**Expt1:**

Algorithm:

1. Initialize the game board with empty spaces.
2. Set the current player to "X".
3. Loop until there is a winner or the game ends in a tie:

a. Print the current state of the game board.

b. Get the current player's move (row and column) from the user.

c. If the chosen space is already occupied, print an error message and ask for another move.

d. If the chosen space is empty, fill it with the current player's symbol ("X" or "O").

e. Check if the current player has won the game. If so, print a message declaring the winner and end the game.

f. If the game board is full and no one has won, print a message declaring a tie and end the game.

g. Switch the current player to the other symbol.

Result:

Thus, the given program is executed and verified successfully.

**Expt2:**

Algorithm:

1. Define the problem: a. Determine the size of the chess board (n x n). b. Define the rules of the game (i.e., how the queen moves and how they can be placed). c. Identify the goal of the game (i.e., to place n queens on the board such that no two queens threaten each other).
2. Develop a representation for the game board: a. Use an n x n array to represent the chess board. b. Represent the presence of a queen with the value "1" and the absence of a queen with the value "0".
3. Develop a method for checking the validity of a given board state: a. For each queen on the board, check if any other queen threatens it. b. If no queens threaten each other, the board state is valid.
4. Develop a method for generating possible next moves: a. Iterate through each column of the board. b. For each column, try placing a queen in each row. c. If the resulting board state is valid, add it to the list of possible moves.
5. Develop a search algorithm to find a solution: a. Start with an initial board state (e.g., all queens placed in the first row). b. Generate possible next moves. c. If a valid solution is found, stop. d. If no valid solution is found, choose the best next move based on a heuristic (e.g., number of queens placed, number of conflicts). e. Repeat steps b-d until a valid solution is found.

Result:

Thus, the given program is executed and verified successfully.

**Expt3:**

Algorithm:

1. Define the problem domain: In this case, we need to place n queens on an n x n chessboard such that no two queens can attack each other.
2. Define the variables: The variables in this problem are the positions of the queens on the board. We can represent each variable as a tuple (i, j), where i represents the row and j represents the column.
3. Define the domains: The domain of each variable is the set of possible values it can take. In this case, each variable can take on values from 1 to n.
4. Define the constraints: The constraints in this problem are that no two queens can attack each other. There are three types of constraints to consider: row constraints (no two queens can be in the same row), column constraints (no two queens can be in the same column), and diagonal constraints (no two queens can be on the same diagonal).
5. Implement a backtracking algorithm: Start with an empty board and recursively try to add queens to the board, making sure to check that the current configuration satisfies all constraints. If a configuration violates a constraint, backtrack to the previous configuration and try a different value for the current variable.

**Expt4:**

Algorithm for DFS (Depth-First Search):

1. Create a Stack to keep track of visited nodes.
2. Initialize the stack with the starting node.
3. Create a visited set to keep track of already visited nodes.
4. While the stack is not empty: a. Pop the top node from the stack b. If the node has not been visited yet: i. Mark, the node as visited ii. Process the node (e.g., print its value, add its neighbours to the stack) iii. Push all unvisited neighbours of the node onto the stack.
5. If there are no more nodes to process, the algorithm terminates.

Algorithm for BFS (Breadth-First Search):

1. Create a Queue to keep track of visited nodes.
2. Enqueue the starting node into the Queue
3. Create a visited set to keep track of already visited nodes.
4. While the Queue is not empty: a. Dequeue the front node from the Queue b. If the node has not been visited yet:
   1. Mark, the node as visited,
   2. Process the node (e.g. print its value, add its neighbours to the Queue),
   3. Enqueue all unvisited neighbours of the node onto the Queue
5. If there are no more nodes to process, the algorithm terminates.

**Expt5:**

Algorithm for Best First Search:

1. Create an empty priority queue and add the initial state to it with a priority based on a heuristic function.
2. While the priority queue is not empty:
   * Remove the state with the highest priority from the queue.
   * If it is the goal state, return the solution.
   * Expand the state by generating all possible successor states and add them to the priority queue with priorities based on the heuristic function.
3. If the priority queue becomes empty before finding a solution, return failure.

Algorithm for A star:

1. Create an empty priority queue and add the initial state to it with a priority based on a cost function and a heuristic function.
2. While the priority queue is not empty:
   * Remove the state with the highest priority from the queue.
   * If it is the goal state, return the solution.
   * Expand the state by generating all possible successor states and add them to the priority queue with priorities based on the cost function and the heuristic function.
3. If the priority queue becomes empty before finding a solution, return failure.

**Expt6:**

Algorithm:

1. Define the game state: Start by defining the current state of the game, including the positions of all pieces, the score, and any other relevant information.
2. Generate all possible future states: Consider all possible moves that the agent and opponent can make from the current state and generate a new state for each.
3. Evaluate each state: For each generated state, evaluate how advantageous it is for the agent. This may involve calculating a score based on factors such as the number of pieces on the board, their positions, and the potential for future moves.
4. Choose the best move: Choose the move that leads to the state with the highest minimum value. To do this, assume that the opponent will always choose the move that leads to the state with the lowest maximum value.

**Expt7:**

Algorithm:

1. Define the problem and gather the premises and rules.
2. Convert the problem into First-Order Logic statements.
3. Identify the terms, constants, variables, and predicates in the statements.
4. Standardize the variables and rename them if necessary to avoid conflicts.
5. Apply unification to unify the terms in the statements.
6. Apply resolution to derive new statements that logically follow from the given premises.
7. Repeat steps 5 and 6 until the desired conclusion is reached or no further statements can be derived.

**Expt8**:

Algorithm:

1. Define the basic classes for the zoo example, including **Animal**, **Predator**, **Prey**, **Competes**, and **Avoids**.
2. Create an instance of **Animal** and assign it to a variable.
3. Create instances of **Predator**, **Prey**, **Competes**, and **Avoids** and assign them to variables.
4. Use the **set** method to set the relationships between the classes.
5. Use the **get** method to retrieve information about the relationships between the classes.
6. Print the output to the console.

**Expt9:**

Algorithm:

1. Collect historical stocks price data for a company.
2. Use collected data to calculate various statistical measures such as mean, standard deviation, variance, etc.
3. Choose parameter of chosen model.
4. Estimate parameter of chosen model.
5. Use estimate model and parameter to predict stock prices. Use uncertainty method.
6. Analyse generated scenarios.

**Expt10:**

Algorithm:

1. Create a visit to represent the world with each element representing a block.
2. Initialise the list with some blocks.
3. Allow the user to input command to manipulate the world.
4. Update the world list based on the user’s input.
5. Printout the updated state of the world.
6. Repeat steps 4-6 until the user choose to exit the program.

**Expt11:**

Algorithm:

1. Load the dataset into the program.
2. Split the dataset into training and testing datasets.
3. Define hypothesis and cost functions.
4. Define gradient descent algorithm to minimize the cost function.
5. Train the model using training set and set the gradient descent algorithm optimized to that particular set.
6. Test the model using testing model later.
7. Print out the model parameter.

**Expt12:**

Algorithm:

1. Load the dataset into the program.
2. Split the dataset into training and testing datasets.
3. Define a set of base models to use in the ensemble.
4. Combine the predictions of base model into ensemble models.
5. Test the ensemble model using testing dataset.
6. Evaluate the performance.

**Expt13:**

Algorithm:

1. Load the dataset.
2. Perform tokenization to split the text into individual words/sentences.
3. Perform part of search logging to assign grammatical logs to each word.
4. Perform named entity recognition to identify named entities like person, organization and location.
5. Perform sentiment analysis to determine sentiment.
6. Perform dependency parsing to identify relationship….
7. Perform machine translation to translate text from one language to other.

**Expt14:**

Algorithm:

1. Load the dataset.
2. Perform tokenization to split the text into individual words.
3. Perform part of speech tagging to assign grammatical tags.
4. Perform NER to identify named entities.
5. Perform sentiment analysis.
6. Do dependency parsing.
7. Perform machine translation to translate text.

**Expt15:**

Algorithm:

1. Collect and pre-process the data.
2. Divide the data into training and validation set.
3. Build a DL model using suitable framework.
4. Train the model using training set.
5. Tune the hyperparameters of a model.
6. Test and Evaluate(T&E) model’s performance using metrics.
7. Deploy the model.