Program 1

Solve the tic tac toe problem using DFS technique.

```python
class TicTacToe:
    def __init__(self):
        # Step 1: Initialize the board
        self.board = [[' ' for _ in range(3)] for _ in range(3)]
        self.player = 'X'  # AI player

    def print_board(self):
        # Step 2: Print the board
        for row in self.board:
            print(' | '.join(row))
            print('-' * 5)

    def is_draw(self):
        # Check if the game is a draw
        for row in self.board:
            if ' ' in row:
                return False
        return True

    def is_game_over(self):
        # Step 3: Check if the game is over
        # Check rows
        for row in self.board:
            if row.count(row[0]) == len(row) and row[0] != ' ':
                return row[0]
        # Check columns
        for col in zip(*self.board):
            if col.count(col[0]) == len(col) and col[0] != ' ':
                return col[0]

        # Check diagonals
        if self.board[0][0] == self.board[1][1] == self.board[2][2] != ' ':
            return self.board[0][0]
```

```
        if self.board[0][2] == self.board[1][1] == self.board[2][0] != '
':
            return self.board[0][2]
        return False

    def dfs(self, board, depth, player):
        # Step 5: DFS logic to choose the best move
        winner = self.is_game_over()
        if winner:
            if winner == 'X':  # AI wins
                return {'score': 1}
            else:  # Human wins
                return {'score': -1}
        elif self.is_draw():
            return {'score': 0}  # Draw

        if player == 'X':
            best = {'score': -float('inf')}
            symbol = 'X'
        else:
            best = {'score': float('inf')}
            symbol = 'O'

        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = symbol
                    score = self.dfs(board, depth + 1, 'O' if player ==
'X' else 'X')
                    board[i][j] = ' '
                    score['row'] = i
                    score['col'] = j

                    if player == 'X':
                        if score['score'] > best['score']:
                            best = score
                    else:
                        if score['score'] < best['score']:
                            best = score
        return best
```

```
    def play(self):
        # Game loop
        while True:
            self.print_board()
            winner = self.is_game_over()
            if winner or self.is_draw():
                print("Game Over.")
                if self.is_draw():
                    print("It's a draw!")
                else:
                    print(f"Player {winner} wins!")
                break

            if self.player == 'X':
                best_move = self.dfs(self.board, 0, 'X')
                self.board[best_move['row']][best_move['col']] = 'X'
            else:
                # Step 4: Accept keyboard input for 'O'
                while True:
                    try:
                        row = int(input("Enter the row number (0-2): "))
                        col = int(input("Enter the column number (0-2):
"))

                        if self.board[row][col] == ' ':
                            self.board[row][col] = 'O'
                            break
                        else:
                            print("Invalid move. Try again.")
                    except (ValueError, IndexError):
                        print("Invalid input. Please enter numbers between
0 and 2.")

            self.player = 'O' if self.player == 'X' else 'X'

game = TicTacToe()
game.play()
```

Program 2

Demonstrate the working of Alpha-Beta Pruning

```
# The minimax function is the heart of the AI. It recursively calculates
the optimal move for the AI.
def minimax(total, turn, alpha, beta):
    # Base case: if total is 20, it's a draw, so return 0
    if total == 20:
        return 0
    # Base case: if total is more than 20, the last player to move loses
    elif total > 20:
        if turn:  # If it's the AI's turn, AI loses, so return -1
            return -1
        else:  # If it's the human's turn, human loses, so return 1
            return 1

    # If it's the AI's turn, we want to maximize the score
    if turn:
        max_eval = -float('inf')  # Initialize max_eval to negative
infinity
        for i in range(1, 4):  # For each possible move (1, 2, or 3)
            # Recursively call minimax for the next state of the game
            eval = minimax(total + i, False, alpha, beta)
            max_eval = max(max_eval, eval)  # Update max_eval if necessary
            alpha = max(alpha, eval)  # Update alpha if necessary
            if beta <= alpha:  # If beta is less than or equal to alpha,
break the loop (alpha-beta pruning)
                break
        return max_eval  # Return the maximum evaluation
    # If it's the human's turn, we want to minimize the score
    else:
        min_eval = float('inf')  # Initialize min_eval to positive
infinity
        for i in range(1, 4):  # For each possible move (1, 2, or 3)
            # Recursively call minimax for the next state of the game
            eval = minimax(total + i, True, alpha, beta)
            min_eval = min(min_eval, eval)  # Update min_eval if necessary
            beta = min(beta, eval)  # Update beta if necessary
```

```
            if beta <= alpha:  # If beta is less than or equal to alpha,
break the loop (alpha-beta pruning)
                break
        return min_eval  # Return the minimum evaluation

# The total score of the game is initially 0
total = 0

# Game loop
while True:
    # Get the human player's move from input and add it to the total
    human_move = int(input("Enter your move (1, 2, or 3): "))
    while human_move not in [1, 2, 3]:  # If the move is not valid, ask
for input again
        print("Invalid move. Please enter 1, 2, or 3.")
        human_move = int(input("Enter your move (1, 2, or 3): "))
    total += human_move
    print(f"After your move, total is {total}")
    if total >= 20:  # If the total is 20 or more after the human's move,
the human wins
        print("You win!")
        break

    # If the game is not over, it's the AI's turn
    print("AI is making its move...")
    ai_move = 1
    max_eval = -float('inf')
    for i in range(1, 4):  # For each possible move (1, 2, or 3)
        # Call minimax to get the evaluation of the move
        eval = minimax(total + i, False, -float('inf'), float('inf'))
        if eval > max_eval:  # If the evaluation is greater than max_eval,
update max_eval and ai_move
            max_eval = eval
            ai_move = i
    total += ai_move  # Add the AI's move to the total
    print(f"AI adds {ai_move}. Total is {total}")
    if total >= 20:  # If the total is 20 or more after the AI's move, the
AI wins
        print("AI wins!")
        break
```

Program 3

Solve the 8-Puzzle problem using A* algorithm

```
import numpy as np
from queue import PriorityQueue

class State:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent

    def __lt__(self, other):
        return False  # Define a default comparison method

class Puzzle:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state


    def print_state(self, state):
        print(state[:, :])

    def is_goal(self, state):
        return np.array_equal(state, self.goal_state)

    def get_possible_moves(self, state):
        possible_moves = []
        zero_pos = np.where(state == 0)
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # Left, Right,
Up, Down
        for direction in directions:
            new_pos = (zero_pos[0] + direction[0], zero_pos[1] +
direction[1])
```

```
            if 0 <= new_pos[0] < 3 and 0 <= new_pos[1] < 3:  # Check
boundaries
                new_state = np.copy(state)
                new_state[zero_pos], new_state[new_pos] =
new_state[new_pos], new_state[zero_pos]  # Swap
                possible_moves.append(new_state)
        return possible_moves

    def heuristic(self, state):
         return np.count_nonzero(state != self.goal_state)


    def solve(self):
        queue = PriorityQueue()
        initial_state = State(self.initial_state, None)
        queue.put((0, initial_state))  # Put State object in queue
        visited = set()

        while not queue.empty():
            priority, current_state = queue.get()
            if self.is_goal(current_state.state):
                return current_state  # Return final state
            for move in self.get_possible_moves(current_state.state):
                move_state = State(move, current_state)  # Create new
State for move
                if str(move_state.state) not in visited:
                    visited.add(str(move_state.state))
                    priority = self.heuristic(move_state.state)
                    queue.put((priority, move_state))  # Put State object
in queue
        return None

# Test the function
initial_state = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])
goal_state = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])
puzzle = Puzzle(initial_state, goal_state)
solution = puzzle.solve()
move1 = -1
if solution is not None:
    moves = []
```

```
    while solution is not None:  # Go through parents to get moves
        moves.append(solution.state)
        solution = solution.parent
    for move in reversed(moves):
        move1+=1 # Print moves in correct order
        puzzle.print_state(move)
    print("no of moves: ",move1)
else:
    print("No solution found.")
```

Program 4

Implement the Hill Climbing search algorithm to maximise a single variable function f(x)

```
import numpy as np



def hill_climbing(func, start, step_size=0.01, max_iterations=1000):

    current_position = start

    current_value = func(current_position)



    for i in range(max_iterations):

        next_position_positive = current_position + step_size

        next_value_positive = func(next_position_positive)



        next_position_negative = current_position - step_size

        next_value_negative = func(next_position_negative)



        if next_value_positive > current_value and next_value_positive >=
next_value_negative:

            current_position = next_position_positive
```

```
            current_value = next_value_positive

        elif next_value_negative > current_value and next_value_negative > next_value_positive:

            current_position = next_position_negative

            current_value = next_value_negative

        else:

            break



    return current_position, current_value



# Get the function from the user

while True:

    func_str = input("\nEnter a function of x: ")

    try:

        # Test the function with a dummy value

        x = 0

        eval(func_str)

        break

    except Exception as e:
```

```
        print(f"Invalid function. Please try again. Error: {e}")


# Convert the string into a function

func = lambda x: eval(func_str)



# Get the starting point from the user

while True:

    start_str = input("\nEnter the starting value to begin the search: ")

    try:

        start = float(start_str)

        break

    except ValueError:

        print("Invalid input. Please enter a number.")



maxima, max_value = hill_climbing(func, start)

print(f"The maxima is at x = {maxima}")

print(f"The maximum value obtained is {max_value}")
```

Program 5

Logisitic Regression algorithm

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler



def sigmoid(z):

    return 1.0 / (1.0 + np.exp(-z))



def logistic_regression(X, y, num_iterations=200, learning_rate=0.001):

    weights = np.zeros(X.shape[1])

    for _ in range(num_iterations):

        z = np.dot(X, weights)

        h = sigmoid(z)

        gradient_val = np.dot(X.T, (h - y)) / y.shape[0]

        weights -= learning_rate * gradient_val

    return weights
```

```
# Load Iris dataset

iris = load_iris()

X = iris.data[:, :2]  # Use only the first two features (sepal length and
width)

y = (iris.target != 0) * 1  # Convert to binary classification



# Split the dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=9)



# Standardize features

sc = StandardScaler()

X_train_std = sc.fit_transform(X_train)

X_test_std = sc.transform(X_test)



# Perform logistic regression

weights = logistic_regression(X_train_std, y_train)
```

```python
# Make predictions

y_pred = sigmoid(np.dot(X_test_std, weights)) > 0.5



# Print accuracy

print(f'Accuracy: {np.mean(y_pred == y_test):.4f}')




# Plot decision boundary

x_min, x_max = X_train_std[:, 0].min() - 1, X_train_std[:, 0].max() + 1

y_min, y_max = X_train_std[:, 1].min() - 1, X_train_std[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),

                     np.arange(y_min, y_max, 0.1))



Z = sigmoid(np.dot(np.c_[xx.ravel(), yy.ravel()], weights)) > 0.5

Z = Z.reshape(xx.shape)



plt.contourf(xx, yy, Z, alpha=0.4)

plt.scatter(X_train_std[:, 0], X_train_std[:, 1], c=y_train, alpha=0.8)
```

```
plt.title('Logistic Regression Decision Boundaries')

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')



plt.savefig('plot.png')
```

Program 6

Naive Bayes Classifier

```
import numpy as np

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split



# Load iris dataset

iris = load_iris()

X, y = iris.data, iris.target

class_names = iris.target_names



class NaiveBayes:

    def fit(self, X, y):

        self._classes = np.unique(y)

        self._mean = np.array([X[y == c].mean(axis=0) for c in
self._classes])

        self._var = np.array([X[y == c].var(axis=0) for c in
self._classes])
```

```python
        self._priors = np.array([X[y == c].shape[0] / len(y) for c in
self._classes])



    def predict(self, X):

        return np.array([self._predict(x) for x in X])



    def _predict(self, x):

        posteriors = [np.log(prior) + np.sum(np.log(self._pdf(idx, x)))

                      for idx, prior in enumerate(self._priors)]

        return self._classes[np.argmax(posteriors)]



    def _pdf(self, class_idx, x):

        mean, var = self._mean[class_idx], self._var[class_idx]

        numerator = np.exp(- (x - mean)**2 / (2 * var))

        denominator = np.sqrt(2 * np.pi * var)

        return numerator / denominator
```

```python
# Split the dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)



# Create and train the Naive Bayes model

nb = NaiveBayes()

nb.fit(X_train, y_train)



# Make predictions

y_pred = nb.predict(X_test)

print('Accuracy: %.4f' % np.mean(y_pred == y_test))

# Print class names instead of class numbers

print("Predictions:", iris.target_names[y_pred])




### Optional confusion matrix



from sklearn.metrics import confusion_matrix, classification_report
```

```
# Print confusion matrix

print("\nConfusion Matrix:")

print(confusion_matrix(y_test, y_pred))



# Print classification report

print("\nClassification Report:")

print(classification_report(y_test, y_pred, target_names=class_names))
```

Program 7

KNN Algorithm

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np
from collections import Counter


# Load iris dataset
iris = load_iris()
X, y = iris.data, iris.target
class_names = iris.target_names

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        # Compute distances between x and all examples in the training set
        distances = []
        for x_train in self.X_train:
            distances.append(np.linalg.norm(x - x_train))
        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Extract the labels of the k nearest neighbor training samples
```

```python
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        # return the most common class label
        most_common = Counter(k_nearest_labels).most_common(1)
        #print(most_common)
        return most_common[0][0]

# Create a k-NN classifier with 3 neighbors
knn = KNN(k=3)

# Train the model using the training sets
knn.fit(X_train, y_train)

# Predict the response for test dataset
y_pred = knn.predict(X_test)
print('Accuracy: %.4f' % np.mean(y_pred == y_test))
print("Predictions:", class_names[y_pred])


# Optional confusion matrix
from sklearn.metrics import classification_report, confusion_matrix
# Print confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Program 8

K-means algorithm

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features (sepal length, sepal width, petal length, petal
width)

def kmeans(X, k):
    centroids = X[np.random.choice(X.shape[0], k, replace=False)]

    for _ in range(100):
        distances = np.linalg.norm(X[:, None] - centroids, axis=2)
        labels = np.argmin(distances, axis=1)
        centroids = np.array([X[labels == i].mean(axis=0) for i in
range(k)])

    return centroids, labels

# Apply custom k-means clustering
k = 3
centroids, labels = kmeans(X, k)

# Define colors for each cluster
colors = ['r', 'g', 'b']

# Plot the original data points with different colors for each cluster
for i in range(k):
```

```
    plt.scatter(X[labels == i, 0], X[labels == i, 1], c=colors[i],
label=f'Cluster {i+1}')

# Plot the final cluster centroids
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', c='black',
label='Centroids')

plt.title('K-Means Clustering on Iris Dataset')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend()
plt.show()
```