Program:

```
import random
class TicTacToe:
    def _init_(self):
        self.board = []
    def create_board(self):
        for i in range(3):
            row = []
            for j in range(3):
                row.append('-')
            self.board.append(row)
    def get_random_first_player(self):
        return random.randint(0, 1)
    def fix_spot(self, row, col, player):
        self.board[row][col] = player
    def is_player_win(self, player):
        win = None
        n = len(self.board)
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[i][j] != player:
                    win = False
                    break
            if win:
                return win
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[j][i] != player:
                    win = False
                    break
            if win:
                return win
        win = True
        for i in range(n):
            if self.board[i][i] != player:
                win = False
                break
        if win:
            return win
        win = True
        for i in range(n):
```

```
            if self.board[i][n - 1 - i] != player:
                win = False
                break
        if win:
            return win
        return False
        for row in self.board:
            for item in row:
                if item == '-':
                    return False
        return True
    def is_board_filled(self):
        for row in self.board:
            for item in row:
                if item == '-':
                    return False
        return True
    def swap_player_turn(self, player):
        return 'X' if player == 'O' else 'O'
    def show_board(self):
        for row in self.board:
            for item in row:
                print(item, end=" ")
            print()
    def start(self):
        self.create_board()
        player = 'X' if self.get_random_first_player() == 1 else 'O'
        while True:
            print(f"Player {player} turn")
            self.show_board()
            row, col = list(
                map(int, input("Enter row and column numbers to fix spot: ").split()))
            print()
            self.fix_spot(row - 1, col - 1, player)
            if self.is_player_win(player):
                print(f"Player {player} wins the game!")
                break
            if self.is_board_filled():
                print("Match Draw!")
                break
            player = self.swap_player_turn(player)
        print()
        self.show_board()
```

```
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

Program:

```
import random
import math
_goal_state = [[1,2,3],
               [4,5,6],
               [7,8,0]]
def index(item, seq):
    if item in seq:
        return seq.index(item)
    else:
        return -1
class EightPuzzle:
    def _init_(self):
        self._hval = 0
        self._depth = 0
        self._parent = None
        self.adj_matrix = []
        for i in range(3):
            self.adj_matrix.append(_goal_state[i][:])
    def _eq_(self, other):
        if self._class_ != other._class_:
            return False
        else:
            return self.adj_matrix == other.adj_matrix
    def _str_(self):
        res = ''
        for row in range(3):
            res += ' '.join(map(str, self.adj_matrix[row]))
            res += '\r\n'
        return res
    def _clone(self):
        p = EightPuzzle()
        for i in range(3):
            p.adj_matrix[i] = self.adj_matrix[i][:]
        return p
    def _get_legal_moves(self):
        row, col = self.find(0)
        free = []
        if row > 0:
            free.append((row - 1, col))
        if col > 0:
            free.append((row, col - 1))
        if row < 2:
```

```
            free.append((row + 1, col))
        if col < 2:
            free.append((row, col + 1))
        return free
    def _generate_moves(self):
        free = self._get_legal_moves()
        zero = self.find(0)
        def swap_and_clone(a, b):
            p = self._clone()
            p.swap(a,b)
            p._depth = self._depth + 1
            p._parent = self
            return p
        return map(lambda pair: swap_and_clone(zero, pair), free)
    def _generate_solution_path(self, path):
        if self._parent == None:
            return path
        else:
            path.append(self)
            return self._parent._generate_solution_path(path)
    def solve(self, h):
        def is_solved(puzzle):
            return puzzle.adj_matrix == _goal_state
        openl = [self]
        closedl = []
        move_count = 0
        while len(openl) > 0:
            x = openl.pop(0)
            move_count += 1
            if (is_solved(x)):
                if len(closedl) > 0:
                    return x._generate_solution_path([]), move_count
                else:
                    return [x]
            succ = x._generate_moves()
            idx_open = idx_closed = -1
            for move in succ:
                idx_open = index(move, openl)
                idx_closed = index(move, closedl)
                hval = h(move)
                fval = hval + move._depth

                if idx_closed == -1 and idx_open == -1:
```

```
                move._hval = hval
                openl.append(move)
            elif idx_open > -1:
                copy = openl[idx_open]
                if fval < copy._hval + copy._depth:
                    copy._hval = hval
                    copy._parent = move._parent
                    copy._depth = move._depth
            elif idx_closed > -1:
                copy = closedl[idx_closed]
                if fval < copy._hval + copy._depth:
                    move._hval = hval
                    closedl.remove(copy)
                    openl.append(move)
        closedl.append(x)
        openl = sorted(openl, key=lambda p: p._hval + p._depth)
    return [], 0
def shuffle(self, step_count):
    for i in range(step_count):
        row, col = self.find(0)
        free = self._get_legal_moves()
        target = random.choice(free)
        self.swap((row, col), target)
        row, col = target
def find(self, value):
    if value < 0 or value > 8:
        raise Exception("value out of range")
    for row in range(3):
        for col in range(3):
            if self.adj_matrix[row][col] == value:
                return row, col
def peek(self, row, col):
    return self.adj_matrix[row][col]
def poke(self, row, col, value):
    self.adj_matrix[row][col] = value
def swap(self, pos_a, pos_b):
    temp = self.peek(*pos_a)
    self.poke(pos_a[0], pos_a[1], self.peek(*pos_b))
    self.poke(pos_b[0], pos_b[1], temp)
```

```
def heur(puzzle, item_total_calc, total_calc):
    t = 0
    for row in range(3):
        for col in range(3):
```

```
            val = puzzle.peek(row, col) - 1
            target_col = val % 3
            target_row = val / 3
            if target_row < 0:
                target_row = 2
            t += item_total_calc(row, target_row, col, target_col)
    return total_calc(t)
def h_manhattan(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: abs(tr - r) + abs(tc - c),
                lambda t : t)
def h_manhattan_lsq(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: (abs(tr - r) + abs(tc - c))**2,
                lambda t: math.sqrt(t))

def h_linear(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: math.sqrt(math.sqrt((tr - r)*2 + (tc - c)*2)),
                lambda t: t)
def h_linear_lsq(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: (tr - r)*2 + (tc - c)*2,
                lambda t: math.sqrt(t))
def h_default(puzzle):
    return 0
def main():
    p = EightPuzzle()
    p.shuffle(20)
    print(p)
    path, count = p.solve(h_manhattan)
    path.reverse()
    for i in path:
        print(i)
    print("Solved with Manhattan distance exploring", count, "states")
    path, count = p.solve(h_manhattan_lsq)
    print ("Solved with Manhattan least squares exploring", count, "states")
    path, count = p.solve(h_linear)
    print ("Solved with linear distance exploring", count, "states")
    path, count = p.solve(h_linear_lsq)
    print ("Solved with linear least squares exploring", count, "states")
if _name_ == "_main_":
    main()
```

Program:

```
N = 8
def solveNQueens(board, col):
    if col == N:
      for i in board:
        print(i)
      return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQueens(board, col + 1):
                return True
            board[i][col] = 0
    return False
def isSafe(board, row, col):
    for x in range(col):
        if board[row][x] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    for x, y in zip(range(row, N, 1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    return True
board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0):
    print("No solution found”)
```

Program:

```
import sys
class Graph():
	def _init_(self, vertices):
		self.V = vertices
		self.graph = [[0 for column in range(vertices)]
					for row in range(vertices)]
	def printMST(self, parent):
		print("Edge \tWeight")
		for i in range(1, self.V):
			print(parent[i], "-", i, "\t", self.graph[i][parent[i]])
	def minKey(self, key, mstSet):
		min = sys.maxsize
		for v in range(self.V):
			if key[v] < min and mstSet[v] == False:
				min = key[v]
				min_index = v
		return min_index
	def primMST(self):
		key = [sys.maxsize] * self.V
		parent = [None] * self.V
		key[0] = 0
		mstSet = [False] * self.V
		parent[0] = -1
		for cout in range(self.V):
			u = self.minKey(key, mstSet)
			mstSet[u] = True
			for v in range(self.V):
				if self.graph[u][v] > 0 and mstSet[v] == False and key[v] >
self.graph[u][v]:
					key[v] = self.graph[u][v]
					parent[v] = u

		self.printMST(parent)
if _name_ == '_main_':
	g = Graph(5)
	g.graph = [[0, 2, 0, 6, 0],
			[2, 0, 3, 8, 5],
			[0, 3, 0, 0, 7],
			[6, 8, 0, 0, 9],
			[0, 5, 7, 9, 0]]

	g.primMST()
```

Program:

```
from collections import defaultdict
class Graph:
	def _init_(self, vertices):
		self.graph = defaultdict(list)
		self.V = vertices
	def addEdge(self, u, v):
		self.graph[u].append(v)
	def topologicalSortUtil(self, v, visited, stack):
		visited[v] = True
		for i in self.graph[v]:
			if visited[i] == False:
				self.topologicalSortUtil(i, visited, stack)
		stack.append(v)
	def topologicalSort(self):
		visited = [False]*self.V
		stack = []
		for i in range(self.V):
			if visited[i] == False:
				self.topologicalSortUtil(i, visited, stack)
		print(stack[::-1])
if _name_ == '_main_':
	g = Graph(6)
	g.addEdge(5, 2)
	g.addEdge(5, 0)
	g.addEdge(4, 0)
	g.addEdge(4, 1)
	g.addEdge(2, 3)
	g.addEdge(3, 1)
	print("Topological Sort:")
	g.topologicalSort()
```

Program:

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True
    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            break
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
source = 0
target = 9
best_first_search(source, target, v)
```

Program:

```
from copy import deepcopy
import numpy as np
import time
def bestsolution(state):
   bestsol = np.array([], int).reshape(-1, 9)
   count = len(state) - 1
   while count != -1:
      bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)
      count = (state[count]['parent'])
   return bestsol.reshape(-1, 3, 3)
def all(checkarray):
   set=[]
   for it in set:
      for checkarray in it:
         return 1
      else:
         return 0
def misplaced_tiles(puzzle,goal):
   mscost = np.sum(puzzle != goal) - 1
   return mscost if mscost > 0 else 0


def coordinates(puzzle):
   pos = np.array(range(9))
   for p, q in enumerate(puzzle):
      pos[q] = p
   return pos
def evaluvate_misplaced(puzzle, goal):
   steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5,
8],  1)],
            dtype =  [('move',  str, 1),('position', list),('head', int)])
   dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]
   costg = coordinates(goal)
   parent = -1
   gn = 0
   hn = misplaced_tiles(coordinates(puzzle), costg)
   state = np.array([(puzzle, parent, gn, hn)], dtstate)
   dtpriority = [('position', int),('fn', int)]
   priority = np.array([(0, hn)], dtpriority)
   while 1:
      priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
      position, fn = priority[0]
```

```
        priority = np.delete(priority, 0, 0)
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
        blank = int(np.where(puzzle == 0)[0])
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable \n")
                        break
                    hn = misplaced_tiles(coordinates(openstates), costg)
                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    fn = gn + hn
                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                    if np.array_equal(openstates, goal):
                        print(' The 8 puzzle is solvable \n')
                        return state, len(priority)
    return state, len(priority)
puzzle = []
puzzle.append(2)
puzzle.append(8)
puzzle.append(3)
puzzle.append(1)
puzzle.append(6)
puzzle.append(4)
puzzle.append(7)
puzzle.append(0)
puzzle.append(5)
goal = []
goal.append(1)
goal.append(2)
goal.append(3)
goal.append(8)
```

```
goal.append(0)
goal.append(4)
goal.append(7)
goal.append(6)
goal.append(5)
state, visited = evaluvate_misplaced(puzzle, goal)
bestpath = bestsolution(state)
print(str(bestpath).replace('[', ' ').replace(']', ''))
totalmoves = len(bestpath) - 1
print('\nSteps to reach goal:',totalmoves)
visit = len(state) - visited
print('Total nodes visited: ',visit, “\n")
```

Program:

```
import math
def minimax (curDepth, nodeIndex,maxTurn, scores,targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

Program:

```
def get_index_comma(string):
  index_list = list()
  par_count = 0
  for i in range(len(string)):
    if string[i] == ',' and par_count == 0:
      index_list.append(i)
    elif string[i] == '(':
      par_count += 1
    elif string[i] == ')':
      par_count-=1
    return index_list
def is_variable(expr):
  for i in expr:
    if i=='(':
      return False
  return True
def process_expression(expr):
  expr = expr.replace('','')
  index = None
  for i in range(len(expr)):
    if expr[i]=='(':
      index=i
      break
  predicate_symbol = expr[:index]
  expr = expr.replace(predicate_symbol,'')
  expr = expr[1:len(expr)-1]
  arg_list = list()
  indices = get_index_comma(expr)
  if len(indices) == 0:
    arg_list.append(expr)
  else:
    arg_list.append(expr[:indices[0]])
    for i,j in zip(indices,indices[1:]):
      arg_list.append(expr[i+1:j])
      arg_list.append(expr[indices[len(indices)-1]+1:])
  return predicate_symbol,arg_list
def get_arg_list(expr):
  _,arg_list = process_expression(expr)

  flag = True
  while flag:
```

```
    flag = False
    for i in arg_list:
      if not is_variable(i):
        flag = True
        _,tmp = process_expression(i)
        for j in tmp:
          if j not in arg_list:
            arg_list.append(j)
          arg_list.remove(i)
  return arg_list
def check_occurs(var,expr):
  arg_list = get_arg_list(expr)
  if var in arg_list:
    return True
  return False
def unify(expr1,expr2):
  if is_variable(expr1) and is_variable(expr2):
    if expr1==expr2:
      return 'Null'
    else:
      return False
  elif is_variable(expr1) and not is_variable(expr2):
    if check_occurs(expr1, expr2):
      return False
    else:
      tmp = str(expr2) + '/'+str(expr1)
      return tmp
  elif not is_variable(expr1) and is_variable(expr2):
    if check_occurs(expr2,expr1):
      return False
    else:
      tmp = str(expr1) + '/' + str(expr2)
      return tmp
  else:
    predicate_symbol_1,arg_list_1 = process_expression(expr1)
    predicate_symbol_2,arg_list_2 = process_expression(expr2)
    if predicate_symbol_1 != predicate_symbol_2:
      return False
    elif len(arg_list_1) != len(arg_list_2):
      return False
    else:
      sub_list = list()
      for i in range(len(arg_list_1)):
```

```
        tmp = unify(arg_list_1[i],arg_list_2[i])
        if not tmp:
          return False
        elif tmp == 'Null':
          pass
        else:
          if type(tmp)==list:
            for j in tmp:
              sub_list.append(j)
          else:
            sub_list.append(tmp)
        return sub_list
if _name=='main_':
  f1 = 'p(b(A),X,f(g(Z)))'
  f2 = 'p(Z,f(Y),f(Y))'
  result = unify(f1,f2)
  if not result:
    print('Unification failed!')
  else:
    print('Unification successfully!')
  print(result)
```

Program:

```
P = "P"
Q = "Q"
R = "R"
kb = [
    (P, "=>", Q),
    (Q, "=>", R),
    (P,),
]
def is_true(sentence, model):
    if sentence[0] == "not":
        return not is_true(sentence[1], model)
    elif sentence[0] in model:
        return model[sentence[0]]
    elif len(sentence) == 1:
        return False
    elif sentence[1] == "and":
        return is_true(sentence[0], model) and is_true(sentence[2], model)
    elif sentence[1] == "or":
        return is_true(sentence[0], model) or is_true(sentence[2], model)
    elif sentence[1] == "=>":
        return not is_true(sentence[0], model) or is_true(sentence[2], model)
    elif sentence[1] == "<=>":
        return is_true(sentence[0], model) == is_true(sentence[2], model)
def is_model_satisfies_kb(model, kb):
    for sentence in kb:
        if not is_true(sentence, model):
            return False
    return True
def generate_models(symbols):
    if not symbols:
        return [{}]
    else:
        symbol = symbols[0]
        rest = symbols[1:]
        models = []
        for model in generate_models(rest):
            models.append(model)
            models.append({*model, *{symbol: True}})
            models.append({*model, *{symbol: False}})
        return models
symbols = [P, Q, R]
models = generate_models(symbols)
```

```
for model in models:
    if is_model_satisfies_kb(model, kb):
        print(model)
```

Program:

```
import numpy as np
import matplotlib.pyplot as plt
x_func = np.linspace(-4,4,100)
y_func = x_func
x_train = np.random.uniform(-3,-2,50)
y_train = x_train + np.random.randn(*x_train.shape) * 0.5
x_train = np.concatenate([x_train, np.random.uniform(2, 3, 50)])

y_train = np.concatenate([y_train, x_train[50:] + np.random.randn(*x_train[50:].shape) *
0.1])
x_test = np.linspace(-10, 10, 100)
fig, ax = plt.subplots(1, 1, figsize = (10, 5))
ax.scatter(x_train, y_train, label = 'training data')
ax.plot(x_func, y_func, ls='--', label = 'real function', color = 'green')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.set_title('Data with uncertainty')
```

Program:

```
def test():
    test_agent = BlockWorldAgent()
    initial_arrangement_1 = [["A", "B", "C"], ["D", "E"]]
    goal_arrangement_1 = [["A", "C"], ["D", "E", "B"]]
    goal_arrangement_2 = [["A", "B", "C", "D", "E"]]
    goal_arrangement_3 = [["D", "E", "A", "B", "C"]]
    goal_arrangement_4 = [["C", "D"], ["E", "A", "B"]]
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_1))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_2))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_3))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_4))
    initial_arrangement_2 = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"]]
    goal_arrangement_5 = [["A", "B", "C", "D", "E", "F", "G", "H", "I"]]
    goal_arrangement_6 = [["I", "H", "G", "F", "E", "D", "C", "B", "A"]]
    goal_arrangement_7 = [["H", "E", "F", "A", "C"], ["B", "D"], ["G", "I"]]
    goal_arrangement_8 = [["F", "D", "C", "I", "G", "A"], ["B", "E", "H"]]
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_5))
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_6))
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_7))
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_8))
if _name_ == "_main_":
    test()
```

Program:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create SVM model
svm = SVC(kernel='linear')

# Train SVM model
svm.fit(X_train, y_train)

# Predict using SVM model
y_pred = svm.predict(X_test)

# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)

# Print accuracy score
print('Accuracy:', accuracy)
```

Program:

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
data = load_breast_cancer()
X = data.data
y = data.target

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create individual models
lr = LogisticRegression(random_state=42)
svc = SVC(kernel='linear', probability=True, random_state=42)
rf = RandomForestClassifier(n_estimators=10, random_state=42)

# Create ensemble model
ensemble = VotingClassifier(estimators=[('lr', lr), ('svc', svc), ('rf', rf)], voting='soft')

# Train ensemble model
ensemble.fit(X_train, y_train)

# Predict using ensemble model
y_pred = ensemble.predict(X_test)

# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)

# Print accuracy score
print('Accuracy:', accuracy)
```

Program:

```
import nltk
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Define sample text
text = "Natural Language Processing (NLP) is a subfield of linguistics, computer science,
and artificial intelligence concerned with the interactions between computers and human
(natural) languages."

# Tokenize text
tokens = word_tokenize(text)

# Remove stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# Lemmatize tokens
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(token) for token in filtered_tokens]

# Print lemmatized tokens
print(lemmatized_tokens)
```

Program:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag
nltk.download('averaged_perceptron_tagger')

# Define sample text
text = "Natural Language Processing (NLP) is a subfield of linguistics, computer science,
and artificial intelligence concerned with the interactions between computers and human
(natural) languages."

# Tokenize text into words
words = word_tokenize(text)

# Tag parts of speech for each word
pos_tags = pos_tag(words)

# Print parts of speech tags
print(pos_tags)
```

Program:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag, ne_chunk
nltk.download('maxent_ne_chunker')
nltk.download('words')

# Define sample text
text = "Barack Obama was the 44th President of the United States."

# Tokenize text into words
words = word_tokenize(text)

# Tag parts of speech for each word
pos_tags = pos_tag(words)

# Recognize named entities
named_entities = ne_chunk(pos_tags)

# Print named entities
print(named_entities)
```

Program:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Define sample text
text = "Natural Language Processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human (natural) languages."

# Tokenize text
tokens = word_tokenize(text)

# Remove stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# Lemmatize tokens
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(token) for token in filtered_tokens]

# Print lemmatized tokens
print(lemmatized_tokens)
```

Program:

```
#importing the required libraries
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
#loading data
(X_train,y_train) , (X_test,y_test)=mnist.load_data()
#reshaping data
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], X_train.shape[2], 1))
X_test = X_test.reshape((X_test.shape[0],X_test.shape[1],X_test.shape[2],1))
#checking the shape after reshaping
print(X_train.shape)
print(X_test.shape)
#normalizing the pixel values
X_train=X_train/255
X_test=X_test/255
#defining model
model=Sequential()
#adding convolution layer
model.add(Conv2D(32,(3,3),activation='relu',input_shape=(28,28,1)))
#adding pooling layer
model.add(MaxPool2D(2,2))
#adding fully connected layer
model.add(Flatten())
model.add(Dense(100,activation='relu'))
#adding output layer
model.add(Dense(10,activation='softmax'))
#compiling the model
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy
'])
#fitting the model
model.fit(X_train,y_train,epochs=10)
```