Problem Solving Workshop

Phase 1: Foundations

Instructor: Abhishek Bansal
Academic Year: 2025

Course Overview

This two-week workshop introduces foundational problem-solving techniques through classic puzzles, algorithmic strategies, and real-world modeling exercises. Students will build confidence in recursion, search algorithms, data structures, and collaborative design.

Phase 1 Curriculum Overview

1. **Sudoku – Baseline Quiz**
   - Sudoku, logic-grid, and handshake-count puzzles ($N(N-1)/2$)
   - Survey: "How did you solve each?"

   Outcome: Gauge current skills and problem-solving intuition.

2. **N-Queens Backtracking**
   - Explain rules
   - Code stub: `def place_queens(row, board): ...`
   - Test on $N = 4$ and $N = 8$

   Outcome: Hands-on recursion and pruning.

3. **Sudoku Solver**
   - Constraint propagation and backtracking
   - Fill in logic for provided Python function signature

   Outcome: Reinforce state-space search.

4. **Graphs via Word Ladders – Mazes**
   - Introduction to graphs (nodes, edges)
   - BFS/DFS on word ladders
   - Maze-path visualization

   Outcome: Bridge puzzles to graph algorithms.

5. **Group Challenge – Extensions**

- Teams choose from N-Queens, Sudoku, word ladders, mazes, or optional puzzles (Lights Out, Josephus, Tower of Hanoi, Coin Change, Word Search)
- Sketch and pitch an improved solution

Outcome: Collaboration and peer teaching.

6. **Queues – Real-World Simulation**

   - Live-code a bank-line simulator using a queue
   - Stack demo: undo/redo

   Outcome: Apply data structures in real-world contexts.

7. **Game AI – Tic-Tac-Toe**

   - Provide stub: `def minimax(board, player): ...`
   - Implement and play versus AI

   Outcome: Expose decision-tree logic.

8. **Event-Driven Modeling: Traffic / Queue**

   - Model traffic lights or a bank queue with events
   - Discuss real-world modeling challenges

   Outcome: Experience event-driven thinking.

9. **Ideation Workshop**

   - Brainstorm campus/community problems (library booking, canteen queues)
   - Structured voting on top ideas

   Outcome: Nurture interdisciplinary vision.

10. **Mini-Pitch – Reflection**

    - Three-minute team pitches with solution outlines
    - Reflection: "What did we learn?"

    Outcome: Build confidence and prepare for Phase 2 capstone.

Setup Instructions

Students should follow these steps to access materials:

- Sign up or log into Replit: replit.com
- Take the Quiz on Google Forms: Click here
- View the Progress Tracker: Click here

Puzzles & Baseline Quiz

Ice-Breaker Puzzles

Complete the following exercises in 20 minutes:

a. **Sudoku (4×4 grid)**: Fill each row, column, and 2×2 block with numbers 1–4 exactly once.

b. **Logic Grid Puzzle**: Three students (A, B, C) each solved *one* of three puzzles (maze, riddle, sudoku-mini) in *one* of three times (5, 10, 15 minutes), and no two students chose the same puzzle or time.

   - A did **not** take 5 minutes.
   - The riddle was solved in 10 minutes.
   - C solved the maze.
   - The sudoku-mini was completed in the shortest time (5 minutes).

   Determine who solved which puzzle and in what time.

c. **Handshake Count**: In a group of 7, every pair shakes hands exactly once. How many total handshakes occur?

Baseline Quiz

Answer the following in the next 10 minutes:

1. Compute the handshake count for $N = 7$ and verify using the formula $\frac{N(N-1)}{2}$.
2. Write pseudocode for computing handshake count for a general N:

```
function handshakeCount(N):
    # Each of N people shakes hands with N-1 others
    # Divide by 2 to avoid double-counting
    return N * (N - 1) / 2
```

3. Reflect: Which problem-solving heuristic did you use most (decomposition, pattern recognition, abstraction)?

Backtracking I – N-Queens

Problem Description

Place one queen in each row of an $N \times N$ chessboard so that no two queens attack each other. A queen attacks along its row, column, and both diagonals.

Recursive Pseudocode

Use the following stub to implement your solution:

```
function placeQueens(row, board):
  if row > N:
    printSolution(board)
    return
  for col = 1 to N:
    if isSafe(row, col, board):
      board[row] = col
      placeQueens(row + 1, board)
      board[row] = 0  # backtrack

function isSafe(r, c, board):
  for prevRow = 1 to r - 1:
    prevCol = board[prevRow]
    if prevCol == c or
       abs(prevCol - c) == abs(prevRow - r):
      return false
  return true
```

Try it online: Run the N-Queens demo on Replit

Exercise

- Implement the pseudocode and print one valid arrangement for $N = 4$.
- Extend your code to count all solutions for $N = 4$ and report the total.
- *(Optional)* Test your solver for $N = 8$ and note the number of solutions.

Outcome: Reinforce backtracking and recursion.

Sudoku Solver Session

Problem Description

Students will implement a backtracking algorithm to solve a 4×4 Sudoku. Each row, column, and 2×2 block must contain numbers 1–4 exactly once.

Recursive Backtracking Stub

Use this function signature:

```
function solveSudoku(grid):
    if no empty cells:
        printSolution(grid)
        return true
    pick an empty cell (r,c)
    for num = 1 to 4:
        if isValid(grid, r, c, num):
            grid[r][c] = num
            if solveSudoku(grid): return true
            grid[r][c] = 0   # backtrack
    return false
```

Try it online: Run the Sudoku Solver on Replit

Exercises

1. a. Write a helper function `findEmptyCell(grid)` that returns the coordinates (r, c) of the next empty cell (`grid[r][c] == 0`), or `null` if none remain.

   b. Fill in the body of `solveSudoku(grid)` using your `findEmptyCell` helper.

2. Run your solver on the sample 4×4 puzzle:

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

3. *(Optional)* Extend your solver to handle full 9×9 puzzles.

Outcome: Reinforce backtracking and constraint propagation.

Graphs via Word Ladders & Mazes

Problem Description

Students will explore graphs (vertices and edges) and then apply graph-search to two puzzles:

- **Word Ladder**
  Transform one word into another by changing exactly one letter at a time; each intermediate word must appear in the dictionary.

- **Maze Navigation**
  Navigate a 2D ASCII-maze from a start (S) to an exit (E), moving only through open cells ".".

Breadth-First Search (BFS) Stub

Use this pseudocode to find the shortest path in an unweighted graph (queue operations implicit):

```
function BFS(start, goal):
  Q ← empty queue
  enqueue(Q, start); mark start visited
  while Q not empty:
    v ← dequeue(Q)
    if v == goal: return path-to(v)
    for each neighbor u of v:
      if not visited(u):
        mark u visited; enqueue(Q, u)
  return "no path"
```

Depth-First Search (DFS) Stub

Use this pseudocode to traverse all reachable nodes:

```
function DFS(v):
  mark v visited
  for each neighbor u of v:
    if not visited(u):
      DFS(u)
```

Sample Inputs

- **Word Ladder**

  ```
  beginWord = "hit"
  endWord   = "cog"
  dict      = {"hot","dot","dog","lot","log","cog"}
  ```

- **Maze (ASCII)**

```
########
#S..#.#.#
#.#.#.#.#
#..E##.##
########
```

(Here `S` = start, `E` = exit, `.` = open, `#` = wall.)

Try it online: Run the BFS Word-Ladder demo on Replit
Try it online: Run the BFS Maze Navigation demo on Replit
Try it online: Run the DFS Word-Ladder demo on Replit
Try it online: Run the DFS Maze Navigation demo on Replit

Exercises

1 Implement **Word Ladder**: use the BFS stub to compute the shortest transformation from *beginWord* to *endWord*, returning the list of intermediate words.

2 **Maze-path Visualization**: represent the maze as a graph (cells as vertices, edges between adjacent open cells). Use BFS to find a shortest path from S→E, and output the path either as a list of coordinates or directional steps.

3 *(Optional)* Compare DFS vs BFS on the same maze: implement a DFS-based solver and observe whether it always finds the shortest path.

Outcome: Bridge puzzles to graph algorithms, reinforcing that both recursion (DFS) and queue-based search (BFS) generalize across problem domains.

Group Challenge — Extensions

After three days of "learning by doing" (N-Queens, Sudoku, Word Ladders, Mazes), today we'll survey several classic puzzles and their high-level algorithms. You won't write full code now, but you should:

1. See how each problem is posed.
2. Follow the pseudocode sketch.
3. Think of one small way you might extend or improve it.

Puzzles & Pseudocode Stubs

- **Lights Out**
  A grid of lights toggles itself and its orthogonal neighbors when you press a cell. Goal: turn all lights off.

```
function solveLightsOut(state):
    if all cells are OFF:
        return solution  # base case
    pick a cell (r,c)
    toggle(state, r, c)            # flip this cell + neighbors
    if solveLightsOut(state):
        return solution
    toggle(state, r, c)            # backtrack
    return failure
```

- **Josephus**
  `n` people stand in a circle; every `k`th person is eliminated until one remains.

```
function josephus(n, k):
    if n == 1:
        return 0                  # survivor index in zero-based
    prev = josephus(n-1, k)
    return (prev + k) mod n       # re-index into current circle
```

- **Tower of Hanoi**
  Move `n` disks among three pegs, one disk at a time, never placing larger atop smaller.

```
function hanoi(n, src, aux, dest):
    if n == 1:
        move disk from src to dest
        return
    hanoi(n-1, src, dest, aux)    # move top n-1 to auxiliary
    move disk from src to dest
    hanoi(n-1, aux, src, dest)    # move n-1 from auxiliary to dest
```

- **Coin Change (Dynamic)**
  Count ways to make amount `X` using unlimited coins of given denominations.

```
function countWays(coins, X):
    dp = array[0..X] with dp[0] = 1
    for each coin in coins:
        for amt = coin to X:
            dp[amt] += dp[amt - coin]
    return dp[X]
```

- **Word Search**
  Given a grid of letters, determine if a target word exists by moving through adjacent cells (up/down/left/right) without reuse.

```
function exists(word, grid):
    for each cell (r,c):
        if dfs((r,c), 0):
            return TRUE
    return FALSE

function dfs(pos, i):
    if i == length(word):
        return TRUE               # full word found
    if grid[pos] != word[i]:
        return FALSE
    mark pos visited
    for each neighbor n of pos:
        if not visited(n) and dfs(n, i+1):
            return TRUE
    unmark pos
    return FALSE
```

Exercises

1. **Trace Lights Out.** On this 3×3 sample, show the first two presses (cells) you'd choose to begin solving:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

2. **Compute Josephus.** For `n=7, k=3`, who survives? Show your calls and return values from the recurrence.

3. **(Optional)** Pick one of today's puzzles and sketch a small extension. For instance:
   - Lights Out on a torus
   - Variable k in Josephus
   - Tower of Hanoi variants
   - Greedy heuristics for Coin Change
   - Word Search backtracking

   Be ready to describe your idea in two sentences.

Outcome: Familiarize everyone with a broader family of recursive, dynamic-programming, and graph-search problems—and give you a chance to look under the hood of each algorithm, even if you're not writing full code today.

Queues – Real-World Simulation

Problem Description:
Model a bank teller's line as a queue: customers arrive, wait in FIFO order, and are served one at a time. We'll also demo a stack-based undo/redo of teller actions.

Live-Code: Bank-Line Simulator
Use a `queue` to enqueue arriving customers and dequeue for service.

Stack Demo: Undo/Redo
Record each served customer on an `undoStack`. Allow "undo" (move from undoStack → redoStack) and "redo" (move back).

Pseudocode Stub

```
function serveBank(arrivals):
    Q = empty queue
    for customer in arrivals:
        enqueue(Q, customer)

    while not empty(Q):
        c = dequeue(Q)
        serve(c)
        push(undoStack, c)

    function undo():
        if not empty(undoStack):
            c = pop(undoStack)
            unserve(c)
            push(redoStack, c)

    function redo():
        if not empty(redoStack):
            c = pop(redoStack)
            serve(c)
            push(undoStack, c)
```

Visual Walkthrough

A quick state-diagram table (front→back):

Step	Queue (front→back)	`undoStack`	`redoStack`
start	$[A, B, C, D]$	[]	[]
serve A	$[B, C, D]$	$[A]$	[]
serve B	$[C, D]$	$[A, B]$	[]
undo	$[C, D]$	$[A]$	$[B]$
redo	$[C, D]$	$[A, B]$	[]

Complexity Note

- All queue operations (‘enqueue‘/‘dequeue‘) and stack operations (‘push‘/‘pop‘) run in `O(1)` time per action.

Sample Input / Scenario

- Arrival order: `[Alice, Bob, Carol, Dave]`
- Actions:
  - Serve Alice, Bob, Carol
  - Undo twice
  - Redo once
- Track queue contents and stack states after each step.

Exercises

1. Given arrivals `[Alice, Bob, Carol]`, serve two, undo one, then list:
   - Remaining queue
   - Contents of `undoStack` and `redoStack`
2. Extend the stub to support “VIP” arrivals that jump to the front of the queue. Sketch your code change.

Outcome:
Apply queues and stacks in a realistic simulation—reinforcing FIFO service and undo/redo mechanics.

Game AI – Tic-Tac-Toe

Problem Description:
Build an AI opponent for Tic-Tac-Toe using a decision-tree search (*minimax*). The AI enumerates possible move sequences and chooses the optimal one.

Pseudocode Stub

```
function minimax(board, player):
    if terminal(board):
        return utility(board)          # win / loss / draw score

    if player == MAX:                  # MAX = 'O'
        bestVal = -infty
        for move in legalMoves(board):
            val = minimax(apply(board, move), MIN)
            bestVal = max(bestVal, val)
        return bestVal
    else:                              # MIN = 'X'
        bestVal = +infty
        for move in legalMoves(board):
            val = minimax(apply(board, move), MAX)
            bestVal = min(bestVal, val)
        return bestVal

function bestMove(board, player):
    best    = null
    bestVal = (player==MAX ? -infty : +infty)
    for move in legalMoves(board):
        val = minimax(apply(board, move), opposite(player))
        if (player==MAX and val > bestVal) or
           (player==MIN and val < bestVal):
            bestVal = val
            best    = move
    return best
```

Exercises

1. **Trace `minimax`.** On the board below, X (= MIN) has just played; O (= MAX) has *one* legal move left—the centre square. Show the call sequence (*minimax* → *utility*) and the utility value finally returned.

X	O	X
O	$-$	O
X	O	X

2. **Implement `utility`.** Sketch pseudocode for `utility(board)` so that it returns $+1$ if O wins, -1 if X wins, and 0 for a draw.

Outcome:
Expose the mechanics of recursive decision-tree search and utility evaluation in simple game AI.

Event-Driven Modeling: Traffic / Queue

Problem Description:
Model a traffic-light controller (or bank queue) as an *event-driven* system. Cars/lights fire *events* (arrivals / departures), and a central loop dispatches them in time order.

Live Code – Event-Loop Simulator
Demonstrate an **event priority-queue** that holds pending events while a dispatcher invokes handlers.

Pseudocode Stub

```
function runSimulation(initialEvents):
    Q = empty priority-queue          # ordered by time
    waitingList = []                  # current cars in line
    nextID = 1                        # customer IDs (c1, c2, ...)

    for e in initialEvents:
        enqueue(Q, e)

    while not empty(Q):
        (t, event) = dequeue(Q)
        handleEvent(event, t)

function handleEvent(ev, t):
    global nextID
    if ev.type == ARRIVAL:
        waitingList.append(nextID); nextID += 1
        if len(waitingList) == 1:              # queue was empty → start service
            schedule(DEPARTURE, t + serviceTime())
        schedule(ARRIVAL, t + arrivalInterval())
    elif ev.type == DEPARTURE:
        waitingList.pop(0)                     # front car departs
        if len(waitingList) > 0:
            schedule(DEPARTURE, t + serviceTime())
    updateSystemState(ev, t, len(waitingList))

function schedule(type, tNew, isPriority=False):
    if isPriority:                             # O(1) front insert
        pushFront(Q, (tNew, Event(type)))
    else:                                      # O(log n) insert
        enqueue(Q, (tNew, Event(type)))

function arrivalInterval():   return 3         # e.g. 1 car every 3s
function serviceTime():       return 4         # e.g. 1 car served in 4s
```

Sample Input / Scenario

- **Warm-up** events `[(0, ARRIVAL), (0, ARRIVAL)]`

- **Exercise set** arrivals `[0, 2, 5]`
- **Handlers**
  - **ARRIVAL** – enqueue next arrival, *increment* `queueLength`
  - **DEPARTURE** – enqueue next departure, *decrement* `queueLength`
- Track both the *event queue* `Q` and the *system state* (`queueLength = len(waitingList)`) at every step.

Exercises

1. **Trace event loop.** For arrivals at times `[0, 2, 5]`, list the first five rows of

$$(\text{step},\ Q,\ \text{dequeued}\ (t, \text{event}),\ \text{waitingList},\ \texttt{queueLength})$$

   and give a short note for each row.

2. **PRIORITY event.** Modify `schedule` so that setting `isPriority = True` forces the event to the head of `Q`.

Outcome:
Understand event-driven architecture—event queues, dispatch loops, dynamic scheduling, and real-time queue-length tracking. tracking.