents the initial state (empty solution).
  - Nodes represent partial solutions.
  - Edges represent choices made to extend a partial solution.
  - Leaves represent either complete solutions or dead ends.

---

1. n-Queens Problem

1. Concept Explanation

- **What:** The n-Queens problem is the challenge of placing `n` non-attacking chess queens on an `n x n` chessboard. This means no two queens can be in the same row, column, or diagonal.
- **Why Backtracking:** This is a classic constraint satisfaction problem. We can try placing queens one by one, column by column (or row by row). If placing a queen in a certain position violates the non-attacking constraint with previously placed queens, we backtrack and try placing it in the next valid spot in the current column. If no spot in the current column works, we backtrack further to the previous column.
- **How:**
  1. Start placing queens from the leftmost column (column 0).
  2. Try placing a queen in the first row (row 0) of the current column.
  3. **Check Safety:** Check if this position `(row, col)` is safe from attacks by queens already placed in previous columns (columns 0 to `col-1`). A position is safe if no other queen is in the same row or on the same diagonals. (We don't need to check the column, as we place only one queen per column).
  4. **If Safe:**
     - Place the queen.
     - Recursively call the function to place a queen in the next column (`col + 1`).
     - **If the recursive call returns true (found a solution):** Return true.
     - **If the recursive call returns false (dead end):** Remove the queen (backtrack) and try the next row in the current column.
  5. **If Not Safe:** Try the next row in the current column.
  6. **If all rows tried:** If no row in the current column is safe, return false (triggering backtracking in the previous column).
  7. **Base Case:** If `col >= n` (all columns filled), a solution is found. Return true.

2. Real-World Analogies and Use-Cases

- **Analogy:** Scheduling `n` tasks, each requiring a specific resource (e.g., a time slot, a machine), where certain pairs of tasks conflict (cannot use the same resource or related resources simultaneously). You try scheduling task by task, backtracking if a placement creates a conflict.
- **Use-Cases:** While a direct application is rare, the backtracking technique used is fundamental in:
  - Constraint programming.
  - Resource allocation problems.
  - VLSI design (placing components on a chip).
  - Generating test cases.

3. Step-by-Step Solved Example

Problem: Solve the 4-Queens problem. Place 4 queens on a 4x4 board.

Board Representation: We can use a 1D array `board[n]` where `board[col]` stores the row number of the queen in column `col`.

Steps (Recursive Calls `solve(col)`):

1. `solve(0)`: Try placing queen in column 0.
   - Try row 0: `board[0]=0`. Is `(0, 0)` safe? Yes (first queen).
   - Call `solve(1)`: Try placing queen in column 1.
     - Try row 0: `board[1]=0`. Is `(1, 0)` safe? Check vs `(0, 0)`. Row 0 unsafe.
     - Try row 1: `board[1]=1`. Is `(1, 1)` safe? Check vs `(0, 0)`. Diagonal unsafe (`1-0 == 1-0`).
     - Try row 2: `board[1]=2`. Is `(1, 2)` safe? Check vs `(0, 0)`. Yes.
     - Call `solve(2)`: Try placing queen in column 2.
       - Try row 0: `board[2]=0`. Is `(2, 0)` safe? Check vs `(0, 0)` (Yes), vs `(1, 2)` (Diagonal unsafe `2-0 == 2-1` is false, `2-0 == 1-2` is false. Row 0 unsafe vs (0,0)). Row 0 unsafe.
       - Try row 1: `board[2]=1`. Is `(2, 1)` safe? Check vs `(0, 0)` (Yes), vs `(1, 2)` (Row 1 unsafe). Row 1 unsafe.
       - Try row 2: `board[2]=2`. Is `(2, 2)` safe? Check vs `(0, 0)` (Diagonal unsafe), vs `(1, 2)` (Row 2 unsafe). Row 2 unsafe.
       - Try row 3: `board[2]=3`. Is `(2, 3)` safe? Check vs `(0, 0)` (Yes), vs `(1, 2)` (Diagonal unsafe `2-1 == 3-2`). Row 3 unsafe.
       - No safe row in col 2. Return `false`.
     - Backtrack: Remove queen from `(1, 2)`.
     - Try row 3: `board[1]=3`. Is `(1, 3)` safe? Check vs `(0, 0)`. Yes.
     - Call `solve(2)`: Try placing queen in column 2.
       - Try row 0: `board[2]=0`. Is `(2, 0)` safe? Check vs `(0, 0)` (Row 0 unsafe). Row 0 unsafe.
       - Try row 1: `board[2]=1`. Is `(2, 1)` safe? Check vs `(0, 0)` (Yes), vs `(1, 3)` (Yes). Safe.
       - Call `solve(3)`: Try placing queen in column 3.
         - Try row 0: `board[3]=0`. Is `(3, 0)` safe? Check vs `(0, 0)` (Row 0 unsafe). Unsafe.
         - Try row 1: `board[3]=1`. Is `(3, 1)` safe? Check vs `(0, 0)` (Diag unsafe), vs `(1, 3)` (Diag unsafe), vs `(2, 1)` (Row 1 unsafe). Unsafe.
         - Try row 2: `board[3]=2`. Is `(3, 2)` safe? Check vs `(0, 0)` (Yes), vs `(1, 3)` (Diag unsafe), vs `(2, 1)` (Diag unsafe). Unsafe.
         - Try row 3: `board[3]=3`. Is `(3, 3)` safe? Check vs `(0, 0)` (Diag unsafe), vs `(1, 3)` (Row 3 unsafe), vs `(2, 1)` (Yes). Row 3 unsafe.
         - No safe row in col 3. Return `false`.
       - Backtrack: Remove queen from `(2, 1)`.
       - Try row 2: `board[2]=2`. Is `(2, 2)` safe? Check vs `(0, 0)` (Diag unsafe), vs `(1, 3)` (Yes). Diag unsafe.
       - Try row 3: `board[2]=3`. Is `(2, 3)` safe? Check vs `(0, 0)` (Yes), vs `(1, 3)` (Row 3 unsafe). Row 3 unsafe.

        - No safe row in col 2. Return `false`.
    - Backtrack: Remove queen from `(1, 3)`.
    - No more rows in col 1. Return `false`.
- Backtrack: Remove queen from `(0, 0)`.
- Try row 1: `board[0]=1`. Is `(0, 1)` safe? Yes.
- Call `solve(1)`:
    - Try row 0: `board[1]=0`. Is `(1, 0)` safe? Check vs `(0, 1)`. Yes.
    - Call `solve(2)`:
        - Try row 0: `board[2]=0`. Is `(2, 0)` safe? Check vs `(0, 1)` (Yes), vs `(1, 0)` (Row 0 unsafe). Unsafe.
        - Try row 1: `board[2]=1`. Is `(2, 1)` safe? Check vs `(0, 1)` (Row 1 unsafe). Unsafe.
        - Try row 2: `board[2]=2`. Is `(2, 2)` safe? Check vs `(0, 1)` (Diag unsafe), vs `(1, 0)` (Diag unsafe). Unsafe.
        - Try row 3: `board[2]=3`. Is `(2, 3)` safe? Check vs `(0, 1)` (Yes), vs `(1, 0)` (Yes). Safe.
        - Call `solve(3)`:
            - Try row 0: `board[3]=0`. Is `(3, 0)` safe? Check vs `(0, 1)` (Yes), vs `(1, 0)` (Row 0 unsafe). Unsafe.
            - Try row 1: `board[3]=1`. Is `(3, 1)` safe? Check vs `(0, 1)` (Row 1 unsafe). Unsafe.
            - Try row 2: `board[3]=2`. Is `(3, 2)` safe? Check vs `(0, 1)` (Diag unsafe), vs `(1, 0)` (Yes), vs `(2, 3)` (Diag unsafe). Unsafe.
            - Try row 3: `board[3]=3`. Is `(3, 3)` safe? Check vs `(0, 1)` (Yes), vs `(1, 0)` (Yes), vs `(2, 3)` (Row 3 unsafe). Unsafe.
            - No safe row in col 3. Return `false`.
        - Backtrack: Remove queen from `(2, 3)`.
        - No more rows in col 2. Return `false`.
    - Backtrack: Remove queen from `(1, 0)`.
    - Try row 1: `board[1]=1`. Is `(1, 1)` safe? Check vs `(0, 1)` (Row 1 unsafe). Unsafe.
    - Try row 2: `board[1]=2`. Is `(1, 2)` safe? Check vs `(0, 1)` (Diag unsafe). Unsafe.
    - Try row 3: `board[1]=3`. Is `(1, 3)` safe? Check vs `(0, 1)`. Yes.
    - Call `solve(2)`:
        - Try row 0: `board[2]=0`. Is `(2, 0)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Yes). Safe.
        - Call `solve(3)`:
            - Try row 0: `board[3]=0`. Is `(3, 0)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Yes), vs `(2, 0)` (Row 0 unsafe). Unsafe.
            - Try row 1: `board[3]=1`. Is `(3, 1)` safe? Check vs `(0, 1)` (Row 1 unsafe). Unsafe.
            - Try row 2: `board[3]=2`. Is `(3, 2)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Diag unsafe), vs `(2, 0)` (Yes). Diag unsafe.

        - Try row 3: `board[3]=3`. Is `(3, 3)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Row 3 unsafe). Unsafe.
        - No safe row in col 3. Return `false`.
      - Backtrack: Remove queen from `(2, 0)`.
      - Try row 1: `board[2]=1`. Is `(2, 1)` safe? Check vs `(0, 1)` (Row 1 unsafe). Unsafe.
      - Try row 2: `board[2]=2`. Is `(2, 2)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Diag unsafe). Unsafe.
      - Try row 3: `board[2]=3`. Is `(2, 3)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Row 3 unsafe). Unsafe.
      - No safe row in col 2. Return `false`.
    - Backtrack: Remove queen from `(1, 3)`.
    - No more rows in col 1. Return `false`.
  - Backtrack: Remove queen from `(0, 1)`.

... This continues. Eventually, a solution is found:

- `solve(0)` -> `board[0]=1`
- `solve(1)` -> `board[1]=3`
- `solve(2)` -> `board[2]=0`
- `solve(3)` -> `board[3]=2`. Is `(3, 2)` safe? Check vs `(0, 1)` (Yes), vs `(1, 3)` (Yes), vs `(2, 0)` (Yes). Safe.
- Call `solve(4)`. Base case `col >= n` (4 >= 4). Return `true`.
- `solve(3)` returns `true`.
- `solve(2)` returns `true`.
- `solve(1)` returns `true`.
- `solve(0)` returns `true`.

Solution Found: `board = {1, 3, 0, 2}`.
Board:

```
. . Q .
Q . . .
. . . Q
. Q . .
```

(Rows are 0-3 top to bottom, Cols 0-3 left to right. Q at (0,1), (1,3), (2,0), (3,2))

4. Pseudocode and Code Implementation

Pseudocode:

```
function NQueens(n):
  board = array[n] // board[col] stores row of queen in that column
```

```
  if solveNQueensUtil(board, 0, n) == false:
    print "Solution does not exist"
    return false
  printSolution(board, n)
  return true

function solveNQueensUtil(board, col, n):
  // Base case: All queens placed
  if col >= n:
    return true // Found a solution

  // Try placing queen in each row of the current column 'col'
  for row from 0 to n-1:
    // Check if queen can be placed at board[col] = row
    if isSafe(board, row, col, n):
      // Place queen
      board[col] = row

      // Recur to place rest of the queens
      if solveNQueensUtil(board, col + 1, n) == true:
        return true // Solution found

      // If placing queen in board[col] = row doesn't lead to a solution,
      // then remove queen (BACKTRACK)
      // (No explicit removal needed if board[col] is overwritten in next iteration)
      // board[col] = -1; // Optional: Mark as empty if needed

  // If queen cannot be placed in any row in this column
  return false

function isSafe(board, row, col, n):
  // Check this row on left side (previous columns)
  // Check upper diagonal on left side
  // Check lower diagonal on left side
  for prev_col from 0 to col-1:
    prev_row = board[prev_col]
    // Check row conflict
    if prev_row == row:
      return false
    // Check diagonal conflict (absolute difference in rows == absolute difference in
columns)
    if abs(prev_row - row) == abs(prev_col - col):
      return false

  // If no conflicts found
  return true

// Function to print the board (optional)
function printSolution(board, n):
  // ... implementation ...
```

Python Implementation:

```
def solve_n_queens(n):
    """Finds one solution to the N-Queens problem."""
    board = [-1] * n # board[col] = row
    solutions = [] # To store all solutions if needed

    def is_safe(r, c):
        # Check previous columns (0 to c-1)
        for prev_c in range(c):
            prev_r = board[prev_c]
            # Check row conflict
            if prev_r == r:
                return False
            # Check diagonal conflict
            if abs(prev_r - r) == abs(prev_c - c):
                return False
        return True

    def solve_util(col):
        # Base case: All columns filled
        if col == n:
            # Found a solution, make a copy
            solutions.append(list(board))
            return True # Return True if only one solution needed, or continue

        # Try placing queen in each row of current column
        found_solution = False
        for row in range(n):
            if is_safe(row, col):
                board[col] = row # Place queen

                # Recur for next column
                # If we only need one solution:
                if solve_util(col + 1):
                     return True # Stop after first solution
                # If we need all solutions, continue search:
                # solve_util(col + 1)

                # Backtrack (implicitly done by overwriting board[col] in next loop
iter)
                # board[col] = -1 # Optional explicit backtrack

        return False # No solution found from this column state if only one needed

    # Start solving from column 0
    solve_util(0) # Find all solutions
    # if not solve_util(0): # Find one solution
```

```
    #     print("No solution exists")
    # else:
    #     print("Solution found:")
    #     print_board(board)

    return solutions # Return all found solutions

def print_board(board_config):
    n = len(board_config)
    for r in range(n):
        line = ""
        queen_col = -1
        for c_idx, r_idx in enumerate(board_config):
            if r_idx == r:
                queen_col = c_idx
                break
        for c in range(n):
            line += "Q " if c == queen_col else ". "
        print(line)
    print("-" * (2*n))


# Example Usage:
n_val = 4
all_solutions = solve_n_queens(n_val)

if not all_solutions:
    print(f"No solution exists for N={n_val}")
else:
    print(f"Found {len(all_solutions)} solutions for N={n_val}:")
    for sol in all_solutions:
        # Convert board array to coordinates if needed, or print directly
        print(f"  Board array: {sol}")
        print_board(sol)

# Output for N=4:
# Found 2 solutions for N=4:
#   Board array: [1, 3, 0, 2]
# . . Q .
# Q . . .
# . . . Q
# . Q . .
# --------
#   Board array: [2, 0, 3, 1]
# . Q . .
# . . . Q
# Q . . .
# . . Q .
# --------
```

C++ Implementation:

```
#include <vector>
#include <cmath> // For std::abs
#include <iostream>

using namespace std;

// Function to print the board
void printBoard(const vector<int>& board) {
    int n = board.size();
    for (int r = 0; r < n; ++r) {
        for (int c = 0; c < n; ++c) {
            if (board[c] == r) {
                cout << "Q ";
            } else {
                cout << ". ";
            }
        }
        cout << endl;
    }
    cout << string(2 * n, '-') << endl;
}

// Function to check if placing queen at board[col] = row is safe
bool isSafe(const vector<int>& board, int row, int col) {
    int n = board.size();
    // Check previous columns (0 to col-1)
    for (int prev_col = 0; prev_col < col; ++prev_col) {
        int prev_row = board[prev_col];
        // Check row conflict
        if (prev_row == row) {
            return false;
        }
        // Check diagonal conflict
        if (abs(prev_row - row) == abs(prev_col - col)) {
            return false;
        }
    }
    return true;
}

// Recursive utility function to solve N-Queens
// Returns true if a solution is found (if finding only one)
// Modifies solutions_list to store all solutions
bool solveNQueensUtil(vector<int>& board, int col, int n, vector<vector<int>>&
solutions_list, bool find_all) {
    // Base case: All queens placed
    if (col == n) {
```

```
        solutions_list.push_back(board); // Store the found solution
        return true; // Indicate a solution was found
    }

    bool found_solution = false;
    // Try placing queen in each row of the current column 'col'
    for (int row = 0; row < n; ++row) {
        if (isSafe(board, row, col)) {
            board[col] = row; // Place queen

            // Recur to place rest of the queens
            if (solveNQueensUtil(board, col + 1, n, solutions_list, find_all)) {
                found_solution = true;
                if (!find_all) { // If only one solution is needed, stop
                    return true;
                }
                // If finding all, continue searching after finding one
            }

            // Backtrack (implicitly handled by overwriting board[col] in next
iteration)
            // board[col] = -1; // Optional explicit backtrack
        }
    }

    return found_solution; // Return true if any solution was found down this path
}

// Main function to solve N-Queens
vector<vector<int>> solveNQueens(int n) {
    vector<int> board(n, -1); // board[col] = row, initialized to -1
    vector<vector<int>> solutions;
    solveNQueensUtil(board, 0, n, solutions, true); // Find all solutions
    return solutions;
}

int main() {
    int n_val = 4;
    vector<vector<int>> all_solutions = solveNQueens(n_val);

    if (all_solutions.empty()) {
        cout << "No solution exists for N=" << n_val << endl;
    } else {
        cout << "Found " << all_solutions.size() << " solutions for N=" << n_val << ":"
<< endl;
        for (const auto& sol : all_solutions) {
            // Print the board configuration array
            cout << "  Board array: [";
            for(size_t i = 0; i < sol.size(); ++i) cout << sol[i] << (i == sol.size()-1
? "" : ", ");
```

```
            cout << "]" << endl;
            printBoard(sol);
        }
    }
    return 0;
}
```

5. Diagrams or Flowcharts

State-Space Tree (Partial for 4-Queens):

```
                  Root
                 / | \ \
Col 0:       Q@0 Q@1 Q@2 Q@3  (Place Q in row 0, 1, 2, or 3 of col 0)
             |   |   ...
Col 1:     (Q@0) | (Q@1)
           / | \   / | \ \
         Q@? Q@? Q@? Q@? Q@? Q@? (Try placing Q in col 1, check safety vs col 0)
         |   X   |   |   X   |   (X = Pruned/Unsafe)
Col 2:   ...   ... ... ... ...
         |
Col 3:   ...
         |
Col 4: Solution Found (if path valid) or Dead End
```

The diagram shows the recursive exploration. Each level corresponds to a column, and branches correspond to trying different rows. Pruning occurs when `isSafe` returns false. Backtracking occurs when a recursive call returns false.

Board Diagram: (See example solution board)

6. Key Formulas and Logic Summary

- **Representation:** `board[col] = row` (1D array) or `board[row][col]` (2D array). 1D is often simpler.
- **Core Logic:** Place queens column by column (or row by row).
- **Constraint Check (`isSafe`):** For queen at `(row, col)`, check against previously placed queens `(prev_row, prev_col)` where `prev_col < col`:
  - Row conflict: `prev_row == row`
  - Diagonal conflict: `abs(prev_row - row) == abs(prev_col - col)`
- **Recursion:** `solve(col)` tries all rows for column `col`, calls `solve(col + 1)` if safe.
- **Base Case:** `col == n` (successfully placed queens in all columns).
- **Backtracking:** Implicitly happens when a loop finishes without finding a safe spot, causing the function to return `false`. Explicitly happens if we undo the placement (`board[col] = -1`) after a recursive call returns `false` (though often not needed if the value is simply overwritten later).

7. Common Mistakes and Edge Cases

- **Incorrect `isSafe` Logic:** Missing checks for rows or diagonals, or checking incorrectly (e.g., checking columns, which is unnecessary with the `board[col]=row` approach). Off-by-one errors in diagonal checks.
- **Base Case:** Incorrect base case (`col > n` instead of `col == n`).
- **Backtracking:** Forgetting to backtrack (if explicit removal is needed) or backtracking incorrectly, leading to incorrect solutions or infinite loops.
- **Shallow Copies:** When storing solutions (especially if finding all solutions), ensure you store a *copy* of the board state, not a reference to the board being modified by backtracking.
- **N=1:** Should place one queen (trivial solution).
- **N=2, N=3:** No solutions exist. The code should handle this correctly by returning an empty solution set or indicating failure.

8. MCQ/Short Questions and Answers

1. **Q:** What is the objective of the n-Queens problem?
   A: To place `n` queens on an `n x n` chessboard such that no two queens threaten each other (same row, column, or diagonal).
2. **Q:** Why is backtracking suitable for the n-Queens problem?
   A: It's a constraint satisfaction problem where partial solutions can be easily checked for validity, allowing invalid paths to be pruned early.
3. **Q:** In the `board[col] = row` representation, what conflicts does the `isSafe(row, col)` function need to check against previously placed queens (at `prev_col < col`)?
   A: Row conflicts (`board[prev_col] == row`) and diagonal conflicts (`abs(board[prev_col] - row) == abs(prev_col - col)`). Column conflicts are implicitly avoided.
4. **Q:** What is the base case for the recursive n-Queens solver function `solve(col)` ?
   A: When `col == n` (all columns have been successfully filled with queens).
5. **Q:** Does a solution exist for n=2 or n=3?
   A: No.

---

2. Hamiltonian Circuit Problem

1. Concept Explanation

- **What:** Given a graph G=(V, E), a Hamiltonian Circuit (or Cycle) is a path that visits each vertex in V exactly once and returns to the starting vertex. The Hamiltonian Circuit Problem asks whether such a circuit exists in a given graph. (A related problem is the Hamiltonian Path problem, which visits each vertex exactly once but doesn't need to return to the start).
- **Why Backtracking:** We can try building a path vertex by vertex. Starting from a vertex, we explore adjacent vertices. We add a vertex to the current path only if it hasn't been visited yet. If we get stuck

(cannot extend the path to an unvisited vertex), we backtrack and try a different neighbor. If we successfully build a path visiting all vertices, we check if the last vertex in the path can connect back to the starting vertex to form the circuit.

- **How:**
  1. Start at an arbitrary vertex (say, vertex 0). Add it to the current path. Mark it as visited.
  2. Try adding the next vertex: Iterate through vertices adjacent to the *last* vertex currently in the path.
  3. **Check Validity:** For an adjacent vertex `v`:
     - Is `v` already in the path? If yes, skip it.
     - If no, add `v` to the path and mark it as visited.
  4. **Recur:** Recursively call the function to find the next vertex in the path (from `v`).
  5. **If Recursive Call Succeeds:** If the recursive call eventually finds a complete circuit, return true.
  6. **If Recursive Call Fails:** Remove `v` from the path (backtrack) and unmark it as visited. Try the next adjacent vertex.
  7. **Base Case / Circuit Check:** When the path contains `n` vertices:
     - Check if the last vertex added is adjacent to the starting vertex (vertex 0).
     - If yes, a Hamiltonian Circuit is found. Return true.
     - If no, this path doesn't form a circuit. Return false (triggering backtracking).
  8. **If all neighbors tried:** If no valid neighbor leads to a solution from the current vertex, return false.

2. Real-World Analogies and Use-Cases

- **Analogy:** The "Traveling Salesperson Problem" (TSP) asks for the *shortest* Hamiltonian circuit. The basic Hamiltonian Circuit problem is like asking: Can a salesperson visit every city on their list exactly once and return home, regardless of the distance? Planning a route for a delivery truck that must visit specific locations once.
- **Use-Cases:**
  - TSP is a major application area (though finding the *shortest* circuit is harder, checking *existence* uses similar ideas).
  - Genome sequencing (arranging DNA fragments).
  - Planning routes for tasks (e.g., circuit board drilling, plotter drawing).
  - Network design and analysis.

3. Step-by-Step Solved Example

Problem: Find a Hamiltonian Circuit in the following graph, starting at vertex 0.

Adjacency Matrix: (1 if edge exists, 0 otherwise)

```
   0 1 2 3 4 5
0 [0 1 0 1 1 0]
1 [1 0 1 0 1 0]
2 [0 1 0 0 1 1]
3 [1 0 0 0 1 0]
4 [1 1 1 1 0 1]
5 [0 0 1 0 1 0]
```

Path Representation: `path` array. `visited` array/set.

Steps (Recursive Calls `solve(k)` where `k` is the index in the path being filled):

1. `solve(1)`: (Path starts with `path[0]=0`, `visited={0}`). Try filling `path[1]`.
    - Neighbors of `path[0]=0`: 1, 3, 4.
    - Try vertex 1: `path[1]=1`. `visited={0, 1}`.
    - Call `solve(2)`: Try filling `path[2]`. Neighbors of `path[1]=1`: 0, 2, 4.
        - Try vertex 0: Visited. Skip.
        - Try vertex 2: `path[2]=2`. `visited={0, 1, 2}`.
        - Call `solve(3)`: Try filling `path[3]`. Neighbors of `path[2]=2`: 1, 4, 5.
            - Try vertex 1: Visited. Skip.
            - Try vertex 4: `path[3]=4`. `visited={0, 1, 2, 4}`.
            - Call `solve(4)`: Try filling `path[4]`. Neighbors of `path[3]=4`: 0, 1, 2, 3, 5.
                - Try vertex 0: Visited. Skip.
                - Try vertex 1: Visited. Skip.
                - Try vertex 2: Visited. Skip.
                - Try vertex 3: `path[4]=3`. `visited={0, 1, 2, 4, 3}`.
                - Call `solve(5)`: Try filling `path[5]`. Neighbors of `path[4]=3`: 0, 4.
                    - Try vertex 0: Visited. Skip.
                    - Try vertex 4: Visited. Skip.
                    - No unvisited neighbors. Need vertex 5. Backtrack. Return `false`.
                - Backtrack: Remove 3. `visited={0, 1, 2, 4}`.
                - Try vertex 5: `path[4]=5`. `visited={0, 1, 2, 4, 5}`.
                - Call `solve(5)`: Try filling `path[5]`. Neighbors of `path[4]=5`: 2, 4.
                    - Try vertex 2: Visited. Skip.
                    - Try vertex 4: Visited. Skip.
                    - No unvisited neighbors. Need vertex 3. Backtrack. Return `false`.
                - Backtrack: Remove 5. `visited={0, 1, 2, 4}`.
                - No more neighbors for `path[3]=4`. Return `false`.
            - Backtrack: Remove 4. `visited={0, 1, 2}`.
            - Try vertex 5: `path[3]=5`. `visited={0, 1, 2, 5}`.
            - Call `solve(4)`: Try filling `path[4]`. Neighbors of `path[3]=5`: 2, 4.

- Try vertex 2: Visited. Skip.
- Try vertex 4: `path[4]=4`. `visited={0, 1, 2, 5, 4}`.
- Call `solve(5)`: Try filling `path[5]`. Neighbors of `path[4]=4`: 0, 1, 2, 3, 5.
  - Try vertex 0: Visited. Skip.
  - Try vertex 1: Visited. Skip.
  - Try vertex 2: Visited. Skip.
  - Try vertex 3: `path[5]=3`. `visited={0, 1, 2, 5, 4, 3}`.
  - Call `solve(6)`: Base Case Check. Path length is 6 (n=6).
    - Is last vertex `path[5]=3` adjacent to start `path[0]=0`? Yes (edge (3,0)).
    - Hamiltonian Circuit Found! Return `true`.
  - `solve(5)` returns `true`.
- `solve(4)` returns `true`.
- `solve(3)` returns `true`.
- `solve(2)` returns `true`.
- `solve(1)` returns `true`.

Solution Found: Path: 0 -> 1 -> 2 -> 5 -> 4 -> 3. Check edge (3, 0) exists. Yes.
Circuit: **0 - 1 - 2 - 5 - 4 - 3 - 0**

4. Pseudocode and Code Implementation

Pseudocode:

```
function HamiltonianCircuit(Graph G):
  n = number of vertices
  path = array[n]
  visited = array[n] initialized to false

  // Start from vertex 0
  path[0] = 0
  visited[0] = true

  if solveHCUtil(G, path, visited, 1) == false: // Start filling from path index 1
    print "No Hamiltonian Circuit exists"
    return false
  else:
    print "Hamiltonian Circuit found:"
    printPath(path)
    return true

function solveHCUtil(G, path, visited, k): // k = current path index to fill
  n = number of vertices

  // Base case: Path contains n vertices
  if k == n:
```

```
    // Check if last vertex connects back to the start vertex
    last_vertex = path[k-1]
    start_vertex = path[0]
    if G.hasEdge(last_vertex, start_vertex):
      return true // Circuit found
    else:
      return false // Path found, but not a circuit

  // Try adding next vertex to the path
  last_vertex_in_path = path[k-1]
  for v from 0 to n-1:
    // Check if v is adjacent to the last vertex AND not already visited
    if G.hasEdge(last_vertex_in_path, v) and visited[v] == false:
      // Choose v
      path[k] = v
      visited[v] = true

      // Recur for the next position in path
      if solveHCUtil(G, path, visited, k + 1) == true:
        return true // Circuit found down this path

      // Backtrack: If choosing v didn't lead to a solution
      visited[v] = false
      // path[k] will be overwritten in next iteration, no need to reset explicitly

  // If no vertex leads to a solution from this state
  return false

// Function to print the path (optional)
function printPath(path):
  // ... implementation ... print path[0] -> path[1] -> ... -> path[n-1] -> path[0]
```

Python Implementation:

```python
class GraphHC:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0] * vertices for _ in range(vertices)] # Adjacency Matrix
        self.path = [-1] * vertices

    def add_edge(self, u, v):
        if 0 <= u < self.V and 0 <= v < self.V:
            self.graph[u][v] = 1
            self.graph[v][u] = 1 # Assuming undirected graph

    def is_safe(self, v, pos):
        # Check if vertex v can be added at index 'pos' in the path
        # 1. Check if v is adjacent to the previous vertex in path
        if self.graph[self.path[pos - 1]][v] == 0:
```

```
            return False
        # 2. Check if v has already been included in the path
        for i in range(pos):
            if self.path[i] == v:
                return False
        return True

    def solve_hc_util(self, pos):
        # Base case: If all vertices are included in the path
        if pos == self.V:
            # Check if the last vertex is connected to the first vertex
            if self.graph[self.path[pos - 1]][self.path[0]] == 1:
                return True # Hamiltonian Circuit found
            else:
                return False # Path found, but not a circuit

        # Try different vertices as the next candidate in the path
        for v in range(self.V): # Try all vertices
            if self.is_safe(v, pos):
                self.path[pos] = v # Add v to path

                # Recur to construct the rest of the path
                if self.solve_hc_util(pos + 1):
                    return True

                # Backtrack: If adding vertex v doesn't lead to a solution
                self.path[pos] = -1 # Remove v (optional if overwritten)

        return False # No vertex can be added from this state

    def find_hamiltonian_circuit(self):
        # Start with vertex 0 as the first vertex in the path
        self.path[0] = 0

        if not self.solve_hc_util(1):
            print("Solution does not exist")
            return False
        else:
            print("Hamiltonian Circuit Found:")
            self.print_solution()
            return True

    def print_solution(self):
        for i in range(self.V):
            print(self.path[i], end=" -> ")
        print(self.path[0]) # Print start node again to show circuit


# Example Usage:
g = GraphHC(6)
```

```
# Edges from example:
g.add_edge(0, 1)
g.add_edge(0, 3)
g.add_edge(0, 4)
g.add_edge(1, 2)
g.add_edge(1, 4)
g.add_edge(2, 4)
g.add_edge(2, 5)
g.add_edge(3, 4)
g.add_edge(4, 5)

g.find_hamiltonian_circuit()

# Output:
# Hamiltonian Circuit Found:
# 0 -> 1 -> 2 -> 5 -> 4 -> 3 -> 0
```

C++ Implementation:

```
#include <vector>
#include <iostream>
#include <numeric> // For std::fill

using namespace std;

class GraphHC_cpp {
private:
    int V;
    vector<vector<bool>> adj; // Adjacency matrix (or list)
    vector<int> path;
    vector<bool> visited; // Alternative to checking path array

    // Check if vertex v can be added at index 'k'
    bool isSafe(int v, int k) {
        // Check adjacency with previous vertex
        if (!adj[path[k - 1]][v]) {
            return false;
        }
        // Check if already visited
        if (visited[v]) {
            return false;
        }
        return true;
    }

    bool solveHCUtil(int k) {
        // Base case: Path complete
        if (k == V) {
            // Check if last vertex connects to start
```

```
            return adj[path[k - 1]][path[0]];
        }

        // Try adding next vertex
        for (int v = 0; v < V; ++v) {
             // Use visited array for check
             if (adj[path[k-1]][v] && !visited[v]) {
                path[k] = v;
                visited[v] = true;

                if (solveHCUtil(k + 1)) {
                    return true;
                }

                // Backtrack
                visited[v] = false;
                path[k] = -1; // Optional reset
             }
        }
        return false;
    }

public:
    GraphHC_cpp(int vertices) : V(vertices) {
        adj.resize(V, vector<bool>(V, false));
        path.resize(V, -1);
        visited.resize(V, false);
    }

    void addEdge(int u, int v) {
        if (u >= 0 && u < V && v >= 0 && v < V) {
            adj[u][v] = true;
            adj[v][u] = true; // Undirected
        }
    }

    void printSolution() {
        for (int i = 0; i < V; ++i) {
            cout << path[i] << " -> ";
        }
        cout << path[0] << endl;
    }

    bool findHamiltonianCircuit() {
        // Start at vertex 0
        path[0] = 0;
        visited[0] = true;

        if (!solveHCUtil(1)) {
            cout << "Solution does not exist" << endl;
```

```
            return false;
        } else {
            cout << "Hamiltonian Circuit Found:" << endl;
            printSolution();
            return true;
        }
    }
};


int main() {
    GraphHC_cpp g(6);
    // Edges from example:
    g.addEdge(0, 1); g.addEdge(0, 3); g.addEdge(0, 4);
    g.addEdge(1, 2); g.addEdge(1, 4);
    g.addEdge(2, 4); g.addEdge(2, 5);
    g.addEdge(3, 4);
    g.addEdge(4, 5);

    g.findHamiltonianCircuit();

    return 0;
}
```

5. Diagrams or Flowcharts

State-Space Tree (Partial):

```
                    Root (Path=[0], k=1)
                   /       |       \
      k=1:       Path=[0,1] Path=[0,3] Path=[0,4] (Try neighbors of 0)
                   / | \      |       / | \
      k=2: [0,1,2] [0,1,4] [0,3,4] [0,4,1] [0,4,2] [0,4,3] [0,4,5] (Try neighbors of
last node, if unvisited)
           / | \      ...
      k=3: [0,1,2,4] [0,1,2,5]
           / | \      / | \
      k=4: ...      [0,1,2,5,4]
                     / | \
      k=5:         [0,1,2,5,4,3]
                      |
      k=6 (Base):   Check edge (3,0)? Yes -> Solution Found! Return True.
```

The tree shows building the path step-by-step (`k`). Branches are trying valid, unvisited neighbors. Backtracking occurs when a path gets stuck or completes but doesn't form a circuit.

6. Key Formulas and Logic Summary

- **Goal:** Find a path visiting each vertex exactly once, returning to the start.
- **Representation:** `path` array stores the sequence of vertices. `visited` array/set tracks visited vertices for O(1) lookup. Adjacency matrix or list for graph structure.
- **Core Logic:** Extend the current path `path[0...k-1]` by trying unvisited neighbors `v` of `path[k-1]`.
- **Constraint Check (`isSafe` or inline):**
  1. Is `v` adjacent to `path[k-1]`?
  2. Is `v` already visited (i.e., in `path[0...k-1]`)?
- **Recursion:** `solve(k)` tries all valid vertices `v` for `path[k]`, calls `solve(k + 1)`.
- **Base Case:** `k == n`. Check if `path[n-1]` is adjacent to `path[0]`.
- **Backtracking:** If recursive call fails, unmark `v` as visited and try the next valid neighbor.

7. Common Mistakes and Edge Cases

- **Visited Check:** Forgetting to check if a vertex is already in the path, leading to non-simple paths (repeated vertices). Using a `visited` array/set is efficient.
- **Base Case:** Forgetting the final check: does the last vertex connect back to the first?
- **Off-by-one Errors:** Incorrect indexing in `path` array (`k` vs `k-1`).
- **Backtracking:** Incorrectly managing the `visited` status or `path` array during backtracking.
- **Disconnected Graphs:** The algorithm might explore only the component containing the start node. A true Hamiltonian circuit requires the graph to be connected (and more).
- **Starting Vertex:** The choice of starting vertex doesn't affect the *existence* of a Hamiltonian circuit, but the specific circuit found might differ. The algorithm usually starts at 0 for simplicity.
- **Complexity:** Hamiltonian Circuit is NP-complete. Backtracking has exponential worst-case time complexity (roughly O(n! *n) or O(n* 2^n) depending on graph density and pruning effectiveness).

8. MCQ/Short questions and answers

1. **Q:** What is a Hamiltonian Circuit?
   A: A cycle in a graph that visits every vertex exactly once and returns to the starting vertex.
2. **Q:** What is the key difference between a Hamiltonian Circuit and a Hamiltonian Path?
   A: A Hamiltonian Circuit must return to the starting vertex, while a Hamiltonian Path does not.
3. **Q:** What two conditions must be met to add a vertex `v` at position `k` in the path during the backtracking search?
   A: 1. `v` must be adjacent to the previous vertex `path[k-1]`. 2. `v` must not have been visited earlier in the path (`path[0...k-1]`).
4. **Q:** What check is performed at the base case (`k == n`)?
   A: Check if an edge exists between the last vertex added (`path[n-1]`) and the starting vertex (`path[0]`).
5. **Q:** Is the Hamiltonian Circuit problem computationally easy or hard?
   A: It is computationally hard (NP-complete). Backtracking provides a way to find a solution but can be

very slow for large graphs.

---

3. Subset-Sum Problem

1. Concept Explanation

- **What:** Given a set (or multiset) of non-negative integers `S` and a target sum `T`, the Subset-Sum problem asks if there exists a subset of `S` whose elements sum up exactly to `T`.
- **Why Backtracking:** We can explore subsets by considering each element one by one. For each element, we have two choices: either include it in the current subset or exclude it. We keep track of the sum of the elements included so far. If the current sum equals `T`, we found a solution. If the current sum exceeds `T`, we know this path cannot lead to a solution, so we backtrack. If we run out of elements and the sum is not `T`, we backtrack.
- **How:**
  1. Sort the input set `S` (optional, but can help with pruning).
  2. Start with an empty subset and a current sum of 0. Consider the first element `S[0]`.
  3. **Recursive Step (for element `S[index]`):**
     - **Choice 1: Include `S[index]`:**
       - Check if `current_sum + S[index] <= T`.
       - If yes, recursively call the function for the next element (`index + 1`) with the updated sum (`current_sum + S[index]`).
       - If the recursive call returns true, return true.
     - **Choice 2: Exclude `S[index]`:**
       - Recursively call the function for the next element (`index + 1`) with the same sum (`current_sum`).
       - If the recursive call returns true, return true.
  4. **Base Cases:**
     - If `current_sum == T`: A solution is found. Return true.
     - If `index == n` (no more elements) and `current_sum != T`: No solution found down this path. Return false.
     - If `current_sum > T`: Prune this path. Return false.
  5. If neither including nor excluding the current element leads to a solution, return false.
- **Optimization/Pruning:**
  - If `current_sum + S[index] > T`, we don't need to explore the "Include" branch.
  - If the set `S` is sorted, and `current_sum + remaining_sum < T` (where `remaining_sum` is the sum of all elements from `index` onwards), we can potentially prune, but calculating `remaining_sum` adds overhead. A simpler prune is just `current_sum > T`.

2. Real-World Analogies and Use-Cases

- **Analogy:** You have a collection of coins/bills (the set `S`) and want to know if you can make exact change for a specific amount (`T`). You try combinations of coins: either include a specific coin or don't, ensuring the total doesn't exceed `T`.
- **Use-Cases:**
  - **Cryptography:** Related to the knapsack cryptosystem (though largely broken).
  - **Resource Allocation:** Can a subset of available resources meet a specific requirement total?
  - **Change-Making Problem:** Can exact change be made? (Though often solved with DP if coin counts are unlimited or large).
  - **Finance:** Selecting assets to meet a target investment value.

3. Step-by-Step Solved Example

Problem: Find if there is a subset of S = {3, 1, 5, 2} that sums to T = 6.

Steps (Recursive Calls `solve(index, current_sum)`):
Let S = {1, 2, 3, 5} (Sorted)

1. `solve(0, 0)`: index=0, sum=0. Element S[0]=1.
   - **Include 1:** `solve(1, 1)` (sum=1 <= 6)
     - `solve(1, 1)`: index=1, sum=1. Element S[1]=2.
     - **Include 2:** `solve(2, 3)` (sum=1+2=3 <= 6)
       - `solve(2, 3)`: index=2, sum=3. Element S[2]=3.
       - **Include 3:** `solve(3, 6)` (sum=3+3=6 <= 6)
         - `solve(3, 6)`: index=3, sum=6. Base Case: `current_sum == T`. Return `true`.
       - `solve(2, 3)` returns `true`.
     - `solve(1, 1)` returns `true`.
   - `solve(0, 0)` returns `true`.

Solution Found: Yes. The path taken was Include 1, Include 2, Include 3. Subset {1, 2, 3} sums to 6.

Alternative Path Example:

1. `solve(0, 0)`: index=0, sum=0. Element S[0]=1.
   - **Exclude 1:** `solve(1, 0)`
     - `solve(1, 0)`: index=1, sum=0. Element S[1]=2.
     - **Include 2:** `solve(2, 2)` (sum=0+2=2 <= 6)
       - `solve(2, 2)`: index=2, sum=2. Element S[2]=3.
       - **Include 3:** `solve(3, 5)` (sum=2+3=5 <= 6)
         - `solve(3, 5)`: index=3, sum=5. Element S[3]=5.
         - **Include 5:** `solve(4, 10)` (sum=5+5=10 > 6). Prune. Return `false`.
         - **Exclude 5:** `solve(4, 5)`
           - `solve(4, 5)`: index=4. Base Case: `index == n`. Return `false`.
         - `solve(3, 5)` returns `false`.

- **Exclude 3:** `solve(3, 2)`
  - `solve(3, 2)`: index=3, sum=2. Element S[3]=5.
  - **Include 5:** `solve(4, 7)` (sum=2+5=7 > 6). Prune. Return `false`.
  - **Exclude 5:** `solve(4, 2)`
    - `solve(4, 2)`: index=4. Base Case: `index == n`. Return `false`.
  - `solve(3, 2)` returns `false`.
- `solve(2, 2)` returns `false`.

- **Exclude 2:** `solve(2, 0)`
  - ... and so on.

4. Pseudocode and Code Implementation

Pseudocode:

```
function SubsetSum(S, n, T): // S=set, n=size, T=target sum
  // Optionally sort S first
  return solveSubsetUtil(S, n, T, 0, 0) // index=0, current_sum=0

function solveSubsetUtil(S, n, T, index, current_sum):
  // Base Cases
  if current_sum == T:
    return true // Found a subset
  if index == n: // Reached end of set
    return false // No subset found down this path
  if current_sum > T: // Pruning optimization
      return false

  // Recursive Steps
  // Try including the current element S[index]
  // Check if including S[index] doesn't immediately exceed T
  if current_sum + S[index] <= T:
      if solveSubsetUtil(S, n, T, index + 1, current_sum + S[index]):
          return true

  // Try excluding the current element S[index]
  if solveSubsetUtil(S, n, T, index + 1, current_sum):
    return true

  // If neither including nor excluding leads to a solution
  return false
```

Note: The pruning `current_sum > T` can be checked at the beginning of the function for clarity.
Refined Pseudocode:

```
function solveSubsetUtil(S, n, T, index, current_sum):
  // Base Cases
```

```
  if current_sum == T:
    return true
  if index == n or current_sum > T: // Reached end or exceeded target
    return false

  // Recursive Steps
  // Choice 1: Include S[index]
  if solveSubsetUtil(S, n, T, index + 1, current_sum + S[index]):
    return true

  // Choice 2: Exclude S[index]
  if solveSubsetUtil(S, n, T, index + 1, current_sum):
    return true

  return false
```

Python Implementation:

```python
def subset_sum(S, target_sum):
    """Checks if a subset of S sums to target_sum using backtracking."""
    n = len(S)
    # Optional: Sort for potential minor optimization (doesn't change worst case)
    # S.sort()
    memo = {} # For memoization (DP optimization, not pure backtracking)

    def solve_util(index, current_sum):
        # Check memoization table
        state = (index, current_sum)
        if state in memo:
            return memo[state]

        # Base Cases
        if current_sum == target_sum:
            return True
        if index == n or current_sum > target_sum:
            return False

        # Choice 1: Include S[index]
        # No need for check current_sum + S[index] <= target_sum here,
        # as the base case current_sum > target_sum handles it in the next call.
        include_current = solve_util(index + 1, current_sum + S[index])

        if include_current:
            memo[state] = True
            return True

        # Choice 2: Exclude S[index]
        exclude_current = solve_util(index + 1, current_sum)
```

```
        memo[state] = exclude_current
        return exclude_current

    return solve_util(0, 0)

# Example Usage:
set_s = [3, 1, 5, 2]
target = 6
print(f"Set: {set_s}, Target: {target}")
print(f"Subset exists? {subset_sum(set_s, target)}") # Output: True

target = 12
print(f"Set: {set_s}, Target: {target}")
print(f"Subset exists? {subset_sum(set_s, target)}") # Output: False (3+1+5+2 = 11)

target = 5
print(f"Set: {set_s}, Target: {target}")
print(f"Subset exists? {subset_sum(set_s, target)}") # Output: True ({5} or {3, 2})
```

(Note: The Python code includes memoization, turning it into a top-down DP solution, which is much more efficient than pure backtracking for Subset Sum due to overlapping subproblems. Pure backtracking would omit the `memo` dictionary).

C++ Implementation (Pure Backtracking):

```
#include <vector>
#include <numeric> // Not strictly needed for backtracking logic
#include <iostream>
#include <algorithm> // Optional: for sort

using namespace std;

// Recursive utility function for subset sum (pure backtracking)
bool solveSubsetUtil(const vector<int>& S, int target_sum, int index, int current_sum)
{
    // Base Cases
    if (current_sum == target_sum) {
        return true; // Found a subset
    }
    // Pruning: If current sum exceeds target OR we've run out of elements
    if (index == S.size() || current_sum > target_sum) {
        return false;
    }

    // Choice 1: Include S[index]
    // Check before recursing to avoid unnecessary calls if sum already exceeds target
    if (solveSubsetUtil(S, target_sum, index + 1, current_sum + S[index])) {
        return true;
    }
```

```
    // Choice 2: Exclude S[index]
    if (solveSubsetUtil(S, target_sum, index + 1, current_sum)) {
        return true;
    }

    // If neither choice leads to a solution from this state
    return false;
}


// Main function for subset sum
bool subsetSum(const vector<int>& S, int target_sum) {
    // Optional: sort S
    // vector<int> sorted_S = S;
    // sort(sorted_S.begin(), sorted_S.end());
    // return solveSubsetUtil(sorted_S, target_sum, 0, 0);
    return solveSubsetUtil(S, target_sum, 0, 0);
}


int main() {
    vector<int> set_s = {3, 1, 5, 2};

    int target1 = 6;
    cout << "Set: {3, 1, 5, 2}, Target: " << target1 << endl;
    cout << "Subset exists? " << (subsetSum(set_s, target1) ? "True" : "False") <<
endl; // Output: True

    int target2 = 12;
    cout << "Set: {3, 1, 5, 2}, Target: " << target2 << endl;
    cout << "Subset exists? " << (subsetSum(set_s, target2) ? "True" : "False") <<
endl; // Output: False

    int target3 = 5;
    cout << "Set: {3, 1, 5, 2}, Target: " << target3 << endl;
    cout << "Subset exists? " << (subsetSum(set_s, target3) ? "True" : "False") <<
endl; // Output: True

    return 0;
}
```

5. Diagrams or Flowcharts

State-Space Tree (Example S={1, 2, 3}, T=3):

```
                    (idx=0, sum=0)
                   /              \
```

```
Element 1:  Inc 1 (idx=1, sum=1)   Exc 1 (idx=1, sum=0)
           /          \           /          \
Element 2: Inc 2(2,3) Exc 2(2,1) Inc 2(2,2) Exc 2(2,0)
           |         /     \      /    \      /    \
Element 3: SOL!   Inc 3(3,4)X Exc 3(3,1) ...   ... Inc 3(3,3) Exc 3(3,0)
                   (Prune)      |                 |       |
idx=3(Base):                  FALSE            SOL!     FALSE
```

The tree shows the two choices (Include/Exclude) at each level (element index). Nodes store `(index, current_sum)`. Paths are pruned if `sum > T`. Solutions are found if `sum == T`.

6. Key Formulas and Logic Summary

- **Goal:** Determine if a subset of `S` sums to `T`.
- **Core Logic:** For each element `S[index]`, recursively explore two branches: one including `S[index]` (updating sum) and one excluding `S[index]` (keeping sum the same).
- **State:** Typically tracked by `(index, current_sum)`.
- **Recursion:** `solve(index, current_sum)` calls `solve(index + 1, current_sum + S[index])` and `solve(index + 1, current_sum)`.
- **Base Cases:**
  - `current_sum == T` -> Found (return true).
  - `index == n` -> Not found down this path (return false).
- **Pruning:** `current_sum > T` -> Cannot lead to solution (return false).

7. Common Mistakes and Edge Cases

- **Base Cases:** Missing or incorrect base cases (e.g., not checking `current_sum == T` first, handling `index == n` incorrectly).
- **Pruning:** Forgetting the `current_sum > T` check, leading to unnecessary exploration.
- **State Variables:** Passing incorrect parameters (index, sum) in recursive calls.
- **Overlapping Subproblems:** Pure backtracking can be very inefficient for Subset Sum because the same subproblem (`solve(index, current_sum)`) can be reached via many different paths (different combinations of including/excluding earlier elements). This is why DP/memoization is usually preferred.
- **Empty Set:** If `S` is empty, a subset exists only if `T` is 0.
- **Target Sum 0:** An empty subset always sums to 0. The algorithm should return true if T=0.
- **Negative Numbers:** The standard problem assumes non-negative integers. If negative numbers are allowed, the `current_sum > T` pruning might be invalid, and the problem becomes more complex.

8. MCQ/Short Questions and Answers

1. **Q:** What does the Subset Sum problem ask?
   A: Whether a subset of a given set of numbers sums up to a specific target value.

2. **Q:** What are the two choices made for each element in the backtracking approach?
   A: Either include the element in the subset or exclude it.
3. **Q:** What are the base cases for the recursive backtracking solution?
   A: `current_sum == target_sum` (success), `index == n` (failure - end of set), `current_sum > target_sum` (pruning).
4. **Q:** Why is pure backtracking often inefficient for Subset Sum?
   A: It recalculates solutions for the same subproblems (same index and current sum) multiple times (Overlapping Subproblems).
5. **Q:** What technique can significantly optimize the backtracking solution for Subset Sum?
   A: Dynamic Programming (either tabulation or memoization) to store and reuse results of subproblems.
