**Tic-tac-toe without AI**

import java.util.Scanner;

public class nonai {

    private char[][] board;

    private char currentPlayer;

    public nonai() {

        board = new char[3][3];

        currentPlayer = 'X';

        initializeBoard();

    }

    private void initializeBoard() {

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                board[i][j] = ' ';

            }

        }

    }

    private void printBoard() {

        System.out.println("  0 1 2");

        for (int i = 0; i < 3; i++) {

            System.out.print(i + "|");

            for (int j = 0; j < 3; j++) {

                System.out.print(board[i][j] + "|");

            }

            System.out.println("\n  -----");

        }

    }

    private boolean makeMove(int row, int col) {

        if (row < 0 || row >= 3 || col < 0 || col >= 3 || board[row][col] != ' ') {

            System.out.println("Invalid move. Try again.");

            return false;

        }

        board[row][col] = currentPlayer;

        return true;

    }

    private boolean checkWin() {

        for (int i = 0; i < 3; i++) {

            if (board[i][0] != ' ' && board[i][0] == board[i][1] && board[i][1] == board[i][2]) {

                return true;

            }

            if (board[0][i] != ' ' && board[0][i] == board[1][i] && board[1][i] == board[2][i]) {

                return true;

            }

        }

        if (board[0][0] != ' ' && board[0][0] == board[1][1] && board[1][1] == board[2][2]) {

            return true;

        if (board[0][2] != ' ' && board[0][2] == board[1][1] && board[1][1] == board[2][0]) {

            return true;

        }

        return false;

    }

    private boolean isBoardFull() {

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                if (board[i][j] == ' ') {

                    return false;

                }

            }

        }

        return true;

    }

    public static void main(String[] args) {

        nonai game = new nonai();

        Scanner scanner = new Scanner(System.in);

        while (true) {

            game.printBoard();

            System.out.println("Player " + game.currentPlayer + ", enter your move (row and column): ");

            int row = scanner.nextInt();

            int col = scanner.nextInt();

            if (game.makeMove(row, col)) {

                if (game.checkWin()) {

                    game.printBoard();

                    System.out.println("Player " + game.currentPlayer + " wins!");

                    break;

                } else if (game.isBoardFull()) {

                    game.printBoard();

                    System.out.println("It's a draw!");

                    break;

                }

                game.currentPlayer = (game.currentPlayer == 'X') ? 'O' : 'X';

            }

        }

        scanner.close();

    }

}

**Ti-Tac-Toe with Ai(minmax)**

import java.util.Scanner;

public class ai {

    private static final char HUMAN\_PLAYER = 'X';

    private static final char AI\_PLAYER = 'O';

    public static void main(String[] args) {

        char[][] board = {

                {' ', ' ', ' '},

                {' ', ' ', ' '},

                {' ', ' ', ' '}

        };

        Scanner scanner = new Scanner(System.in);

        while (true) {

            printBoard(board);

            // Human move

            makeMove(board, HUMAN\_PLAYER);

            // Check for a win or a draw

            if (checkWin(board, HUMAN\_PLAYER)) {

                printBoard(board);

                System.out.println("You win!");

                break;

            } else if (isBoardFull(board)) {

                printBoard(board);

                System.out.println("It's a draw!");

                break;

            }

            // AI move

            makeAIMove(board);

            // Check for a win or a draw

            if (checkWin(board, AI\_PLAYER)) {

                printBoard(board);

                System.out.println("AI wins!");

                break;

            } else if (isBoardFull(board)) {

                printBoard(board);

                System.out.println("It's a draw!");

                break;

            }

        }

        scanner.close();

    }

    private static void printBoard(char[][] board) {

        System.out.println("  0 1 2");

        for (int i = 0; i < 3; i++) {

            System.out.print(i + "|");

            for (int j = 0; j < 3; j++) {

                System.out.print(board[i][j] + "|");

            }

            System.out.println("\n  -----");

        }

    }

    private static void makeMove(char[][] board, char player) {

        Scanner scanner = new Scanner(System.in);

        while (true) {

            System.out.println("Enter your move (row and column): ");

            int row = scanner.nextInt();

            int col = scanner.nextInt();

            if (row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == ' ') {

                board[row][col] = player;

                break;

            } else {

                System.out.println("Invalid move. Try again.");

            }

        }

    }

    private static void makeAIMove(char[][] board) {

        int[] bestMove = findBestMove(board);

        board[bestMove[0]][bestMove[1]] = AI\_PLAYER;

    }

    private static int[] findBestMove(char[][] board) {

        int[] bestMove = {-1, -1};

        int bestScore = Integer.MIN\_VALUE;

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                if (board[i][j] == ' ') {

                    board[i][j] = AI\_PLAYER;

                    int score = minimax(board, 0, false);

                    board[i][j] = ' '; // Undo the move

                    if (score > bestScore) {

                        bestScore = score;

                        bestMove[0] = i;

                        bestMove[1] = j;

                    }

                }

            }

        }

        return bestMove;

    }

    private static int minimax(char[][] board, int depth, boolean isMaximizing) {

        if (checkWin(board, HUMAN\_PLAYER)) {

            return -1;

        } else if (checkWin(board, AI\_PLAYER)) {

            return 1;

        } else if (isBoardFull(board)) {

            return 0;

        }

        if (isMaximizing) {

            int bestScore = Integer.MIN\_VALUE;

            for (int i = 0; i < 3; i++) {

                for (int j = 0; j < 3; j++) {

                    if (board[i][j] == ' ') {

                        board[i][j] = AI\_PLAYER;

                        int score = minimax(board, depth + 1, false);

                        board[i][j] = ' '; // Undo the move

                        bestScore = Math.max(bestScore, score);

                    }

                }

            }

            return bestScore;

        } else {

            int bestScore = Integer.MAX\_VALUE;

            for (int i = 0; i < 3; i++) {

                for (int j = 0; j < 3; j++) {

                    if (board[i][j] == ' ') {

                        board[i][j] = HUMAN\_PLAYER;

                        int score = minimax(board, depth + 1, true);

                        board[i][j] = ' '; // Undo the move

                        bestScore = Math.min(bestScore, score);

                    }

                }

            }

            return bestScore;

        }

    }

    private static boolean checkWin(char[][] board, char player) {

        for (int i = 0; i < 3; i++) {

            if (board[i][0] == player && board[i][1] == player && board[i][2] == player) {

                return true;

            }

            if (board[0][i] == player && board[1][i] == player && board[2][i] == player) {

                return true;

            }

        }

        if (board[0][0] == player && board[1][1] == player && board[2][2] == player) {

            return true;

        }

        if (board[0][2] == player && board[1][1] == player && board[2][0] == player) {

            return true;

        }

        return false;

    }

    private static boolean isBoardFull(char[][] board) {

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                if (board[i][j] == ' ') {

                    return false;

                }

            }

        }

        return true;

    }

}

**Tic-tac-toe using AI(Alpha-Beta Pruning)**

import java.util.\*;

public class TicTacToeAlphaBeta {

    private static final char HUMAN\_PLAYER = 'X';

    private static final char AI\_PLAYER = 'O';

    private static final char EMPTY = ' ';

    public static void main(String[] args) {

        char[][] board = {

                {EMPTY, EMPTY, EMPTY},

                {EMPTY, EMPTY, EMPTY},

                {EMPTY, EMPTY, EMPTY}

        };

        Scanner scanner = new Scanner(System.in);

        while (true) {

            printBoard(board);

            // Human move

            makeMove(board, HUMAN\_PLAYER, scanner);

            // Check for a win or a draw

            if (checkWin(board, HUMAN\_PLAYER)) {

                printBoard(board);

                System.out.println("You win!");

                break;

            } else if (isBoardFull(board)) {

                printBoard(board);

                System.out.println("It's a draw!");

                break;

            }

            // AI move

            makeAIMove(board);

            // Check for a win or a draw

            if (checkWin(board, AI\_PLAYER)) {

                printBoard(board);

                System.out.println("AI wins!");

                break;

            } else if (isBoardFull(board)) {

                printBoard(board);

                System.out.println("It's a draw!");

                break;

            }

        }

        scanner.close();

    }

    private static void printBoard(char[][] board) {

        System.out.println("  0 1 2");

        for (int i = 0; i < 3; i++) {

            System.out.print(i + "|");

            for (int j = 0; j < 3; j++) {

                System.out.print(board[i][j] + "|");

            }

            System.out.println("\n  -----");

        }

    }

    private static void makeMove(char[][] board, char player, Scanner scanner) {

        while (true) {

            System.out.println("Enter your move (row and column): ");

            int row = scanner.nextInt();

            int col = scanner.nextInt();

            if (row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == EMPTY) {

                board[row][col] = player;

                break;

            } else {

                System.out.println("Invalid move. Try again.");

            }

        }

    }

    private static void makeAIMove(char[][] board) {

        int[] bestMove = findBestMove(board);

        board[bestMove[0]][bestMove[1]] = AI\_PLAYER;

    }

    private static int[] findBestMove(char[][] board) {

        int[] bestMove = {-1, -1};

        int bestScore = Integer.MIN\_VALUE;

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                if (board[i][j] == EMPTY) {

                    board[i][j] = AI\_PLAYER;

                    int score = minimax(board, 0, false, Integer.MIN\_VALUE, Integer.MAX\_VALUE);

                    board[i][j] = EMPTY; // Undo the move

                    if (score > bestScore) {

                        bestScore = score;

                        bestMove[0] = i;

                        bestMove[1] = j;

                    }

                }

            }

        }

        return bestMove;

    }

    private static int minimax(char[][] board, int depth, boolean isMaximizing, int alpha, int beta) {

        if (checkWin(board, HUMAN\_PLAYER)) {

            return -1;

        } else if (checkWin(board, AI\_PLAYER)) {

            return 1;

        } else if (isBoardFull(board)) {

            return 0;

        }

        if (isMaximizing) {

            int bestScore = Integer.MIN\_VALUE;

            for (int i = 0; i < 3; i++) {

                for (int j = 0; j < 3; j++) {

                    if (board[i][j] == EMPTY) {

                        board[i][j] = AI\_PLAYER;

                        int score = minimax(board, depth + 1, false, alpha, beta);

                        board[i][j] = EMPTY; // Undo the move

                        bestScore = Math.max(bestScore, score);

                        alpha = Math.max(alpha, bestScore);

                        if (beta <= alpha) {

                            break; // Beta cutoff

                        }

                    }

                }

            }

            return bestScore;

        } else {

            int bestScore = Integer.MAX\_VALUE;

            for (int i = 0; i < 3; i++) {

                for (int j = 0; j < 3; j++) {

                    if (board[i][j] == EMPTY) {

                        board[i][j] = HUMAN\_PLAYER;

                        int score = minimax(board, depth + 1, true, alpha, beta);

                        board[i][j] = EMPTY; // Undo the move

                        bestScore = Math.min(bestScore, score);

                        beta = Math.min(beta, bestScore);

                        if (beta <= alpha) {

                            break; // Alpha cutoff

                        }

                    }

                }

            }

            return bestScore;

        }

    }

    private static boolean checkWin(char[][] board, char player) {

        // Check rows, columns, and diagonals for a win

        for (int i = 0; i < 3; i++) {

            if (board[i][0] == player && board[i][1] == player && board[i][2] == player) {

                return true; // Row win

            }

            if (board[0][i] == player && board[1][i] == player && board[2][i] == player) {

                return true; // Column win

            }

        }

        if (board[0][0] == player && board[1][1] == player && board[2][2] == player) {

            return true; // Diagonal win (top-left to bottom-right)

        }

        if (board[0][2] == player && board[1][1] == player && board[2][0] == player) {

            return true; // Diagonal win (top-right to bottom-left)

        }

        return false;

    }

    private static boolean isBoardFull(char[][] board) {

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                if (board[i][j] == EMPTY) {

                    return false;

                }

            }

        }

        return true;

    }

}

**Missionaries And Cannibals using BFS and DFS**

import java.util.\*;

public class MissionariesAndCannibals {

    // State class to represent the state of the problem

    static class State {

        int missionariesLeft;

        int cannibalsLeft;

        int missionariesRight;

        int cannibalsRight;

        boolean boatOnLeft;

        public State(int missionariesLeft, int cannibalsLeft, int missionariesRight, int cannibalsRight, boolean boatOnLeft) {

            this.missionariesLeft = missionariesLeft;

            this.cannibalsLeft = cannibalsLeft;

            this.missionariesRight = missionariesRight;

            this.cannibalsRight = cannibalsRight;

            this.boatOnLeft = boatOnLeft;

        }

        // Check if the state is valid

        public boolean isValid() {

            if (missionariesLeft < 0 || cannibalsLeft < 0 || missionariesRight < 0 || cannibalsRight < 0 ||

                    (missionariesLeft != 0 && missionariesLeft < cannibalsLeft) ||

                    (missionariesRight != 0 && missionariesRight < cannibalsRight)) {

                return false;

            }

            return true;

        }

        // Check if the state is goal state

        public boolean isGoal() {

            return missionariesLeft == 0 && cannibalsLeft == 0;

        }

        // Override equals and hashCode methods for hashing and comparison

        @Override

        public boolean equals(Object o) {

            if (this == o) return true;

            if (o == null || getClass() != o.getClass()) return false;

            State state = (State) o;

            return missionariesLeft == state.missionariesLeft &&

                    cannibalsLeft == state.cannibalsLeft &&

                    missionariesRight == state.missionariesRight &&

                    cannibalsRight == state.cannibalsRight &&

                    boatOnLeft == state.boatOnLeft;

        }

        @Override

        public int hashCode() {

            return Objects.hash(missionariesLeft, cannibalsLeft, missionariesRight, cannibalsRight, boatOnLeft);

        }

    }

    // Method to perform Breadth First Search

    public static void bfs() {

        Queue<State> queue = new LinkedList<>();

        Set<State> visited = new HashSet<>();

        Map<State, State> parentMap = new HashMap<>();

        State initialState = new State(3, 3, 0, 0, true);

        queue.offer(initialState);

        visited.add(initialState);

        while (!queue.isEmpty()) {

            State currentState = queue.poll();

            if (currentState.isGoal()) {

                printSolution(parentMap, currentState);

                return;

            }

            List<State> nextPossibleStates = getNextStates(currentState);

            for (State nextState : nextPossibleStates) {

                if (!visited.contains(nextState)) {

                    queue.offer(nextState);

                    visited.add(nextState);

                    parentMap.put(nextState, currentState);

                }

            }

        }

    }

    // Method to perform Depth First Search

    public static void dfs() {

        Stack<State> stack = new Stack<>();

        Set<State> visited = new HashSet<>();

        Map<State, State> parentMap = new HashMap<>();

        State initialState = new State(3, 3, 0, 0, true);

        stack.push(initialState);

        visited.add(initialState);

        while (!stack.isEmpty()) {

            State currentState = stack.pop();

            if (currentState.isGoal()) {

                printSolution(parentMap, currentState);

                return;

            }

            List<State> nextPossibleStates = getNextStates(currentState);

            for (State nextState : nextPossibleStates) {

                if (!visited.contains(nextState)) {

                    stack.push(nextState);

                    visited.add(nextState);

                    parentMap.put(nextState, currentState);

                }

            }

        }

    }

    // Method to get next possible states from current state

    public static List<State> getNextStates(State currentState) {

        List<State> nextStates = new ArrayList<>();

        int[] missionaries = {1, 0, 2, 0, 1};

        int[] cannibals = {0, 1, 0, 2, 1};

        for (int i = 0; i < missionaries.length; i++) {

            int deltaML = missionaries[i];

            int deltaCL = cannibals[i];

            int deltaMR = -deltaML;

            int deltaCR = -deltaCL;

            if (currentState.boatOnLeft) {

                State nextState = new State(currentState.missionariesLeft - deltaML, currentState.cannibalsLeft - deltaCL,

                        currentState.missionariesRight - deltaMR, currentState.cannibalsRight - deltaCR, false);

                if (nextState.isValid()) {

                    nextStates.add(nextState);

                }

            } else {

                State nextState = new State(currentState.missionariesLeft + deltaML, currentState.cannibalsLeft + deltaCL,

                        currentState.missionariesRight + deltaMR, currentState.cannibalsRight + deltaCR, true);

                if (nextState.isValid()) {

                    nextStates.add(nextState);

                }

            }

        }

        return nextStates;

    }

    // Method to print solution

    public static void printSolution(Map<State, State> parentMap, State goalState) {

        List<State> solution = new ArrayList<>();

        State currentState = goalState;

        while (currentState != null) {

            solution.add(currentState);

            currentState = parentMap.get(currentState);

        }

        Collections.reverse(solution);

        for (State state : solution) {

            System.out.println(state.missionariesLeft + "M " + state.cannibalsLeft + "C | BOAT | " +

                    state.missionariesRight + "M " + state.cannibalsRight + "C");

        }

    }

    public static void main(String[] args) {

        System.out.println("Breadth First Search Solution:");

        bfs();

        System.out.println("\nDepth First Search Solution:");

        dfs();

    }

}

**Missionaries And Cannibals using DLS(Depth Limiting Search)**

import java.util.ArrayList;

import java.util.List;

// based on the depth-limited search algorithm present on the 3o Edition of the

// "Artificial Intelligence A Modern Approach".

enum Position {RIGHT, LEFT}

class State {

    private int cannibalLeft;

    private int missionaryLeft;

    private int cannibalRight;

    private int missionaryRight;

    private Position boat;

    private State parentState;

    public State(int cannibalLeft, int missionaryLeft, Position boat,

            int cannibalRight, int missionaryRight) {

        this.cannibalLeft = cannibalLeft;

        this.missionaryLeft = missionaryLeft;

        this.boat = boat;

        this.cannibalRight = cannibalRight;

        this.missionaryRight = missionaryRight;

    }

    public boolean isGoal() {

        return cannibalLeft == 0 && missionaryLeft == 0;

    }

    public boolean isValid() {

        if (missionaryLeft >= 0 && missionaryRight >= 0 && cannibalLeft >= 0 && cannibalRight >= 0

                   && (missionaryLeft == 0 || missionaryLeft >= cannibalLeft)

                   && (missionaryRight == 0 || missionaryRight >= cannibalRight)) {

            return true;

        }

        return false;

    }

    public List<State> generateSuccessors() {

        List<State> successors = new ArrayList<State>();

        if (boat == Position.LEFT) {

            testAndAdd(successors, new State(cannibalLeft, missionaryLeft - 2, Position.RIGHT,

                    cannibalRight, missionaryRight + 2)); // Two missionaries cross left to right.

            testAndAdd(successors, new State(cannibalLeft - 2, missionaryLeft, Position.RIGHT,

                    cannibalRight + 2, missionaryRight)); // Two cannibals cross left to right.

            testAndAdd(successors, new State(cannibalLeft - 1, missionaryLeft - 1, Position.RIGHT,

                    cannibalRight + 1, missionaryRight + 1)); // One missionary and one cannibal cross left to right.

            testAndAdd(successors, new State(cannibalLeft, missionaryLeft - 1, Position.RIGHT,

                    cannibalRight, missionaryRight + 1)); // One missionary crosses left to right.

            testAndAdd(successors, new State(cannibalLeft - 1, missionaryLeft, Position.RIGHT,

                    cannibalRight + 1, missionaryRight)); // One cannibal crosses left to right.

        } else {

            testAndAdd(successors, new State(cannibalLeft, missionaryLeft + 2, Position.LEFT,

                    cannibalRight, missionaryRight - 2)); // Two missionaries cross right to left.

            testAndAdd(successors, new State(cannibalLeft + 2, missionaryLeft, Position.LEFT,

                    cannibalRight - 2, missionaryRight)); // Two cannibals cross right to left.

            testAndAdd(successors, new State(cannibalLeft + 1, missionaryLeft + 1, Position.LEFT,

                    cannibalRight - 1, missionaryRight - 1)); // One missionary and one cannibal cross right to left.

            testAndAdd(successors, new State(cannibalLeft, missionaryLeft + 1, Position.LEFT,

                    cannibalRight, missionaryRight - 1)); // One missionary crosses right to left.

            testAndAdd(successors, new State(cannibalLeft + 1, missionaryLeft, Position.LEFT,

                    cannibalRight - 1, missionaryRight)); // One cannibal crosses right to left.

        }

        return successors;

    }

    private void testAndAdd(List<State> successors, State newState) {

        if (newState.isValid()) {

            newState.setParentState(this);

            successors.add(newState);

        }

    }

    public int getCannibalLeft() {

        return cannibalLeft;

    }

    public void setCannibalLeft(int cannibalLeft) {

        this.cannibalLeft = cannibalLeft;

    }

    public int getMissionaryLeft() {

        return missionaryLeft;

    }

    public void setMissionaryLeft(int missionaryLeft) {

        this.missionaryLeft = missionaryLeft;

    }

    public int getCannibalRight() {

        return cannibalRight;

    }

    public void setCannibalRight(int cannibalRight) {

        this.cannibalRight = cannibalRight;

    }

    public int getMissionaryRight() {

        return missionaryRight;

    }

    public void setMissionaryRight(int missionaryRight) {

        this.missionaryRight = missionaryRight;

    }

    public void goToLeft() {

        boat = Position.LEFT;

    }

    public void goToRight() {

        boat = Position.RIGHT;

    }

    public boolean isOnLeft() {

        return boat == Position.LEFT;

    }

    public boolean isOnRigth() {

        return boat == Position.RIGHT;

    }

    public State getParentState() {

        return parentState;

    }

    public void setParentState(State parentState) {

        this.parentState = parentState;

    }

    @Override

    public String toString() {

        if (boat == Position.LEFT) {

            return "(" + cannibalLeft + "," + missionaryLeft + ",L,"

                    + cannibalRight + "," + missionaryRight + ")";

        } else {

            return "(" + cannibalLeft + "," + missionaryLeft + ",R,"

                    + cannibalRight + "," + missionaryRight + ")";

        }

     }

    @Override

    public boolean equals(Object obj) {

        if (!(obj instanceof State)) {

            return false;

        }

        State s = (State) obj;

        return (s.cannibalLeft == cannibalLeft && s.missionaryLeft == missionaryLeft

                && s.boat == boat && s.cannibalRight == cannibalRight

                && s.missionaryRight == missionaryRight);

    }

}

class DepthLimitedSearch {

    public State exec(State initialState) {

        int limit = 20;

        return recursiveDLS(initialState, limit);

    }

    private State recursiveDLS(State state, int limit) {

        if (state.isGoal()) {

            return state;

        } else if (limit == 0) {

            return null;

        } else {

            List<State> successors = state.generateSuccessors();

            for (State child : successors) {

                State result = recursiveDLS(child, limit - 1);

                if (null != result) {

                    return result;

                }

            }

            return null;

        }

    }

}

public class depthlimit {

    public static void main(String[] args) {

        State initialState = new State (3, 3, Position.LEFT, 0, 0);

        executeDLS(initialState);

}

    private static void executeDLS(State initialState) {

        DepthLimitedSearch search = new DepthLimitedSearch();

        State solution = search.exec(initialState);

        printSolution(solution);

    }

    private static void printSolution(State solution) {

        if (null == solution) {

            System.out.print("\nNo solution found.");

        } else {

            System.out.println("\nSolution (cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight): ");

            List<State> path = new ArrayList<State>();

            State state = solution;

            while(null!=state) {

                path.add(state);

                state = state.getParentState();

            }

            int depth = path.size() - 1;

            for (int i = depth; i >= 0; i--) {

                state = path.get(i);

                if (state.isGoal()) {

                    System.out.print(state.toString());

                } else {

                    System.out.print(state.toString() + " -> ");

                }

            }

            System.out.println("\nDepth: " + depth);

        }

    }

}

**Missionaries And Cannibals using IDFS(Iterative Deepening First Search)**

//import java.util.Queue;

import java.util.ArrayList;

import java.util.Collections;

public class idfs {

    State SolutionState;

    int count = 0;

    ArrayList<State> actions = new ArrayList<>(); //List of the 5 possible moves

    public idfs() {

        actions.add(new State(1,0,1));

        actions.add(new State(2,0,1));

        actions.add(new State(0,1,1));

        actions.add(new State(0,2,1));

        actions.add(new State(1,1,1));

    }

    //returns an arrayList of the Solution Path or null if no solution is found

    public ArrayList<State> findSolution(State startState) {

       //State result = RecursiveDLS(startState,20,1);

       State result = IDS(startState, 20);

        if (result  != null) {

            return getSolutionPath(result);

        }

        else

            return null;

    }

    public State IDS(State root, int cutoff) {

        if (root.isSolution()) return root;

        State result = null;

        for (int i = 1; result == null; i++) {

            result =  RecursiveDLS(root,i, 0);

            if (result != null && result.isSolution())

                return result;

        }

        return result;

    }

    public State RecursiveDLS(State cur,int limit, int depth) {

    //  System.out.print("count:"+count+" curLim: "+ depth+"  ");

    //  System.out.println(cur.toString());

        if (depth < limit) {

            State result, child;

            for (State a: actions) {

                child = cur.getChild(a);

                if (child != null && child.isValid()) {

                    count++;

                    if (child.isSolution()) return child;

                    if ((result = RecursiveDLS(child,limit, depth+1)) != null)

                        return result;

                }

            }

        }

        return null;

    }

    //Traces the solution's path back to the initial state and returns an

    //ArrayList of all the states that lead to the solution

    public ArrayList<State> getSolutionPath(State solutionState) {

        ArrayList<State> solutionPath = new ArrayList<>();

        solutionPath.add(solutionState);

        State par = solutionState.parent;

        while (par != null) {

            solutionPath.add(par);

            par = par.parent;

        }

        Collections.reverse(solutionPath);

        return solutionPath;

    }

    public static void main(String[] args) {

        ArrayList<State> solution = new idfs().findSolution(new State(3,3,1));

        if (solution == null) System.out.println("No Solution Found");

        else {

            System.out.println("\n\*\*Solution was Found\*\*");

            System.out.println("           < START >            < GOAL >");

            System.out.println("          | M, C, B |  )   )  | M, C, B |");

            State r = solution.remove(0);

            System.out.println("\*ROOT:    "+ r.toString()+" (   (   "+ r.inverse().toString() );

            solution.stream().forEach((s) -> {

            String river = (s.depth()%2 == 0) ? " (   (   " : "  )   )  ";

            String sp = (s.depth() < 10) ? "   " : "  ";

            System.out.println("depth:"+s.depth() +sp+ s.toString()+river+ s.inverse().toString() );

        });

        }

    }

   /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    \* This class represents the different states of the Cannibals vs.   \*

    \* Missionaries Problem. The 'parent' variable is used to trace the  \*

    \* path of the solution back to the root, and the 'prevAct' variable \*

    \* is used keep the search tree unidirectional, or to stop a state   \*

    \* from making its child state the same as its parent.               \*

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    private static class State {

        public int numMis;

        public int numCan;

        public int numBoats;

        public int depth;

        public State parent;

        public State prevAct;

        //used for the starting state and action states

        public State(int m, int c, int b) {

            numMis = m;

            numCan = c;

            numBoats = b;

            parent = null;

            depth = 0;

            prevAct = null;

        }

        //used for descendent states

        public State(int m, int c, int b, State p, int d) {

            numMis = m;

            numCan = c;

            numBoats = b;

            parent = p;

            depth = d;

            prevAct = null;

        }

        //checks if a State is valid

        public boolean isValid() {

            if ( numMis > 3 || numMis < 0 ||

                 numCan > 3 || numCan < 0 ||

                 (numMis < numCan && numMis > 0) ||

                 (3-numMis < 3-numCan && 3-numMis > 0) )

                 return false;

            else

                return true;

        }

        //returns a child state based on an action state

        public State getChild(State a) {

            if (prevAct != null && prevAct == a) return null;

            State newState;

            if(numBoats == 1)

                newState = new State(numMis - a.numMis,

                                     numCan - a.numCan,

                                     numBoats - a.numBoats,

                                     this, this.depth + 1);

            else

                newState = new State(numMis + a.numMis,

                                     numCan + a.numCan,

                                     numBoats + a.numBoats,

                                     this, this.depth + 1);

            newState.prevAct = a;

            return newState;

        }

        public boolean isSolution() { return numMis == 0 && numCan == 0; }

        public State inverse() {

            return new State(3 - numMis, 3-numCan, 1 - numBoats, this.parent, this.depth);

        }

        public int depth() {

            return this.depth;

        }

        @Override

        public String toString() {

            if (parent != null)

               // return "depth:"+depth+"  <"+numMis+", "+numCan+", "+numBoats+">";

            return "| "+numMis+", "+numCan+", "+numBoats+" |";

          else

            //return "root:    <"+numMis+", "+numCan+", "+ numBoats+">";

              return "| "+numMis+", "+numCan+", "+ numBoats+" |";

        }

    }

}

**Missionaries and Cannibals using BFS(Best First Search)**

import java.util.\*;

class State {

    int missionariesLeft;

    int cannibalsLeft;

    int boat;

    int missionariesRight;

    int cannibalsRight;

    State parentState;

    public State(int missionariesLeft, int cannibalsLeft, int boat,

                 int missionariesRight, int cannibalsRight) {

        this.missionariesLeft = missionariesLeft;

        this.cannibalsLeft = cannibalsLeft;

        this.boat = boat;

        this.missionariesRight = missionariesRight;

        this.cannibalsRight = cannibalsRight;

    }

    public boolean isGoal() {

        return missionariesLeft == 0 && cannibalsLeft == 0;

    }

    public boolean isValid() {

        if (missionariesLeft >= 0 && cannibalsLeft >= 0 &&

                missionariesRight >= 0 && cannibalsRight >= 0 &&

                (missionariesLeft == 0 || missionariesLeft >= cannibalsLeft) &&

                (missionariesRight == 0 || missionariesRight >= cannibalsRight)) {

            return true;

        }

        return false;

    }

    public List<State> generateSuccessors() {

        List<State> successors = new ArrayList<>();

        if (boat == 0) {

            // Boat is on the left side

            for (int i = 0; i <= 2; i++) {

                for (int j = 0; j <= 2; j++) {

                    if (i + j >= 1 && i + j <= 2) {

                        State newState = new State(missionariesLeft - i, cannibalsLeft - j, 1,

                                missionariesRight + i, cannibalsRight + j);

                        if (newState.isValid()) {

                            newState.parentState = this;

                            successors.add(newState);

                        }

                    }

                }

            }

        } else {

            // Boat is on the right side

            for (int i = 0; i <= 2; i++) {

                for (int j = 0; j <= 2; j++) {

                    if (i + j >= 1 && i + j <= 2) {

                        State newState = new State(missionariesLeft + i, cannibalsLeft + j, 0,

                                missionariesRight - i, cannibalsRight - j);

                        if (newState.isValid()) {

                            newState.parentState = this;

                            successors.add(newState);

                        }

                    }

                }

            }

        }

        return successors;

    }

    public int getHeuristic() {

        // Using a simple heuristic: Number of missionaries and cannibals left on the left side

        return missionariesLeft + cannibalsLeft;

    }

}

class BestFirstSearch {

    public static void main(String[] args) {

        State initialState = new State(3, 3, 0, 0, 0);

        System.out.println("Initial State:");

        printSolution(initialState);

        State solution = solve(initialState);

        System.out.println("\nSolution:");

        printSolution(solution);

    }

    public static State solve(State initialState) {

        PriorityQueue<State> openSet = new PriorityQueue<>(Comparator.comparingInt(s -> s.getHeuristic()));

        Set<State> closedSet = new HashSet<>();

        openSet.add(initialState);

        while (!openSet.isEmpty()) {

            State currentState = openSet.poll();

            if (currentState.isGoal()) {

                return currentState;

            }

            closedSet.add(currentState);

            for (State successor : currentState.generateSuccessors()) {

                if (!closedSet.contains(successor)) {

                    openSet.add(successor);

                }

            }

        }

        return null;

    }

    public static void printSolution(State solution) {

        if (solution == null) {

            System.out.println("No solution found.");

            return;

        }

        List<State> path = new ArrayList<>();

        State currentState = solution;

        while (currentState != null) {

            path.add(currentState);

            currentState = currentState.parentState;

        }

        for (int i = path.size() - 1; i >= 0; i--) {

            currentState = path.get(i);

            System.out.println(String.format("Left: %d missionaries, %d cannibals | Right: %d missionaries, %d cannibals | Boat: %s",

                    currentState.missionariesLeft, currentState.cannibalsLeft,

                    currentState.missionariesRight, currentState.cannibalsRight,

                    currentState.boat == 0 ? "Left" : "Right"));

        }

    }

}

**Missionaries and Cannibals using A\***

import java.util.\*;

class State {

    int missionariesLeft;

    int cannibalsLeft;

    int boat;

    int missionariesRight;

    int cannibalsRight;

    State parentState;

    public State(int missionariesLeft, int cannibalsLeft, int boat,

                 int missionariesRight, int cannibalsRight) {

        this.missionariesLeft = missionariesLeft;

        this.cannibalsLeft = cannibalsLeft;

        this.boat = boat;

        this.missionariesRight = missionariesRight;

        this.cannibalsRight = cannibalsRight;

    }

    public boolean isGoal() {

        return missionariesLeft == 0 && cannibalsLeft == 0;

    }

    public boolean isValid() {

        if (missionariesLeft >= 0 && cannibalsLeft >= 0 &&

                missionariesRight >= 0 && cannibalsRight >= 0 &&

                (missionariesLeft == 0 || missionariesLeft >= cannibalsLeft) &&

                (missionariesRight == 0 || missionariesRight >= cannibalsRight)) {

            return true;

        }

        return false;

    }

    public List<State> generateSuccessors() {

        List<State> successors = new ArrayList<>();

        if (boat == 0) {

            // Boat is on the left side

            for (int i = 0; i <= 2; i++) {

                for (int j = 0; j <= 2; j++) {

                    if (i + j >= 1 && i + j <= 2) {

                        State newState = new State(missionariesLeft - i, cannibalsLeft - j, 1,

                                missionariesRight + i, cannibalsRight + j);

                        if (newState.isValid()) {

                            newState.parentState = this;

                            successors.add(newState);

                        }

                    }

                }

            }

        } else {

            // Boat is on the right side

            for (int i = 0; i <= 2; i++) {

                for (int j = 0; j <= 2; j++) {

                    if (i + j >= 1 && i + j <= 2) {

                        State newState = new State(missionariesLeft + i, cannibalsLeft + j, 0,

                                missionariesRight - i, cannibalsRight - j);

                        if (newState.isValid()) {

                            newState.parentState = this;

                            successors.add(newState);

                        }

                    }

                }

            }

        }

        return successors;

    }

    public int getHeuristic() {

        // Using a simple heuristic: Number of missionaries and cannibals left on the left side

        return missionariesLeft + cannibalsLeft;

    }

}

class AStarSearch {

    public static void main(String[] args) {

        State initialState = new State(3, 3, 0, 0, 0);

        System.out.println("Initial State:");

        printSolution(initialState);

        State solution = solve(initialState);

        System.out.println("\nSolution:");

        printSolution(solution);

    }

    public static State solve(State initialState) {

        PriorityQueue<State> openSet = new PriorityQueue<>(Comparator.comparingInt(s -> s.getHeuristic()));

        Set<State> closedSet = new HashSet<>();

        openSet.add(initialState);

        while (!openSet.isEmpty()) {

            State currentState = openSet.poll();

            if (currentState.isGoal()) {

                return currentState;

            }

            closedSet.add(currentState);

            for (State successor : currentState.generateSuccessors()) {

                if (!closedSet.contains(successor)) {

                    openSet.add(successor);

                }

            }

        }

        return null;

    }

    public static void printSolution(State solution) {

        if (solution == null) {

            System.out.println("No solution found.");

            return;

        }

        List<State> path = new ArrayList<>();

        State currentState = solution;

        while (currentState != null) {

            path.add(currentState);

            currentState = currentState.parentState;

        }

        for (int i = path.size() - 1; i >= 0; i--) {

            currentState = path.get(i);

            System.out.println(String.format("Left: %d missionaries, %d cannibals | Right: %d missionaries, %d cannibals | Boat: %s",

                    currentState.missionariesLeft, currentState.cannibalsLeft,

                    currentState.missionariesRight, currentState.cannibalsRight,

                    currentState.boat == 0 ? "Left" : "Right"));

        }

    }

}

**Missionaries and Cannibals using AO\***

import java.util.\*;

class State {

    int missionaries;

    int cannibals;

    boolean boat;

    public State(int missionaries, int cannibals, boolean boat) {

        this.missionaries = missionaries;

        this.cannibals = cannibals;

        this.boat = boat;

    }

    @Override

    public boolean equals(Object o) {

        if (this == o) return true;

        if (o == null || getClass() != o.getClass()) return false;

        State state = (State) o;

        return missionaries == state.missionaries &&

                cannibals == state.cannibals &&

                boat == state.boat;

    }

    @Override

    public int hashCode() {

        return Objects.hash(missionaries, cannibals, boat);

    }

}

class Node {

    State state;

    Node parent;

    int cost;

    int heuristic;

    public Node(State state, Node parent, int cost, int heuristic) {

        this.state = state;

        this.parent = parent;

        this.cost = cost;

        this.heuristic = heuristic;

    }

}

public class aostar {

    public static List<State> getSuccessors(State state) {

        List<State> successors = new ArrayList<>();

        if (state.boat) {

            for (int m = 0; m <= 2; m++) {

                for (int c = 0; c <= 2; c++) {

                    if (m + c <= 2 && m + c > 0 && (m == 0 || m >= c)) {

                        successors.add(new State(state.missionaries - m, state.cannibals - c, !state.boat));

                    }

                }

            }

        } else {

            for (int m = 0; m <= 2; m++) {

                for (int c = 0; c <= 2; c++) {

                    if (m + c <= 2 && m + c > 0 && (m == 0 || m >= c)) {

                        successors.add(new State(state.missionaries + m, state.cannibals + c, !state.boat));

                    }

                }

            }

        }

        return successors;

    }

    public static int heuristic(State state) {

        // In this case, a simple heuristic could be the total number of missionaries and cannibals remaining to move

        return state.missionaries + state.cannibals;

    }

    public static List<State> aoStar(State initialState) {

        PriorityQueue<Node> open = new PriorityQueue<>(Comparator.comparingInt(a -> a.cost + a.heuristic));

        Map<State, Integer> costMap = new HashMap<>();

        Map<State, Node> parentMap = new HashMap<>();

        open.add(new Node(initialState, null, 0, heuristic(initialState)));

        costMap.put(initialState, 0);

        while (!open.isEmpty()) {

            Node currentNode = open.poll();

            State currentState = currentNode.state;

            if (currentState.missionaries == 0 && currentState.cannibals == 0 && !currentState.boat) {

                List<State> path = new ArrayList<>();

                while (currentNode != null) {

                    path.add(currentNode.state);

                    currentNode = currentNode.parent;

                }

                Collections.reverse(path);

                return path;

            }

            for (State successor : getSuccessors(currentState)) {

                int newCost = currentNode.cost + 1;

                if (!costMap.containsKey(successor) || newCost < costMap.get(successor)) {

                    int heuristic = heuristic(successor);

                    open.add(new Node(successor, currentNode, newCost, heuristic));

                    costMap.put(successor, newCost);

                    parentMap.put(successor, currentNode);

                }

            }

        }

        return null; // No solution found

    }

    public static void main(String[] args) {

        State initialState = new State(3, 3, true);

        List<State> solution = aoStar(initialState);

        if (solution != null) {

            for (State state : solution) {

                System.out.println("Missionaries: " + state.missionaries + ", Cannibals: " + state.cannibals + ", Boat: " + (state.boat ? "Left" : "Right"));

            }

        } else {

            System.out.println("No solution found!");

        }

    }

}

**Missionaries and Cannibals using hill climbing**

import java.util.\*;

public class MissionariesAndCannibals {

    // Representation of a state

    static class State {

        int missionaries;

        int cannibals;

        boolean boat;

        State(int missionaries, int cannibals, boolean boat) {

            this.missionaries = missionaries;

            this.cannibals = cannibals;

            this.boat = boat;

        }

        @Override

        public boolean equals(Object obj) {

            if (obj == this) return true;

            if (!(obj instanceof State)) return false;

            State other = (State) obj;

            return missionaries == other.missionaries && cannibals == other.cannibals && boat == other.boat;

        }

        @Override

        public int hashCode() {

            return Objects.hash(missionaries, cannibals, boat);

        }

    }

    // Hill Climbing algorithm with an informed strategy

    static List<State> hillClimbing() {

        List<State> path = new ArrayList<>();

        State currentState = new State(3, 3, true);

        path.add(currentState);

        while (!goalState(currentState)) {

            List<State> neighbors = getNeighbors(currentState);

            State nextState = null;

            int minHeuristic = Integer.MAX\_VALUE;

            for (State neighbor : neighbors) {

                int heuristic = heuristic(neighbor);

                if (heuristic < minHeuristic) {

                    minHeuristic = heuristic;

                    nextState = neighbor;

                }

            }

            if (nextState == null)

                break;

            path.add(nextState);

            currentState = nextState;

        }

        return path;

    }

    // Heuristic function: number of missionaries and cannibals on the wrong side

    static int heuristic(State state) {

        int onWrongSide = state.missionaries + state.cannibals;

        if (state.missionaries != 0 && state.missionaries < state.cannibals)

            onWrongSide += state.cannibals - state.missionaries;

        return onWrongSide;

    }

    // Check if a state is the goal state

    static boolean goalState(State state) {

        return state.missionaries == 0 && state.cannibals == 0 && !state.boat;

    }

    // Get possible neighbors (successor states)

    static List<State> getNeighbors(State state) {

        List<State> neighbors = new ArrayList<>();

        int m = state.missionaries;

        int c = state.cannibals;

        if (state.boat) {

            for (int i = 0; i <= 2; i++) {

                for (int j = 0; j <= 2; j++) {

                    if (i + j >= 1 && i + j <= 2 && m - i >= 0 && c - j >= 0 && (m - i == 0 || m - i >= c - j))

                        neighbors.add(new State(m - i, c - j, false));

                }

            }

        } else {

            for (int i = 0; i <= 2; i++) {

                for (int j = 0; j <= 2; j++) {

                    if (i + j >= 1 && i + j <= 2 && m + i >= 0 && c + j >= 0 && (m + i == 0 || m + i >= c + j))

                        neighbors.add(new State(m + i, c + j, true));

                }

            }

        }

        return neighbors;

    }

    public static void main(String[] args) {

        List<State> path = hillClimbing();

        System.out.println("Solution path:");

        for (State state : path) {

            System.out.println(state.missionaries + " missionaries, " + state.cannibals + " cannibals, boat: " + (state.boat ? "left" : "right"));

        }

    }

}

**CSP(Constraint Satisfaction Problem) using for graph coloring**

public class csp\_graph\_coloring {

    final int V = 4;

    int color[];

    /\* A utility function to check if the current

       color assignment is safe for vertex v \*/

    boolean isSafe(int v, int graph[][], int color[],

                   int c)

    {

        for (int i = 0; i < V; i++)

            if (graph[v][i] == 1 && c == color[i])

                return false;

        return true;

    }

    /\* A recursive utility function to solve m

       coloring  problem \*/

    boolean graphColoringUtil(int graph[][], int m,

                              int color[], int v)

    {

        /\* base case: If all vertices are assigned

           a color then return true \*/

        if (v == V)

            return true;

        /\* Consider this vertex v and try different

           colors \*/

        for (int c = 1; c <= m; c++)

        {

            /\* Check if assignment of color c to v

               is fine\*/

            if (isSafe(v, graph, color, c))

            {

                color[v] = c;

                /\* recur to assign colors to rest

                   of the vertices \*/

                if (graphColoringUtil(graph, m,

                            color, v + 1))

                    return true;

                /\* If assigning color c doesn't lead

                   to a solution then remove it \*/

                color[v] = 0;

            }

        }

        /\* If no color can be assigned to this vertex

           then return false \*/

        return false;

    }

    /\* This function solves the m Coloring problem using

       Backtracking. It mainly uses graphColoringUtil()

       to solve the problem. It returns false if the m

       colors cannot be assigned, otherwise return true

       and  prints assignments of colors to all vertices.

       .\*/

    boolean graphColoring(int graph[][], int m)

    {

        // Initialize all color values as 0. This

        // initialization is needed correct functioning

        // of isSafe()

        color = new int[V];

        for (int i = 0; i < V; i++)

            color[i] = 0;

        // Call graphColoringUtil() for vertex 0

        if (!graphColoringUtil(graph, m, color, 0))

        {

            System.out.println("Solution does not exist");

            return false;

        }

        // Print the solution

        printSolution(color);

        return true;

    }

    /\* A utility function to print solution \*/

    void printSolution(int color[])

    {

        System.out.println("Solution Exists: Following" +

                           " are the assigned colors");

        for (int i = 0; i < V; i++)

            System.out.print(" " + color[i] + " ");

        System.out.println();

    }

    // driver program to test above function

    public static void main(String args[])

    {

        csp\_graph\_coloring Coloring = new csp\_graph\_coloring ();

        /\* Create following graph and test whether it is

           3 colorable

          (1)---(2)

           | \   |

           |  \  |

           |   \ |

          (3)---(4)

        \*/

        int graph[][] = {{0, 1, 1, 1},

            {1, 0, 0, 1},

            {1, 0, 0, 1},

            {1, 1, 1, 0},

        };

        int m = 3; // Number of colors

        Coloring.graphColoring(graph, m);

    }

}

**Prolog-Predicate Logic**

**% Define facts and rules for a family tree with Indian names.**

**% Facts: Define parent-child relationships.**

**parent(ram, priya).**

**parent(ram, anil).**

**parent(priya, amita).**

**parent(anil, anjali).**

**parent(shyam, ram).**

**parent(shyam, priya).**

**% Rules: Define sibling and grandparent relationships.**

**sibling(X, Y) :-**

**parent(Z, X),**

**parent(Z, Y),**

**X \= Y.**

**grandparent(X, Y) :-**

**parent(X, Z),**

**parent(Z, Y).**

**% Sample queries:**

**% Check if Ram is the parent of Priya.**

**% ?- parent(ram, priya).**

**% Output: true**

**% Find siblings of Anil.**

**% ?- sibling(anil, Sibling).**

**% Output: Sibling = priya**

**% Find the grandparents of Amita.**

**% ?- grandparent(Grandparent, amita).**

**% Output: Grandparent = ram**

**Prolog-Expert system**

**% Define dynamic predicates for balance and transactions**

**:- dynamic balance/1.**

**:- dynamic income/2.**

**:- dynamic expense/2.**

**% Initialize balance**

**initialize\_balance(InitialBalance) :-**

**assertz(balance(InitialBalance)).**

**% Add income**

**add\_income(Description, Amount) :-**

**balance(Balance),**

**NewBalance is Balance + Amount,**

**retract(balance(Balance)),**

**assertz(balance(NewBalance)),**

**assertz(income(Description, Amount)),**

**write('Income added. New balance: $'), write(NewBalance), nl.**

**% Add expense**

**add\_expense(Description, Amount) :-**

**balance(Balance),**

**NewBalance is Balance - Amount,**

**retract(balance(Balance)),**

**assertz(balance(NewBalance)),**

**assertz(expense(Description, Amount)),**

**write('Expense added. New balance: $'), write(NewBalance), nl.**

**% View balance**

**view\_balance :-**

**balance(Balance),**

**write('Current balance: $'), write(Balance), nl.**

**% View transaction history**

**view\_transactions :-**

**write('Transaction History:'), nl,**

**findall(Income, income(IncomeDescription, Income), IncomeList),**

**findall(Expense, expense(ExpenseDescription, Expense), ExpenseList),**

**print\_transactions(IncomeList, 'Income'),**

**print\_transactions(ExpenseList, 'Expense').**

**print\_transactions([], \_).**

**print\_transactions([H|T], Category) :-**

**tab(2),**

**write(Category), write(': $'), write(H), nl,**

**print\_transactions(T, Category).**

**% Entry point for finance management**

**start\_finance\_management :-**

**write('Welcome to Finance Management System.'), nl,**

**write('Enter the initial balance: $'),**

**read(InitialBalance),**

**initialize\_balance(InitialBalance),**

**repeat,**

**write('Choose an option:'), nl,**

**write('1. Add income'), nl,**

**write('2. Add expense'), nl,**

**write('3. View balance'), nl,**

**write('4. View transaction history'), nl,**

**write('5. Exit'), nl,**

**read(Option),**

**process\_option(Option),**

**Option = 5,**

**!.**

**process\_option(1) :-**

**write('Enter income description: '),**

**read(Description),**

**write('Enter income amount: $'),**

**read(IncomeAmount),**

**add\_income(Description, IncomeAmount).**

**process\_option(2) :-**

**write('Enter expense description: '),**

**read(Description),**

**write('Enter expense amount: $'),**

**read(ExpenseAmount),**

**add\_expense(Description, ExpenseAmount).**

**process\_option(3) :-**

**view\_balance.**

**process\_option(4) :-**

**view\_transactions.**

**process\_option(5) :-**

**write('Exiting Finance Management System.'), nl.**

**process\_option(\_) :-**

**write('Invalid option. Please choose a valid option.'), nl.**

**% Run the finance management system**

**:- start\_finance\_managemen**