Q85: Flood Fill

1. Problem Understanding

- We are given a 2D matrix wall[m][n], where each element represents a color.
- Starting from a given cell (x, y), we must fill that cell and all 4-directionally connected cells that have the same original color with a new color c.

2. Constraints

- 1 ≤ m, n ≤ 50
- 0 ≤ w[i][j] ≤ 216
- 0 ≤ x < m
- 0 ≤ y < n

3. Edge Cases

- The new color c is the same as the old color → no change needed.
- Starting cell (x, y) is on an edge or corner.
- Entire wall has the same color.
- Single-cell wall.

4. Examples

```
Input:

3 3
0 1 1
0 1 1
1 0 1
1 1 2


Output:

0 2 2
0 2 2
1 0 2


Explanation:
Start from (1,1) where color = 1.
All connected 1s are changed to 2.
```

5. Approaches

Approach 1: DFS (Recursion)

Idea:

- Use Depth-First Search recursively:
- Replace the color at (x, y) with newColor.
- Then, move up, down, left, and right recursively to all connected cells having the same old color.

Steps:

- Get the original color oldColor = wall[x][y].
- If oldColor == c, return (nothing to fill).
- Define a recursive function dfs to fill the same color neighbors.
- For each cell, if it's within bounds and matches oldColor, fill it and recursively call for its 4 neighbors.

Java Code:

```
void floodFill(int[][] wall, int x, int y, int c) {
    int oldColor = wall[x][y];
    if (oldColor == c) return; // No need to fill if already the same
    dfs(wall, x, y, oldColor, c);
}

void dfs(int[][] wall, int i, int j, int oldColor, int newColor) {
    // Boundary or color mismatch check
    if (i < 0 || j < 0 || i >= wall.length || j >= wall[0].length) return;
    if (wall[i][j] != oldColor) return;

    // Fill color
    wall[i][j] = newColor;

    // Recursive calls in 4 directions
    dfs(wall, i - 1, j, oldColor, newColor); // up
    dfs(wall, i + 1, j, oldColor, newColor); // down
    dfs(wall, i, j - 1, oldColor, newColor); // left
    dfs(wall, i, j + 1, oldColor, newColor); // right
}









Recursion Tree (for Example Input)

Input:
```

```
wall =
0 1 1
0 1 1
1 0 1
x=1, y=1, c=2


oldColor = 1

Execution Flow:
dfs(1,1)
│
├── wall[1][1] = 2
│
├── dfs(0,1) → wall[0][1] = 2
│    ├── dfs(-1,1) → out of bounds
│    ├── dfs(1,1) → already 2
│    ├── dfs(0,0) → 0 ≠ 1
│    └── dfs(0,2) → wall[0][2] = 2
│         ├── dfs(-1,2) → out of bounds
│         ├── dfs(1,2) → wall[1][2] = 2
│         │    ├── dfs(0,2) → already 2
│         │    ├── dfs(2,2) → wall[2][2] = 2
│         │    └── others invalid
│         └── dfs(0,3) → out of bounds
│
├── dfs(2,1) → 0 ≠ 1
├── dfs(1,0) → 0 ≠ 1
└── dfs(1,2) → already filled


🧩 Final Matrix:

0 2 2
0 2 2
1 0 2
```

Complexity (Time & Space):

- Time: O(m × n) (each cell visited once)
- Space: O(m × n) (recursion stack)

Approach 2: BFS (Queue-Based)

Idea:

- Perform level-wise traversal using a queue (iterative version of DFS).
- Each time we process a cell, we color it and add all its valid neighbors of the same color to the queue.

Java Code:

```
void floodFillBFS(int[][] wall, int x, int y, int c) {
    int oldColor = wall[x][y];
    if (oldColor == c) return;

    int[][] dirs = {{1,0}, {-1,0}, {0,1}, {0,-1}};
    Queue<int[]> q = new LinkedList<>();
    q.add(new int[]{x, y});

    while (!q.isEmpty()) {
        int[] curr = q.poll();
        int i = curr[0], j = curr[1];
        wall[i][j] = c;

        for (int[] d : dirs) {
            int ni = i + d[0], nj = j + d[1];
            if (ni >= 0 && nj >= 0 && ni < wall.length && nj < wall[0].length &&
wall[ni][nj] == oldColor) {
                q.add(new int[]{ni, nj});
            }
        }
    }
}

Visualization (Level Order)
Start: (1,1)
Queue: [(1,1)]
→ Fill (1,1), add (0,1), (1,2)
Queue: [(0,1), (1,2)]
→ Fill (0,1), add (0,2)
Queue: [(1,2), (0,2)]
→ Fill (1,2), add (2,2)
Queue: [(0,2), (2,2)]
→ Fill (0,2), then (2,2)
Queue empty ☑


🧩 Final Matrix:

0 2 2
0 2 2
1 0 2
```

Complexity (Time & Space):

- Time: O(m × n)
- Space: O(m × n) (queue)

Approach 3: Iterative DFS (Stack)

Idea:

- Same as DFS but uses an explicit stack instead of recursion (to avoid stack overflow).

Java Code:

```java
void floodFillIterativeDFS(int[][] wall, int x, int y, int c) {
    int oldColor = wall[x][y];
    if (oldColor == c) return;

    Stack<int[]> st = new Stack<>();
    st.push(new int[]{x, y});
    int[][] dirs = {{1,0}, {-1,0}, {0,1}, {0,-1}};

    while (!st.isEmpty()) {
        int[] cell = st.pop();
        int i = cell[0], j = cell[1];
        wall[i][j] = c;

        for (int[] d : dirs) {
            int ni = i + d[0], nj = j + d[1];
            if (ni >= 0 && nj >= 0 && ni < wall.length && nj < wall[0].length &&
wall[ni][nj] == oldColor) {
                st.push(new int[]{ni, nj});
            }
        }
    }
}
```

Complexity (Time & Space):

- Time: O(m × n)
- Space: O(m × n)

---

6. Justification / Proof of Optimality

- All methods correctly fill the connected region:
- DFS (Recursive) → simplest and intuitive
- BFS (Queue) → avoids deep recursion
- Iterative DFS (Stack) → same as recursion but stack-based

---

7. Variants / Follow-Ups

- 8-direction flood fill: Include diagonal cells too.
- Boundary fill algorithm: Used in graphics to fill enclosed areas.
- Multi-color fill: Handle multiple start points.

---

8. Tips & Observations

- Always check if oldColor == newColor before recursing.
- DFS is easier to write, but BFS/iterative DFS are safer for large inputs.

- Flood Fill is conceptually the same as finding a connected component in a 2D grid.

Q86: N Queens

---

1. Problem Understanding

- We need to place n queens on an n × n chessboard such that no two queens attack each other, meaning:
- No two queens share the same row.
- No two queens share the same column.
- No two queens share the same diagonal.
- We must count the number of possible valid configurations (not print them).

---

2. Constraints

- 1 ≤ n ≤ 10
- Output: number of valid configurations

---

3. Edge Cases

- n = 1 → only one configuration.
- n = 2 or n = 3 → no valid solutions.
- Larger n (like 4 or 5) → multiple valid configurations.

---

4. Examples

```
Example 2
Input:
4
Output:
2
Explanation:
There are exactly two valid ways to place 4 queens safely.
```

---

5. Approaches

Approach 1: Backtracking (Recursive with Board Matrix)

Idea:

- We try to place queens row by row.
- At each row, we explore all columns and check if placing a queen is safe.

- If safe → place it and recursively place queens in the next row.
- If not → backtrack and try another column.

Steps:

- Use an n×n matrix to represent the board.
- Start from row 0.
- For each column:
    - Check if it's safe to place a queen.
    - If safe, place it and move to the next row recursively.
    - Backtrack by removing the queen.
- When row == n, increment the count (found a valid configuration).

Java Code:

```
public int totalNQueens(int n) {
    int[][] board = new int[n][n];
    return solve(board, 0, n);
}

private int solve(int[][] board, int row, int n) {
    if (row == n) return 1; // valid configuration found
    int count = 0;
    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col, n)) {
            board[row][col] = 1;        // place queen
            count += solve(board, row + 1, n); // move to next row
            board[row][col] = 0;        // backtrack
        }
    }
    return count;
}

private boolean isSafe(int[][] board, int row, int col, int n) {
    // check column
    for (int i = 0; i < row; i++)
        if (board[i][col] == 1) return false;

    // check left diagonal
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1) return false;

    // check right diagonal
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++)
        if (board[i][j] == 1) return false;

    return true;
}


Recursion Tree (for n = 4)
```

```
We try to place queens row by row:

Row 0 → Try col 0
│
├── Not safe down the line → backtrack
│
Row 0 → Try col 1
│
├── Place Q at (0,1)
│   ├── Row 1 → Try col 3 (safe)
│   │   ├── Row 2 → Try col 0 (safe)
│   │   │   ├── Row 3 → Try col 2 (safe) ☑ Solution 1
│   │   │   └── Backtrack
│   │   └── Backtrack
│   └── Backtrack
│
Row 0 → Try col 2
│
├── Place Q at (0,2)
│   ├── Row 1 → Try col 0 (safe)
│   │   ├── Row 2 → Try col 3 (safe)
│   │   │   ├── Row 3 → Try col 1 (safe) ☑ Solution 2
│   │   │   └── Backtrack
│   │   └── Backtrack
│   └── Backtrack
│
Row 0 → Try col 3 → no valid solution


☑ Total Solutions = 2
```

Complexity (Time & Space):

- Time: O(N!) — exploring all column placements per row.
- Space: O(N^2) — recursion + board storage.

Approach 2: Optimized Backtracking (Using Arrays)

Idea:

- Instead of checking the board each time, keep arrays to mark attacks on columns and diagonals:
  - cols[col] → if a queen is in that column.
  - diag1[row + col] → left diagonal.
  - diag2[row - col + n - 1] → right diagonal.
- This avoids scanning the whole board on each check.

Java Code:

```java
public int totalNQueens(int n) {
    boolean[] cols = new boolean[n];
    boolean[] diag1 = new boolean[2 * n];
```

```java
        boolean[] diag2 = new boolean[2 * n];
        return backtrack(0, n, cols, diag1, diag2);
    }

    private int backtrack(int row, int n, boolean[] cols, boolean[] diag1, boolean[]
    diag2) {
        if (row == n) return 1;
        int count = 0;
        for (int col = 0; col < n; col++) {
            int d1 = row + col;
            int d2 = row - col + n - 1;
            if (cols[col] || diag1[d1] || diag2[d2]) continue;

            cols[col] = diag1[d1] = diag2[d2] = true;
            count += backtrack(row + 1, n, cols, diag1, diag2);
            cols[col] = diag1[d1] = diag2[d2] = false; // backtrack
        }
        return count;
    }
```

Complexity (Time & Space):

- Time: O(N!) (same order but faster in practice)
- Space: O(N)

Approach 3: Bitmask Optimization (Most Efficient for N ≤ 15)

Idea:

- Use bitmasks instead of boolean arrays to store attacked columns and diagonals.
- Each bit position represents whether a column/diagonal is occupied.

Java Code:

```java
    int solve(int n) {
        return place(0, 0, 0, 0, n);
    }

    int place(int row, int cols, int d1, int d2, int n) {
        if (row == n) return 1;
        int count = 0;
        int available = ((1 << n) - 1) & (~(cols | d1 | d2));
        while (available != 0) {
            int pos = available & (-available); // pick rightmost available bit
            available -= pos;
            count += place(row + 1,
                           cols | pos,
                           (d1 | pos) << 1,
                           (d2 | pos) >> 1,
                           n);
        }
        return count;
    }
```

Complexity (Time & Space):

- Time: O(N!) but extremely fast in practice due to bit-level pruning.
- Space: O(N)

---

6. Justification / Proof of Optimality

- Backtracking guarantees exploration of all valid board configurations, and pruning ensures we only go deeper on safe placements.
- Bitmask and array optimizations make the checking O(1).

---

7. Variants / Follow-Ups

- Print all N-Queens configurations instead of counting them.
- Return board states as List<List>.
- M-Queens variant → different attack patterns.

---

8. Tips & Observations

- n = 2 or 3 → always 0 solutions.
- Use arrays or bitmasks to speed up safe checks.
- Backtracking is ideal because of branch pruning.
- Bitmask method is best for competitive programming.

Q87: Knight’s Tour 2

1. Problem Understanding

- We are given a chessboard of size n x n and a knight starting at position (row, col).
- The goal is to move the knight so that it visits every square exactly once, following the standard L-shaped knight moves (2 in one direction + 1 in perpendicular direction).
- We must print all possible configurations where the knight covers every cell exactly once.

---

2. Constraints

- $1 \le n \le 5$
- $0 \le row, col < n$

---

3. Edge Cases

- n = 1 → only one configuration [1].
- Knight cannot complete tour for small boards like n < 5 except special cases.
- Ensure boundary and visited checks to avoid infinite recursion.

---

4. Examples

```
Input:

5
2 0


Output:
(prints all valid knight tours covering all 25 cells once)
```

---

5. Approaches

Approach 1: Brute Force (Pure Backtracking)

Idea:

- Try all 8 possible moves from each cell recursively until all cells are visited.
- If stuck, backtrack and try another path.

Steps:

- Create a board[n][n] initialized with -1.
- Place the knight at the start cell (x, y) = (0, 0) → step = 0.
- Recursively move to the next valid unvisited cell.
- If all cells are filled (step == n*n), print or count the tour.
- Backtrack if no move is possible.

Java Code:

```java
public class KnightTour {
    static int[] dx = {2, 1, -1, -2, -2, -1, 1, 2};
    static int[] dy = {1, 2, 2, 1, -1, -2, -2, -1};

    public static void knightTour(int n) {
        int[][] board = new int[n][n];
        for (int[] row : board) Arrays.fill(row, -1);

        board[0][0] = 0; // start from (0,0)
        if (solve(board, 0, 0, 1, n))
            printBoard(board, n);
        else
            System.out.println("No tour exists.");
    }

    private static boolean solve(int[][] board, int x, int y, int step, int n) {
        if (step == n * n) return true;

        for (int i = 0; i < 8; i++) {
            int nx = x + dx[i], ny = y + dy[i];
            if (isValid(board, nx, ny, n)) {
                board[nx][ny] = step;
                if (solve(board, nx, ny, step + 1, n))
                    return true;
                board[nx][ny] = -1; // backtrack
            }
        }
        return false;
    }

    private static boolean isValid(int[][] board, int x, int y, int n) {
        return (x >= 0 && y >= 0 && x < n && y < n && board[x][y] == -1);
    }

    private static void printBoard(int[][] board, int n) {
        for (int[] row : board) {
            for (int cell : row)
                System.out.printf("%2d ", cell);
            System.out.println();
        }
    }
}

Recursion Tree (Example for 4×4 starting at (0, 0))
(0,0)
├── (2,1)
│   ├── (3,3)
│   │   ├── ...
│   │   └── Backtrack
│   ├── (1,3)
│   └── ...
├── (1,2)
│   ├── (3,3)
│   └── ...
└── ...


Each branch represents one possible knight move.
If a move leads to a dead end (no unvisited cell), recursion backtracks.


for accio just integrated in the boiler plate


import java.io.*;
import java.util.*;
```

```java
public class Main {

    public static void main(String[] args) throws Exception     {
        Scanner scn = new Scanner(System.in);
        int n = scn.nextInt();
        int r = scn.nextInt();
        int c = scn.nextInt();
        int[][] chess = new int[n][n];
        printKnightsTour(chess, r, c, 1);
    }

  public static void printKnightsTour(int[][] chess, int r, int c, int move) {
    int n = chess.length;

    // Base case: all cells visited
    if (move == n * n) {
        chess[r][c] = move;
        displayBoard(chess);
        chess[r][c] = 0;  // backtrack
        return;
    }

    // Mark current cell with move number
    chess[r][c] = move;

    // Knight's 8 possible moves (clockwise starting from top-right)
    int[] dr = {-2, -1, 1, 2, 2, 1, -1, -2};
    int[] dc = {1, 2, 2, 1, -1, -2, -2, -1};

    for (int i = 0; i < 8; i++) {
        int nr = r + dr[i]; //next row
        int nc = c + dc[i]; //next column

        if (nr >= 0 && nc >= 0 && nr < n && nc < n && chess[nr][nc] == 0) {
            printKnightsTour(chess, nr, nc, move + 1);
        }
    }

    // Backtrack
    chess[r][c] = 0;
}

    public static void displayBoard(int[][] chess){
        for(int i = 0; i < chess.length; i++){
            for(int j = 0; j < chess[0].length; j++){
                System.out.print(chess[i][j] + " ");
            }
            System.out.println();
        }

        System.out.println();
    }
}
```

Complexity (Time & Space):

- ⏱ Time Complexity
    - Worst-case: O(8^(n^2))
        - Each cell can have up to 8 recursive calls.
        - The recursion depth = number of cells = n^2.
    - Very slow for large n (exponential).
- 💾 Space Complexity
    - O(n^2) → due to board matrix and recursion stack.

Approach 2: Warnsdorff's Heuristic (Optimized, not required here)

Idea:

- Instead of trying all 8 moves randomly, choose the move that has the least number of onward valid moves (i.e., moves leading to fewer further possibilities).
- This reduces backtracking dramatically.
- To reduce backtracking, always move the knight to the square with the fewest onward moves. This ensures higher chance of successful tours early.

Steps:

- From the current cell, check all valid knight moves.
- For each move, count how many onward valid moves exist.
- Always choose the cell with the minimum onward count first.
- Continue until board is full or stuck.

Java Code:

```java
public class KnightTourHeuristic {
    static int[] dx = {2, 1, -1, -2, -2, -1, 1, 2};
    static int[] dy = {1, 2, 2, 1, -1, -2, -2, -1};

    public static void knightTourWarnsdorff(int n) {
        int[][] board = new int[n][n];
        for (int[] row : board) Arrays.fill(row, -1);
        int x = 0, y = 0;
        board[x][y] = 0;

        for (int step = 1; step < n * n; step++) {
            int nextMove = -1, minDegree = 9, nx = -1, ny = -1;

            for (int i = 0; i < 8; i++) {
                int xx = x + dx[i], yy = y + dy[i];
                if (isValid(board, xx, yy, n)) {
                    int degree = countOnwardMoves(board, xx, yy, n);
                    if (degree < minDegree) {
                        minDegree = degree;
                        nx = xx; ny = yy;
                        nextMove = i;
                    }
                }
            }

            if (nextMove == -1) break; // stuck
            x = nx; y = ny;
            board[x][y] = step;
        }

        printBoard(board, n);
    }

    private static int countOnwardMoves(int[][] board, int x, int y, int n) {
        int count = 0;
        for (int i = 0; i < 8; i++) {
            int xx = x + dx[i], yy = y + dy[i];
            if (isValid(board, xx, yy, n)) count++;
        }
        return count;
    }

    private static boolean isValid(int[][] board, int x, int y, int n) {
        return (x >= 0 && y >= 0 && x < n && y < n && board[x][y] == -1);
    }

    private static void printBoard(int[][] board, int n) {
        for (int[] row : board) {
            for (int cell : row)
                System.out.printf("%2d ", cell);
            System.out.println();
        }
    }
}


🌿 Visualization (Warnsdorff for 5×5)
scss
Copy code
(0,0)
↓ (2,1)
↓ (4,2)
↓ (3,4)
↓ (1,3)
↓ (0,1)
↓ (2,0)
↓ ...
→ covers all 25 squares sequentially
The knight never backtracks, because the heuristic prevents getting trapped early.
```

Complexity (Time & Space):

- ⏱ Time Complexity

    - Average case: $O(n^2)$ (linear traversal using heuristic)
    - Worst case: $O(n^2 \times 8) \approx O(n^2)$ (still practical)
        - Each move checks 8 directions and counts onward moves.
- 💾 Space Complexity
    - $O(n^2)$ for the board matrix.

---

6. Justification / Proof of Optimality

- Backtracking: Works for small boards but exponential.
- Warnsdorff's heuristic: Extremely efficient — works for almost all $n \geq 5$ directly.

---

7. Variants / Follow-Ups

- Closed Knight's Tour: Last move connects to the first cell (forms a loop).
- Count All Tours: Count total unique paths (needs backtracking).
- Open Knight's Tour: Regular version (does not loop back).

---

8. Tips & Observations

- Use Warnsdorff's rule whenever only one path is needed.
- Use backtracking when you must count or print all tours.
- Pruning + move ordering drastically improves performance.
- Knight's Tour → excellent example of heuristic-guided recursion.

Q88: Combination Sum

1. Problem Understanding

- We are given a set of distinct positive integers nums and a target integer target.
- We need to find all unique combinations of numbers from nums that sum to target.
    - ➡ You can use each number unlimited times.
    - ➡ The order of numbers in a combination doesn't matter.
    - ➡ The output can be in any order.

---

2. Constraints

- $1 \leq$ nums.length ≤ 30
- $2 \leq$ nums[i] ≤ 40
- $1 \leq$ target ≤ 40
- All nums[i] are distinct

---

3. Edge Cases

- If nums is empty → return empty list
- If no combination sums to target → return empty list
- If target < min(nums) → no possible combinations

---

4. Examples

```
Input:

nums = [6, 2, 7, 5], target = 16


Output:

[2,2,2,2,2,2,2,2]
[2,2,2,2,2,6]
[2,2,2,5,5]
[2,2,5,7]
[2,2,6,6]
[2,7,7]
[5,5,6]
```

---

5. Approaches

Approach 1: Recursive Backtracking (DFS)

Idea:

- Use recursion to try each number multiple times until the running sum exceeds the target.
- If the sum equals target, record the current path as a valid combination.

Steps:

- Sort nums (optional, helps with pruning).
- Start recursion from index 0 with an empty combination.
- At each step:
  - If sum == target → store result.
  - If sum > target → return.
  - Else → explore:
    - include current element again (i same)
    - exclude and move to next (i+1)

Java Code:

```java
import java.util.*;

public class Solution {
    public static List<List<Integer>> combinationSum(int[] nums, int target) {
        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(nums); // Optional: helps with pruning
        backtrack(nums, target, 0, new ArrayList<>(), ans);
        return ans;
    }

    private static void backtrack(int[] nums, int target, int idx, List<Integer>
curr, List<List<Integer>> ans) {
        if (target == 0) {
            ans.add(new ArrayList<>(curr));
            return;
        }
        if (target < 0 || idx == nums.length) return;

        // Include current element
        curr.add(nums[idx]);
        backtrack(nums, target - nums[idx], idx, curr, ans);
        curr.remove(curr.size() - 1);

        // Exclude current element
        backtrack(nums, target, idx + 1, curr, ans);
    }
}


Recursion Tree Example

Example Input: nums = [2, 3, 5], target = 7

backtrack([], target=7, idx=0)
│
├── include 2 → [2], target=5
│    ├── include 2 → [2,2], target=3
│    │    ├── include 2 → [2,2,2], target=1 ✖
│    │    └── exclude 2 → idx=1
│    │         ├── include 3 → [2,2,3], target=0 ☑
│    │         └── exclude 3 → idx=2
│    │              ├── include 5 → [2,2,5], target=-2 ✖
│    │              └── exclude 5 ✖
│    └── exclude 2 → idx=1
│         ├── include 3 → [2,3], target=2
│         │    ├── include 3 → [2,3,3], target=-1 ✖
│         │    └── exclude 3 → idx=2
│         │         ├── include 5 → [2,3,5], target=-3 ✖
│         │         └── exclude 5 ✖
│         └── exclude 3 → idx=2
│              ├── include 5 → [2,5], target=0 ☑
│              └── exclude 5 ✖
│
└── exclude 2 → idx=1
     ├── include 3 → [3], target=4
     │    ├── include 3 → [3,3], target=1 ✖
     │    └── exclude 3 → idx=2
     │         ├── include 5 → [3,5], target=-1 ✖
```

```
        │          └── exclude 5 ❌
        └── exclude 3 → idx=2
                 ├── include 5 → [5], target=2 ❌
                 └── exclude 5 ❌


☑ Valid combinations:

[2,2,3]
[2,5]
```

Complexity (Time & Space):

- ⏱ Time Complexity
  - Worst case: O(2^(target/min(nums))) — each element can be included/excluded many times
  - On average: Exponential (backtracking)
- 💾 Space Complexity
  - O(target/min(nums)) recursion depth (stack + combination list storage)

Approach 2: Iterative (Using DP - Tabulation)

Idea:

- Use dynamic programming to build up combinations for all targets ≤ target.

Steps:

- Create a dp list where dp[i] stores all combinations summing to i.
- For each num in nums, update combinations from num → target.
- For each i, append num to every combination in dp[i - num].

Java Code:

```java
import java.util.*;

public class SolutionDP {
    public static List<List<Integer>> combinationSum(int[] nums, int target) {
        Arrays.sort(nums);
        List<List<List<Integer>>> dp = new ArrayList<>(target + 1);
        for (int i = 0; i <= target; i++) dp.add(new ArrayList<>());

        dp.get(0).add(new ArrayList<>()); // base case

        for (int num : nums) {
            for (int t = num; t <= target; t++) {
                for (List<Integer> prev : dp.get(t - num)) {
                    List<Integer> newComb = new ArrayList<>(prev);
                    newComb.add(num);
                    dp.get(t).add(newComb);
                }
            }
        }
        return dp.get(target);
    }
}
```

Complexity (Time & Space):

- ⏱ Time Complexity
  - O(target × n × avg_combinations) (depends on combination count)
  - Usually faster than recursion for small targets
- 💾 Space Complexity
  - O(target × avg_combinations)

---

6. Justification / Proof of Optimality

- Backtracking approach explores all possible sums systematically.
- DP approach can optimize recomputation but may use more memory.

---

7. Variants / Follow-Ups

- Combination Sum II → Each number used once only.
- Combination Sum III → Only use numbers 1–9, select k numbers.

---

8. Tips & Observations

- Sort nums → helps prune unnecessary recursive calls.
- Backtrack efficiently → stop recursion as soon as target < 0.
- Store results in a set (if duplicates possible).

Q89: Combination Sum II

---

1. Problem Understanding

- We are given an array C (possibly containing duplicates) and a target sum T.
- We need to find all unique combinations of numbers in C that sum to T.
- Each number in the array can be used at most once, and we must avoid duplicate combinations.

---

2. Constraints

- $1 \le n \le 100$
- $1 \le C[i] \le 50$
- $1 \le target \le 30$
- Array may contain duplicates

- Elements in each combination must be sorted in non-decreasing order

---

3. Edge Cases

- No combination sums to target → print nothing.
- All elements > target → no result.
- Array contains duplicates → skip repeated combinations.

---

4. Examples

```
Input:

7
10 1 2 7 6 1 5
8


Output:

1 1 6
1 2 5
1 7
2 6
```

---

5. Approaches

Approach 1: Backtracking with Duplicate Handling (Optimized DFS) ✅

Idea:

- We use recursion to generate combinations but:
  - Sort the array first to handle duplicates easily.
  - Skip duplicate elements at the same recursion level.
  - Allow each element to be used only once (move to next index after using one).

Steps:

- Sort the input array.
- Use a recursive function dfs(index, target, currentList).
- If target == 0 → add the current list to result.
- For each element i from index to end:
  - If i > index and C[i] == C[i-1] → skip duplicate.
  - If C[i] > target → break (no need to continue).
  - Include C[i] and recurse for i+1 (since reuse not allowed).

Java Code:

```
import java.util.*;

public class Solution {
    public static List<List<Integer>> combinationSum2(int[] candidates, int
target) {
        Arrays.sort(candidates); // sort to handle duplicates
        List<List<Integer>> ans = new ArrayList<>();
        backtrack(candidates, target, 0, new ArrayList<>(), ans);
        return ans;
    }

    private static void backtrack(int[] arr, int target, int start, List<Integer>
curr, List<List<Integer>> ans) {
        if (target == 0) {
            ans.add(new ArrayList<>(curr));
            return;
        }

        for (int i = start; i < arr.length; i++) {
            if (i > start && arr[i] == arr[i - 1]) continue; // skip duplicates
            if (arr[i] > target) break;

            curr.add(arr[i]);
            backtrack(arr, target - arr[i], i + 1, curr, ans); // move to next
index
            curr.remove(curr.size() - 1);
        }
    }
}

🌳 Recursion Tree Example

Example Input:
C = [1, 1, 2, 5, 6, 7, 10], target = 8

dfs([], 8, 0)
│
├── include 1 → [1], target = 7
│   ├── include 1 → [1,1], target = 6
│   │   ├── include 2 → [1,1,2], target = 4
│   │   │   ├── include 5 → [1,1,2,5], target = -1 ✖
│   │   │   └── ...
│   │   ├── include 5 → [1,1,5], target = 0 ☑
│   │   └── include 6 → [1,1,6], target = 0 ☑
│   └── include 2 → [1,2], target = 5
│       ├── include 5 → [1,2,5], target = 0 ☑
│       └── include 6 → [1,2,6], target = -1 ✖
│
├── include 2 → [2], target = 6
│   ├── include 5 → [2,5], target = 1 ✖
│   ├── include 6 → [2,6], target = 0 ☑
│   └── ...
│
```

```
├── include 5 → [5], target = 3 ❌
└── include 6 → [6], target = 2 ❌


☑ Valid combinations:

[1,1,6]
[1,2,5]
[1,7]
[2,6]
```

Complexity (Time & Space):

- ⏱ Time Complexity
  - O(2^n) in the worst case (every element is either taken or not).
  - Pruning and duplicate skipping reduce actual time.
- 💾 Space Complexity
  - O(n) recursion stack depth (since each number used once per path).

Approach 2: Iterative (Using Set to Avoid Duplicates)

Idea:

- Use a queue or list to simulate recursion iteratively, keeping a Set<List> to prevent duplicates.
- Drawback:
  - More memory heavy.
  - Less intuitive than recursion.

Java Code:

```java
import java.util.*;

public class SolutionIterative {
    public static List<List<Integer>> combinationSum2(int[] arr, int target) {
        Arrays.sort(arr);
        Set<List<Integer>> set = new HashSet<>();
        Queue<int[]> queue = new LinkedList<>(); // state: index, current sum,
mask bit
        queue.offer(new int[]{0, 0, 0});

        while (!queue.isEmpty()) {
            int[] state = queue.poll();
            int index = state[0], sum = state[1];
            int mask = state[2];

            if (sum == target) {
                List<Integer> comb = new ArrayList<>();
                for (int i = 0; i < arr.length; i++)
                    if ((mask & (1 << i)) != 0) comb.add(arr[i]);
                set.add(comb);
                continue;
            }

            if (sum > target || index == arr.length) continue;

            // include current
            queue.offer(new int[]{index + 1, sum + arr[index], mask | (1 <<
index)});
            // exclude current
            queue.offer(new int[]{index + 1, sum, mask});
        }

        return new ArrayList<>(set);
    }
}
```

Complexity (Time & Space):

- ⏱ Time Complexity
  - O(2^n) states
  - Extra cost for duplicate checks in HashSet
- 💾 Space Complexity
  - O(2^n) due to state storage

---

6. Justification / Proof of Optimality

- Backtracking (DFS) is most efficient and readable.
- Iterative approach works but is not preferred due to overhead.

---

7. Variants / Follow-Ups

- Combination Sum I → Numbers can be reused multiple times.
- Combination Sum III → Choose exactly k numbers from 1–9.
- Subset Sum II → Only check if target is achievable (True/False).

---

8. Tips & Observations

- Sorting is mandatory to easily skip duplicates.
- Always skip same element at same recursion depth using:
  - if (i > start && arr[i] == arr[i-1]) continue;
- For printing in sorted order, sort both the array and the result list.

Q90: Coins Permutations - 1

---

1. Problem Understanding

- You are given a list of coin denominations and a total amount amt.
- You need to print all permutations (different orders) of coins that sum up to amt.
- Unlike combinations, here order matters — e.g., [2,3,7] and [3,2,7] are considered different permutations.

---

2. Constraints

- 1 <= n <= 30
- 1 <= coin[i] <= 20
- 0 <= amt <= 50

---

3. Edge Cases

- If amt == 0, print the permutation and return.
- If amt < 0, stop exploring that path.
- If no valid permutation exists, no output.
- Duplicates in coins array should not produce duplicate permutations.

---

4. Examples

```
Input:

5
2 3 5 6 7
12


Output:

2 3 7
2 7 3
3 2 7
3 7 2
5 7
7 2 3
7 3 2
7 5


Explanation:
These are all distinct permutations of coins that sum to 12.
```

---

5. Approaches

Approach 1: Recursive Backtracking (Pick Each Coin Every Time)

Idea:

- Explore each coin as the first choice.
- Subtract its value from the amount.
- Recurse to find the remaining amount.
- When amt == 0, print the current permutation.

Steps:

- Start with an empty permutation string and given amt.
- For each coin:
  - If coin ≤ amt → choose it and recurse with reduced amount.
  - If amt == 0 → print the permutation.
- Backtrack to explore other choices.

Java Code:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scn = new Scanner(System.in);
        int n = scn.nextInt();
        int[] coins = new int[n];
        for (int i = 0; i < n; i++) coins[i] = scn.nextInt();
        int amt = scn.nextInt();
        coinPermutations(coins, amt, "");
    }

    public static void coinPermutations(int[] coins, int amt, String asf) {
        if (amt == 0) {
            System.out.println(asf.trim());
            return;
        }
        if (amt < 0) return;

        for (int coin : coins) {
            if (coin <= amt)
                coinPermutations(coins, amt - coin, asf + coin + " ");
        }
    }
}

🕸 Recursion Tree (Example)
For coins = [2, 3, 7] and amt = 5:

coinPermutations([], 5)
│
├── choose 2 → coinPermutations([2], 3)
│      ├── choose 2 → coinPermutations([2,2], 1)
│      └── choose 3 → coinPermutations([2,3], 0) ☑ print "2 3"
│
```

```
├── choose 3 → coinPermutations([3], 2)
│      ├── choose 2 → coinPermutations([3,2], 0) ☑ print "3 2"
│
└── choose 7 → skipped (7 > 5)


☑ Output:

2 3
3 2



Solution by sir
   public static void helper(int[] coins,String prem, int amt,boolean used[]){
        //Write your code here
        if(amt==0) {
            System.out.println(prem);
            return;
        }
        if(amt<0) {
            return;
        }
        for(int i=0;i<coins.length;i++){
            if(!used[i]){
                used[i]=true;
            helper(coins,prem+coins[i]+" ",amt-coins[i],used);
                used[i]=false; // remove this to sollve Coins Permutations - 2
            }
        }
    }
```

Complexity (Time & Space):

- ⏱ Time Complexity
  - O(k^n) where n = number of coins and k ≈ (amt / smallest coin value).
  - Each recursive path explores multiple branches → exponential.
- 💾 Space Complexity
  - O(amt) (maximum recursion depth = number of coins used to make amt).

---

6. Justification / Proof of Optimality

- Every permutation is explored using recursive DFS.
- Backtracking ensures no partial permutation persists after printing.
- Suitable for small n and amt values.

---

7. Variants / Follow-Ups

- Coin Combinations - 1: Order doesn't matter (use index-based recursion).
- Coin Change (DP): Count ways, not print them.
- Coin Permutations - 2: Each coin can be used only once.

---

8. Tips & Observations

- Always reset asf(answer so far.) on backtracking.
- Be careful between "permutation" (order matters) and "combination" (order doesn't).
- Avoid global variables — pass data in parameters.

Q91: Subsets of Array

---

1. Problem Understanding

- You are given an array of distinct positive integers.
- Your task is to generate all possible subsets (the power set) of this array and print them in lexicographically sorted order.
- Each subset should contain elements in the same relative order as the original array.

---

2. Constraints

- 1 ≤ N ≤ 10
- 1 ≤ A[i] ≤ 20

---

3. Edge Cases

- Array with only one element — only one subset possible.
- Empty subset is not printed (based on the given examples).
- The array is distinct, so no duplicate subsets.

---

4. Examples

```
Example 1

Input:

3
10 15 20


Output:

10
```

```
10 15
10 15 20
10 20
15
15 20
20
```

5. Approaches

Approach 1: Recursive Backtracking (Inclusion-Exclusion method)

Idea:

- At each index, you have two choices:
  - Include the current element in the subset.
  - Exclude it and move to the next.
- This generates all possible subsets recursively.

Steps:

- Start with an empty list (current subset).
- At each step, include arr[i] and recurse.
- Then, exclude arr[i] and recurse.
- Once you reach the end of the array, print the subset (if not empty).
- Sort subsets lexicographically before printing.

Java Code:

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();

        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(arr);
        helper(arr, 0, new ArrayList<>(), ans);

        // Print subsets (excluding empty subset)
        for (List<Integer> subset : ans) {
            if (!subset.isEmpty()) {
                for (int num : subset) System.out.print(num + " ");
                System.out.println();
            }
        }
    }
    public static void helper(int[] arr, int idx, List<Integer> current,
List<List<Integer>> ans) {
        if (idx == arr.length) {
            ans.add(new ArrayList<>(current));
            return;
        }

        // Include arr[idx]
        current.add(arr[idx]);
        helper(arr, idx + 1, current, ans);

        // Backtrack (remove last element)
        current.remove(current.size() - 1);

        // Exclude arr[idx]
        helper(arr, idx + 1, current, ans);
    }
}


🌳 Recursion Tree (for arr = [10, 15, 20])
helper([], 0)
├── Include 10 → helper([10], 1)
│    ├── Include 15 → helper([10,15], 2)
│    │    ├── Include 20 → helper([10,15,20], 3) → print [10,15,20]
│    │    └── Exclude 20 → helper([10,15], 3) → print [10,15]
│    └── Exclude 15 → helper([10], 2)
│         ├── Include 20 → helper([10,20], 3) → print [10,20]
│         └── Exclude 20 → helper([10], 3) → print [10]
└── Exclude 10 → helper([], 1)
     ├── Include 15 → helper([15], 2)
     │    ├── Include 20 → helper([15,20], 3) → print [15,20]
     │    └── Exclude 20 → helper([15], 3) → print [15]
     └── Exclude 15 → helper([], 2)
          ├── Include 20 → helper([20], 3) → print [20]
          └── Exclude 20 → helper([], 3) → print []


(We skip printing the empty subset in final output)
```

Complexity (Time & Space):

- ⏱ Time Complexity
  - O(2^N) subsets possible.
  - For each subset, up to O(N) work (copying/printing).
  - ☑ Overall: O(N × 2^N)
- 💾 Space Complexity
  - Recursion stack: O(N)
  - Storage for subsets: O(N × 2^N)

---

6. Justification / Proof of Optimality

- This approach explores every combination via inclusion/exclusion recursion — the standard power set generation pattern. Sorting ensures lexicographic output.

---

7. Variants / Follow-Ups

- Subsets including empty subset (LeetCode “Subsets” problem).
- Subsets with duplicates (use a set to avoid repetition).
- Iterative (bitmask) approach to generate all subsets.

---

8. Tips & Observations

- For lexicographic output → sort input first.
- Each level in recursion represents one element’s inclusion/exclusion.
- Useful recursion template for many subset/combinations problems.

Q92: Subsets 2

---

1. Problem Understanding

- Given an array nums that may contain duplicates, print all unique subsets (non-empty), sorted in lexicographical order, and each subset itself in sorted order.
- Subsets are combinations where order doesn’t matter, but duplicates must be handled carefully.

---

2. Constraints

- 1 <= nums.length <= 10
- -10 <= nums[i] <= 10
- Duplicates may exist
- Subsets must be unique and ignore the empty set

---

3. Edge Cases

- Array has all duplicates → [1,1,1]
- Array has negative numbers → [-1,0,1]
- Only one element → [5]

---

4. Examples

```
Input
3
2 1 2
```

```
Output
1
1 2
1 2 2
2
2 2

Explanation

All unique subsets of sorted [1, 2, 2] are:
[1], [1,2], [1,2,2], [2], [2,2]
```

---

5. Approaches

Approach 1: Recursion with List<List>

Idea:

- Use backtracking to explore two choices at every step:
  - Include current element
  - Exclude current element
  - But to avoid duplicates:
  - Sort the input array
  - Skip identical elements at the same recursion depth (if (i > idx && nums[i] == nums[i - 1]) continue)

Steps:

- Sort the array
- Create a recursive function helper(idx, current, ans)
- Add current subset to answer list
- For each element from idx → n:
- Skip duplicates
- Include the element and recurse
- Backtrack (remove element)
- Remove the empty subset before returning

Java Code:

```java
import java.util.*;

public class Subsets2 {
    public static List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums); // Step 1: sort
        List<List<Integer>> ans = new ArrayList<>();
        helper(nums, 0, new ArrayList<>(), ans);
        // Removes all empty subsets ([]) from the result list.
        // Equivalent to: ans.removeIf(list -> list.isEmpty());
        ans.removeIf(List::isEmpty); // remove empty subset
        return ans;
    }

    static void helper(int[] nums, int idx, List<Integer> current,
List<List<Integer>> ans) {
        ans.add(new ArrayList<>(current));
        for (int i = idx; i < nums.length; i++) {
            if (i > idx && nums[i] == nums[i - 1]) continue; // skip duplicates
            current.add(nums[i]);
            helper(nums, i + 1, current, ans);
            current.remove(current.size() - 1); // backtrack
        }
    }

    public static void main(String[] args) {
        int[] nums = {2, 1, 2};
        List<List<Integer>> result = subsetsWithDup(nums);
        for (List<Integer> l : result) System.out.println(l);
    }
}


🌳 Recursion Tree (Example: nums = [1,2,2])
Start → []
|
|-- Include 1 → [1]
|      |
|      |-- Include 2 → [1,2]
|      |      |
|      |      |-- Include 2 → [1,2,2]
|      |      └-- Exclude → [1,2]
|      └-- Exclude → [1]
|
|-- Exclude 1 → []
       |
       |-- Include 2 → [2]
       |      |
       |      |-- Include 2 → [2,2]
       |      └-- Exclude → [2]


Collected subsets (excluding empty):
[1], [1,2], [1,2,2], [2], [2,2]
```

Complexity (Time & Space):

- Time Complexity:
  - Generating all subsets → O(2^n × n)
    - There are 2^n subsets.
    - Copying each subset (up to size n) costs O(n).
  - ➡ Overall: O(2^n × n)
- Space Complexity:

  - Recursion stack → O(n) (depth of recursion tree).
  - Result list storage → O(2^n × n) (to hold all subsets).
  - Temporary list (current) → O(n) at max depth.
  - ➡ Overall: O(2^n × n)

Approach 2: Convert List<List> → int[][] (for judge)

Idea:

- Judges often require 2D arrays instead of List<List<>>.
- After generating all subsets, convert the result manually.

Java Code:

```java
import java.io.*;
import java.util.*;

class Solution {
    static int[][] solve(int n, int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> ans = new ArrayList<>();
        helper(nums, 0, new ArrayList<>(), ans);

        ans.removeIf(List::isEmpty);

        int[][] result = new int[ans.size()][];
        for (int i = 0; i < ans.size(); i++) {
            List<Integer> subset = ans.get(i);
            result[i] = new int[subset.size()];
            for (int j = 0; j < subset.size(); j++) {
                result[i][j] = subset.get(j);
            }
        }
        return result;
    }

    static void helper(int[] nums, int idx, List<Integer> current,
List<List<Integer>> ans) {
        ans.add(new ArrayList<>(current));
        for (int i = idx; i < nums.length; i++) {
            if (i > idx && nums[i] == nums[i - 1]) continue;
            current.add(nums[i]);
            helper(nums, i + 1, current, ans);
            current.remove(current.size() - 1);
        }
    }
}
```

Complexity (Time & Space):

- Time Complexity:

    - Subset generation → O(2^n × n) (same as above).
    - Conversion (List<List> → int[][]) → O(2^n × n) (copying all elements).
    - ➡ Overall: O(2^n × n)
- Space Complexity:
    - Recursion stack → O(n)
    - Result (2D array) → O(2^n × n)
    - Temporary subset array (during conversion) → O(1) (built directly into result).
    - ➡ Overall: O(2^n × n)

---

6. Justification / Proof of Optimality

- Sorting ensures subsets are lexicographically ordered.
- The skip condition prevents duplicate subsets.
- Manual conversion to int[][] gives flexibility for judge-based problems.

---

7. Variants / Follow-Ups

- Subsets I: no duplicates → simpler version.
- Combination Sum II: similar duplicate handling but with sum constraints.
- Permutations II: same duplicate handling logic but for permutation ordering.

---

8. Tips & Observations

- While converting List<List> → int[][]
    - Never initialize with a fixed [n][n] matrix.
    - Use dynamic sizing based on subset list lengths.
        - int[][] result = new int[ans.size()][];
    - Each inner list can have different lengths, so create it dynamically:
        - result[i] = new int[l.size()];
- Use nested loops to fill values safely.
- Don't forget to sort input array before recursion.
- Always remove the empty subset if question asks to ignore it.

Q93: Valid Sudoku

---

1. Problem Understanding

- You're given a 9×9 Sudoku board (partially filled) with digits 1–9 and . representing empty cells.
- You must determine if the current board is valid, i.e., it follows all Sudoku rules for filled cells.
- Sudoku Rules:
    - Each row must have digits 1–9 without repetition.
    - Each column must have digits 1–9 without repetition.
    - Each of the nine 3×3 sub-boxes must also have digits 1–9 without repetition.

- 👉 You don’t have to solve the Sudoku — just check for validity of existing numbers.

---

2. Constraints

- Grid size: 9 × 9
- board[i][j] = '.' or '1'–'9'
- Validate only filled cells.

---

3. Edge Cases

- All cells are empty → ☑ valid.
- Duplicate number in any row/column/box → ✕ invalid.
- Invalid characters (not . or 1–9) → ✕ invalid.
- Partially filled but consistent → ☑ valid.

---

4. Examples

```
Input
. 6 9 1 2 . . 8 3
. 5 . 3 6 7 2 . 9
3 . 2 5 8 . 6 1 7
1 2 5 9 . 3 8 . 6
. 3 . 8 1 . 9 2 4
4 9 . 2 7 6 3 5 1
. 1 . 6 9 8 7 3 5
9 . . 4 . . 1 6 8
5 8 6 7 3 1 4 9 2

Output
correct
```

---

5. Approaches

Approach 1: Using Boolean Arrays (Straightforward & Clear)

Idea:

- Maintain three boolean matrices:
  - rows[9][10] → marks if a digit has appeared in a row
  - cols[9][10] → marks if a digit has appeared in a column
  - boxes[9][10] → marks if a digit has appeared in a 3×3 box
- If any duplicate found → invalid Sudoku.

Java Code:

```
💻 Main Function
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    char[][] board = new char[9][9];
    for (int r = 0; r < 9; r++) {
        for (int c = 0; c < 9; c++) {
            board[r][c] = sc.next().charAt(0);
        }
    }
    System.out.println(isValidSudoku(board) ? "correct" : "incorrect");
}

⚙️ Helper Function
static boolean isValidSudoku(char[][] board) {
    boolean[][] rows = new boolean[9][10];
    boolean[][] cols = new boolean[9][10];
    boolean[][] boxes = new boolean[9][10];

    for (int r = 0; r < 9; r++) {
        for (int c = 0; c < 9; c++) {
            char ch = board[r][c];
            if (ch == '.') continue;
            int d = ch - '0';
            int boxIndex = (r / 3) * 3 + (c / 3);
            if (rows[r][d] || cols[c][d] || boxes[boxIndex][d]) return false;
            rows[r][d] = cols[c][d] = boxes[boxIndex][d] = true;
        }
    }
    return true;
}
```

Complexity (Time & Space):

- ⏱️ Time Complexity (Approach 1)
  - Process all 81 cells → O(81) = O(1)
  - Constant-time checks and updates → O(1)
- 💾 Space Complexity (Approach 1)
  - Three 9×10 boolean matrices → O(1)
- ☑️ Overall
  - Time: O(1)
  - Space: O(1)

Approach 2: Bitmasking (Optimized Space)

Idea:

- Use integer bitmasks to represent seen digits in rows, columns, and boxes.
- Each digit d corresponds to a bit (1 << d).

Java Code:

```
static boolean isValidSudokuBit(char[][] board) {
    int[] rows = new int[9];
    int[] cols = new int[9];
    int[] boxes = new int[9];

    for (int r = 0; r < 9; r++) {
        for (int c = 0; c < 9; c++) {
            char ch = board[r][c];
            if (ch == '.') continue;
            int d = ch - '0';
            int bit = 1 << d;
            int boxIndex = (r / 3) * 3 + (c / 3);

            if ((rows[r] & bit) != 0 || (cols[c] & bit) != 0 || (boxes[boxIndex] &
bit) != 0)
                return false;

            rows[r] |= bit;
            cols[c] |= bit;
            boxes[boxIndex] |= bit;
        }
    }
    return true;
}
```

Complexity (Time & Space):

- ⏱ Time Complexity (Approach 2)
    - Check all cells → O(1)
    - Each bitwise operation → O(1)
- 💾 Space Complexity (Approach 2)
    - Three integer arrays (rows, cols, boxes) → O(1)
- ☑ Overall
    - Time: O(1)
    - Space: O(1)

Approach 3: HashSet (Concise & Readable)

Idea:

- Use a single Set to track unique combinations like:
- "5 in row 3", "5 in col 6", "5 in box 1-2"
- If any duplicate string appears → invalid Sudoku.

Java Code:

```
static boolean isValidSudokuHashSet(char[][] board) {
    Set<String> seen = new HashSet<>();

    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            char num = board[i][j];
            if (num == '.') continue;

            if (!seen.add(num + " in row " + i) ||
                !seen.add(num + " in col " + j) ||
                !seen.add(num + " in box " + i/3 + "-" + j/3))
                return false;
        }
    }
    return true;
}
```

Complexity (Time & Space):

- Time: O(1)
- Space: O(1)
- Pros: Easy to code and understand
- Cons: Slightly more memory than bitmasking

Approach 4: Recursive Validation (DFS Style)

Idea:

- Validate the board recursively cell by cell.
- If a filled cell violates Sudoku rules → return false immediately.

Java Code:

```
    static boolean validate(char[][] board, int row, int col) {
        if (row == 9) return true; // reached end
        if (col == 9) return validate(board, row + 1, 0);
        if (board[row][col] == '.') return validate(board, row, col + 1);

        if (!isSafe(board, row, col, board[row][col])) return false;

        return validate(board, row, col + 1);
    }

    static boolean isSafe(char[][] board, int r, int c, char num) {
        for (int i = 0; i < 9; i++) {
            if (i != c && board[r][i] == num) return false;
            if (i != r && board[i][c] == num) return false;
        }

        int boxRow = (r / 3) * 3;
        int boxCol = (c / 3) * 3;

        for (int i = boxRow; i < boxRow + 3; i++) {
            for (int j = boxCol; j < boxCol + 3; j++) {
                if ((i != r || j != c) && board[i][j] == num) return false;
            }
        }
        return true;
    }


    🌳 Recursion Visualization

    For the input:

    1 . .
    . 2 .
    . . 3


    Recursion Flow:

    validate(0,0) → checks '1' → safe ✅ → validate(0,1)
    validate(0,1) → '.' → skip → validate(0,2)
    validate(0,2) → '.' → skip → validate(1,0)
    ...
    validate(2,2) → '3' → safe ✅ → end reached → returns true


    All cells validated → board is correct ✅
```

Complexity (Time & Space):

- ⏱️ Time Complexity (Approach 4)
  - Each of 81 cells validated once.
  - Each validation scans:
  - 9 cells in row
  - 9 cells in col
  - 9 in box → O(27) = O(1)
  - Total: O(81×27) ≈ O(1)
- 💾 Space Complexity (Approach 4)
  - Recursive stack depth ≤ 81 → O(1)
- ✅ Overall
  - Time: O(1)
  - Space: O(1)
  - Useful for conceptual clarity.

---

6. Justification / Proof of Optimality

- The time and space complexities are constant because the Sudoku grid size is always 9×9, meaning all loops and checks run a fixed number of times.
- All approaches (boolean arrays, bitmasking, HashSet, recursion) simply verify constraints — they don't modify or solve the board.
- The boolean array method is most efficient and commonly used in interviews due to:

- O(1) lookups and updates
- Simple readability
- The bitmasking approach achieves the same logic in a space-optimized manner using integer operations.
- The recursive approach validates conceptually how Sudoku rules propagate cell by cell but is less practical for performance-sensitive environments.

---

7. Variants / Follow-Ups

- Sudoku Solver 🔍
  - Extend this validation to fill empty cells (.) recursively (backtracking).
  - Each step places a valid digit, checks constraints using isSafe, and continues recursively.
- Sudoku Generator 🧩
  - Create valid Sudoku puzzles by generating a solved board first, then remove random cells while keeping it solvable.
- Partial Sudoku Validator (Non-9×9) 🧠
  - Adapt validation logic for n×n Sudoku where n is a perfect square (e.g., 4×4, 16×16).
  - Formula for sub-box size becomes sqrt(n).
- Sudoku Validator for Streams ⚡
  - Validate a Sudoku board as numbers stream in (online validation).
  - Maintain hashmaps or bitmasks dynamically while reading input.
- Error Reporting Variant 📃
  - Instead of boolean return, return coordinates of invalid cells for debugging or UI display.

---

8. Tips & Observations

- Formula for box index: (r / 3) * 3 + (c / 3) → maps 9 subgrids uniquely.
- Short-circuit (return false) early to save time.
- Iterative methods are faster; recursion helps understand constraints deeply.
- All approaches have constant complexity because board size is fixed (9×9).

Q94: Combination Sum 3

---

1. Problem Understanding

- You are given two integers k and n.
- Your task is to find all unique combinations of k numbers (from 1–9) that add up to n.
- Rules:
  - Each number can be used at most once.
  - Only numbers 1–9 are allowed.
  - You must return all unique combinations that satisfy the sum condition.

---

2. Constraints

- 2 <= k <= 9
- 1 <= n <= 60
- Only digits 1–9 allowed once per combination.

---

3. Edge Cases

- If n < smallest possible sum of k numbers → return empty.
- If n > largest possible sum of k numbers → return empty.
- If no valid combination exists → return empty list.
- k = 1 → Only return [n] if 1 <= n <= 9.

---

4. Examples

```
Input:
k = 3, n = 8

Output:

1 2 5
1 3 4


Explanation:
All 3-number combinations from 1–9 that sum to 8 are [1,2,5] and [1,3,4].
```

---

5. Approaches

Approach 1: Recursive Backtracking (DFS)

Idea:

- We explore all numbers from 1 to 9 recursively and build combinations:
- Include a number → reduce n by that number and decrement k.
- Skip it → move to the next.
- Stop when:
  - n == 0 && k == 0 → valid combination.
  - n < 0 or k < 0 → invalid path.

Steps:

- Start from number 1.
- At each step, choose to:
  - Include current number in the path.
  - Recurse with next number.
- If sum becomes 0 and size = k → add to result.
- Backtrack by removing the last number.

Java Code:

```
import java.util.*;

class Solution {
    public static List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> res = new ArrayList<>();
        helper(1, k, n, new ArrayList<>(), res);
        return res;
    }

    static void helper(int start, int k, int n, List<Integer> temp,
List<List<Integer>> res) {
        if (n == 0 && temp.size() == k) {
            res.add(new ArrayList<>(temp));
            return;
        }
        if (n < 0 || temp.size() > k) return;

        for (int i = start; i <= 9; i++) {
            temp.add(i);
            helper(i + 1, k, n - i, temp, res);
            temp.remove(temp.size() - 1); // backtrack
        }
    }
}
🌳 Recursion Tree Example

For k=3, n=7:

helper(1,3,7,[])
 ├── pick 1 → helper(2,3,6,[1])
 │      ├── pick 2 → helper(3,3,4,[1,2])
 │      │      ├── pick 3 → helper(4,3,1,[1,2,3])
 │      │      └── pick 4 → helper(5,3,0,[1,2,4]) ✅ [1,2,4]
 │      └── pick 3 → helper(4,3,3,[1,3]) ...
 └── pick 2 → helper(3,3,5,[2]) ...


➡️ Final valid combinations: [1,2,4], [1,3,3] (invalid due to repetition,
skipped).
```

Complexity (Time & Space):

- Time Complexity: O(2^9) → Each number from 1–9 has two choices (include/exclude).
- Space Complexity: O(k) → Recursion stack depth + temporary list.

Approach 2: Recursive with Pruning (Optimized)

Idea:

- Same as Approach 1, but prune unnecessary branches early:
  - Stop recursion when n < 0 or temp.size() == k.
  - Also limit upper loop bound: i <= 9 - (k - temp.size()) + 1 (ensures enough numbers remain).

Java Code:

```
import java.util.*;

class Solution {
    public static List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> ans = new ArrayList<>();
        backtrack(1, k, n, new ArrayList<>(), ans);
        return ans;
    }

    static void backtrack(int start, int k, int n, List<Integer> curr,
List<List<Integer>> ans) {
        if (n == 0 && curr.size() == k) {
            ans.add(new ArrayList<>(curr));
            return;
        }

        for (int i = start; i <= 9; i++) {
            if (n - i < 0 || curr.size() >= k) break; // pruning
            curr.add(i);
            backtrack(i + 1, k, n - i, curr, ans);
            curr.remove(curr.size() - 1);
        }
    }
}
🌳 Recursion Tree Visualization (for k=3, n=8)
(1) → (2) → (5) ☑ sum=8
(1) → (3) → (4) ☑ sum=8
(1) → (4) → (5) ❌ sum=10 (pruned)
(2) → (3) → (5) ❌ sum=10 (pruned)
```

Complexity (Time & Space):

- Time Complexity: O(C(9, k)) → Only valid combinations of 9 choose k are explored.
- Space Complexity: O(k) → Recursion + path storage.
- Overall: Most efficient for given constraints.

Approach 3: Bitmask Enumeration (Iterative)

Idea:

- We can use bitmasking to represent subsets of {1..9}.
- Each subset corresponds to one possible combination of numbers.
- For example:
  - Bitmask 010011001 → represents numbers {2,6,7,9}.

- We iterate through all possible bitmasks (from 1 to (1<<9)-1), and for each mask:
  - Count how many bits are set → that's the number of elements (should equal k).
  - Compute the sum of those selected numbers → should equal n.
  - If both match, record the combination.

Steps:

- Loop through all 512 (2^9) possible subsets.
- For each subset:
  - Extract elements (1–9) based on bits set.
  - If count == k and sum == n → add to result.

Java Code:

```
import java.util.*;

class Solution {
    public static List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> ans = new ArrayList<>();

        // Iterate over all subsets of {1,2,3,4,5,6,7,8,9}
        for (int mask = 1; mask < (1 << 9); mask++) {
            List<Integer> curr = new ArrayList<>();
            int sum = 0;
            for (int i = 0; i < 9; i++) {
                if ((mask & (1 << i)) != 0) {
                    curr.add(i + 1);
                    sum += i + 1;
                }
            }
            if (curr.size() == k && sum == n) ans.add(curr);
        }

        return ans;
    }
}
Example Walkthrough (k=3, n=7)

Mask = 001011001 → represents {3,4,7}

sum = 14 → invalid

Mask = 000101101 → represents {1,3,5,7}

size = 4 → invalid

Mask = 000011011 → represents {1,2,4} ☑

size = 3, sum = 7 → valid combination
```

Complexity (Time & Space):

- Time Complexity: O(2^9 × 9) → Iterate all 512 subsets × up to 9 operations each.
- Space Complexity: O(k) → Temporary subset storage.
- Overall: Constant-time feasible (small fixed set), clean iterative alternative.

---

6. Justification / Proof of Optimality

- All three approaches (Recursive DFS, Optimized DFS with Pruning, and Bitmask Enumeration) explore subsets of numbers from 1–9 to find combinations that sum up to n.
- The Recursive DFS approach is intuitive — it systematically explores all inclusion/exclusion paths — but may explore unnecessary branches.
- The Optimized DFS (with Pruning) improves performance by stopping recursion early when the current sum exceeds the target (n) or when too many numbers are picked (path.size() > k).
    - This drastically reduces the search space while maintaining correctness.
- The Bitmask Enumeration method, on the other hand, replaces recursion with an iterative and memory-efficient subset generation technique using bitwise operations.
    - It’s particularly clean and effective when the range of numbers is small (like 1–9), since there are only 2^9 = 512 subsets to check.

---

7. Variants / Follow-Ups

- Combination Sum I – Can reuse same number (no upper bound of 9).
- Combination Sum II – Numbers may repeat in input, avoid duplicates.
- Combination Sum IV – Order matters (DP-based counting problem).

---

8. Tips & Observations

- For small ranges (like {1..9}), bitmasking can be just as efficient as recursion and often simpler to debug.
- Pruning is the biggest performance boost — it avoids exploring paths that can never meet the target sum.
- Always increment start in recursive calls to avoid duplicate combinations and maintain ascending order.
- Use backtracking (remove last element) after each recursive call to restore the state.
- The bitmask method is purely iterative — no recursion stack, which can be advantageous in constrained environments.
- When using recursion, always consider base cases carefully:
- If sum == target and size == k → valid combination.
- If sum > target or size > k → stop exploring.
- Keep combinations sorted lexicographically by iterating numbers in increasing order.
- Both recursive and bitmask methods are efficient enough here since the input space is constant (only 9 elements).
