BFS:

```
visited = []
queue = []
found=0

def bfs(visited, graph, node, goal_state):
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")
    if(m==goal_state):
      print("GOAL", end= " ")
      global found
      found=1
      break


    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)


# graph = {
#   '5' : ['3','7'],
#   '3' : ['2', '4'],
#   '7' : ['8'],
#   '2' : [],
#   '4' : ['8'],
#   '8' : []
# }

graph = {
  '1' : ['2','3','4'],
  '2' : ['5', '6'],
  '3' : ['7'],
```

```
  '4' : ['8','9'],
  '5' : [],
  '6' : [],
  '7' : [],
  '8' : [],
  '9' : []
}

print("Following is the Breadth-First Search")
bfs(visited, graph, '1','9')

if found==0:
  print("Not found")
```

DFS:

```
def dfs(graph, node,path = []):
    if node not in path:
        path.append(node)
        for neighbour in graph[node]:
            path = dfs(graph, neighbour, path)
    return path

graph = {
    '5' : ['3','7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [],
    '4' : ['8'],
    '8' : []
}
#       5
#     /   \
#    3     7
#  /   \     \
# 2     4----8

print("Depth-First Search traversal sequence")
```

```python
path = dfs(graph, "5")
goal_state = '8'
new_path=[]
for item in path:
    if(item==goal_state):
        break
    else:
        new_path.append(item)
new_path.append("GOAL FOUND🎊")
print(" ".join(new_path))
```

DFID:

```python
from collections import defaultdict
class Graph:
    def __init__(self,vertices):
        self.V = vertices # Number of vertices
        self.graph = defaultdict(list)

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def DLS(self,src,target,maxDepth):
        if src == target : return True
        if maxDepth <= 0 : return False
        for i in self.graph[src]:
            if(self.DLS(i,target,maxDepth-1)):
                return True
        return False

    def IDDFS(self,src, target, maxDepth):
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False

# Create a graph
#         0
```

```python
#       /   \
#      1      2
#     / \     | \
#    3   4   5   6

g = Graph (7)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 3
maxDepth = 1
src = 0

if g.IDDFS(src, target, maxDepth) == True:
    print ("Target "+str(target) +" is reachable from source
within max depth of "+str(maxDepth))
else :
    print ("Target "+str(target) +" is NOT reachable from
source within max depth of "+str(maxDepth))
```

A-star:

```python
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H={
            'A':1,
            'B':1,
            'C':1,
            'D':1
        }
```

```
        return H[n]

    def a_star(self, start, stop):

        open = set([start])
        closed = set([])

        g = {}
        g[start] = 0

        parents = {}
        parents[start] = start

        while len(open)>0:
            n = None
            print("Open List: ",open,"Closed List: ",closed)

            for v in open:
                if n==None or g[v]+self.h(v)<g[n]+self.h(n):
                    n = v

            if n==None:
                print("not found")
                return None

            if n==stop:
                open.remove(stop)
                closed.add(stop)
                print("Open List: ",open,"Closed List:
",closed)

                path=[]
                while parents[n]!=n:
                    path.append(n)
                    n = parents[n]
                path.append(start)
                path.reverse()

                return path

            for (m, weight) in self.get_neighbors(n):
                if m not in open and m not in closed:
```

```
                    open.add(m)
                    g[m] = g[n]+weight
                    parents[m] = n
                else:
                    if g[m]>g[n]+weight:
                        g[m] = g[n]+weight
                        parents[m]=n

                        if m in closed:
                            open.add(m)
                            closed.remove(m)
            open.remove(n)
            closed.add(n)

        print("path doesnt exist")
        return None

adjacency_list = {
    'A': [('B',1),('C',3),('D',7)],
    'B': [('D',5)],
    'C': [('D',12)],
}
g = Graph(adjacency_list)
g.a_star('A','D')
```

Hill Climbing:

```
import copy
visited_states = []

def heuristic(curr_state,goal_state):
    goal_=goal_state[3]
    val=0
    for i in range(len(curr_state)):
        check_val=curr_state[i]
        if len(check_val)>0:
            for j in range(len(check_val)):
                if check_val[j]!=goal_[j]:
                    val-=j
                else:
                    val+=j
    return val
```

```
def generate_next(curr_state,prev_heu,goal_state):
    global visited_states
    state = copy.deepcopy(curr_state)

    for i in range(len(state)):
        temp = copy.deepcopy(state)

        if len(temp[i]) > 0:
            elem = temp[i].pop()

            for j in range(len(temp)):
                temp1 = copy.deepcopy(temp)

                if j != i:
                    temp1[j] = temp1[j] + [elem]

                    if (temp1 not in visited_states):
                        curr_heu=heuristic(temp1,goal_state)
                        if curr_heu>prev_heu:
                            child = copy.deepcopy(temp1)
                            return child
    return 0

def solution_(init_state,goal_state):
    global visited_states
    if (init_state == goal_state):
        print (goal_state)
        print("solution found!")
        return
    current_state = copy.deepcopy(init_state)
    while(True):
        visited_states.append(copy.deepcopy(current_state))
        print(current_state)
        prev_heu=heuristic(current_state,goal_state)
        child =
generate_next(current_state,prev_heu,goal_state)
        if child==0:
            print("Final state - ",current_state)
            return
        current_state = copy.deepcopy(child)

def solver():
    global visited_states
    init_state = [[],[],[],['B','C','D','A']]
```

```
    goal_state = [[],[],[],['A','B','C','D']]
    solution_(init_state,goal_state)
solver()
```

Genetic:

```
import random

def generate_population(population_size, gene_length):
    return [''.join(random.choice('01') for _ in
range(gene_length)) for _ in range(population_size)]

def calculate_fitness(individual, target):
    return sum(1 for a, b in zip(individual, target) if a == b)

def select_parents(population, target):
    fitness_scores = [calculate_fitness(individual, target) for
individual in population]
    total_fitness = sum(fitness_scores)

    if total_fitness == 0:
        return random.sample(population, 2)

    probabilities = [score / total_fitness for score in
fitness_scores]

    parents = random.choices(population, probabilities, k=2)

    return parents

def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(parent1) - 1)
    child1 = parent1[:crossover_point] +
parent2[crossover_point:]
    child2 = parent2[:crossover_point] +
parent1[crossover_point:]
    return child1, child2

def mutate(individual, mutation_rate):
    mutated_individual = ''.join(
        bit if random.random() > mutation_rate else
random.choice('01')
        for bit in individual
    )
    return mutated_individual
```

```
def genetic_algorithm(target, population_size, mutation_rate,
generations):
    population = generate_population(population_size,
len(target))

    for generation in range(generations):
        population = sorted(population, key=lambda x:
calculate_fitness(x, target), reverse=True)

        print(f"Generation {generation}: {population[0]}
(Fitness: {calculate_fitness(population[0], target)})")

        if calculate_fitness(population[0], target) ==
len(target):
            print("Target achieved!")
            break

        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = select_parents(population,
target)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
            new_population.extend([child1, child2])

        population = new_population

    if calculate_fitness(population[0], target) != len(target):
        print("Nearest match reached is: ", population[0])
        binary_string = str(population[0])

        decimal_number_again = int(binary_string, 2)
        print(decimal_number_again)



if __name__ == "__main__":

    num = int(input("Enter a number: "))

    binNum = bin(num).replace("0b","")
    print(binNum)
```

```
    target_string = str(binNum)
    population_size = 100
    mutation_rate = 0.01
    generations = 1000

    genetic_algorithm(target_string, population_size,
mutation_rate, generations)
```

Perceptron

```
import numpy as np

X1 = np.array([1, -2, 0, -1])
X2 = np.array([0, 1.5, -0.5, -1])
X3 = np.array([-1, 1, 0.5, -1])

X = np.array([X1, X2, X3])
W = np.array([1, -1, 0, 0.5])

d = np.array([-1, -1, 1])

c = 0.1
epochs = 1


for i in range(epochs):
    print("Iteration ", i+1)
    for j in range(len(X)):
        net = np.dot(X[j], W)

        if (net <= 0):
            op = -1
        elif net > 0:
            op = 1

        error = d[j] - op

        dW = c*error*X[j]
        W += dW
        print("W", j,  W)

    print("\nW after ", i+1, " epochs ", W)
    # c=c/2
print("Final W after ", epochs, "epochs:")
print(W)
```

prolog:

```
parent(sunil, prerna).
parent(sunil, diksha).
parent(sunil, krishna).
parent(swati, prerna).
parent(swati, diksha).
parent(swati, krishna).

parent(sujit, kajal).
parent(sujit, yash).
parent(malti, kajal).
parent(malti, yash).

parent(sulochana, sunil).
parent(sulochana, sujit).
parent(ram, sunil).
parent(ram, sujit).

female(prerna).
female(diksha).
female(kajal).
female(swati).
female(malti).
female(sulochana).

male(ram).
male(sunil).
male(sujit).
male(yash).
male(krishna).

/*Rules*/
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X).
grandmother(X,Y) :- parent(Z,Y), parent(X,Z), female(X).
grandfather(X,Y) :- parent(Z,Y), parent(X,Z), male(X).
```

grandmother(sulochana, kajal)

true

Next | 10 | 100 | 1,000 | Stop

grandmother(malti, kajal)

false

mother(malti, kajal)

true

brother(yash, kajal)

true

Next | 10 | 100 | 1,000 | Stop

grandfather(ram, sunil)

false

grandfather(ram, krishna)

true

Next | 10 | 100 | 1,000 | Stop

father(sunil, krishna)