Alpha beta pruning:

```python
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):
    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best
```

```
# Driver Code
if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1 ]
    print("The optimal value is :", minimax(0, 0, True, values,
MIN, MAX))
```

Astar:

```
import heapq

def a_star(grid, start, end):
    # Helper function to calculate the heuristic cost
    def heuristic(a, b):
        return abs(b[0] - a[0]) + abs(b[1] - a[1])

    # Create a priority queue to store the next steps to explore
    queue = []
    heapq.heappush(queue, (0, start))

    # Create a dictionary to store the cost of each point
    cost = {start: 0}

    # Create a dictionary to store the parent of each point
    parent = {start: None}

    # Create a set to store the visited points
    visited = set()

    # While there are points to explore
    while queue:
        # Get the point with the lowest cost
        current = heapq.heappop(queue)[1]

        # If we have reached the end point, construct the path
        if current == end:
            path = []
            while current != start:
                path.append(current)
                current = parent[current]
            path.append(start)
            return path[::-1]

        # Mark the current point as visited
        visited.add(current)
```

```
        # Explore the neighboring points
        for i, j in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            neighbor = current[0] + i, current[1] + j
            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] <
len(grid[0]) and grid[neighbor[0]][neighbor[1]] == 0 and neighbor
not in visited:
                # Calculate the new cost
                new_cost = cost[current] + 1

                # If the neighbor has not been visited or the new
cost is lower
                if neighbor not in cost or new_cost <
cost[neighbor]:
                    # Update the cost and parent of the neighbor
                    cost[neighbor] = new_cost
                    priority = new_cost + heuristic(end, neighbor)
                    heapq.heappush(queue, (priority, neighbor))
                    parent[neighbor] = current
    return None

# Example usage:
grid = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
start = (0, 0)
end = (7, 6)
path = a_star(grid, start, end)
print(path)
```

BFS:

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
```

```python
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
```

```
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                " (starting from vertex 2)")
g.BFS(2)

# This code is contributed by Neelam Yadav
```

DFS:

```
# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print(v),

        # Recur for all the vertices adjacent to
        # this vertex
        for i in self.graph[v]:
            if visited[i] == False:
```

```
                self.DFSUtil(i, visited)


    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self):
        V = len(self.graph) #total vertices

        # Mark all the vertices as not visited
        visited =[False]*(V)

        # Call the recursive helper function to print
        # DFS traversal starting from all vertices one
        # by one
        for i in range(V):
            if visited[i] == False:
                self.DFSUtil(i, visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is Depth First Traversal")
g.DFS()

# This code is contributed by Neelam Yadav
```

Magic square:

```
def generateSquare(n):
    magicSquare = [[0 for x in range(n)]
                   for y in range(n)]
    i = n // 2
    j = n - 1
    num = 1
    while num <= (n * n):
        if i == -1 and j == n:  # 3rd condition
```

```
            j = n - 2
            i = 0
        else:

            # next number goes out of
            # right side of square
            if j == n:
                j = 0
            if i < 0:
                i = n - 1

        if magicSquare[int(i)][int(j)]:  # 2nd condition
            j = j - 2
            i = i + 1
            continue
        else:
            magicSquare[int(i)][int(j)] = num
            num = num + 1

        j = j + 1
        i = i - 1  # 1st condition

    # Printing magic square
    print("Magic Square for n =", n)
    print("Sum of each row or column",
          n * (n * n + 1) // 2, "\n")

    for i in range(0, n):
        for j in range(0, n):
            print('%2d ' % (magicSquare[i][j]),
                  end='')

            # To display output
            # in matrix form
            if j == n - 1:
                print()
n = 7
generateSquare(n)
```

Minimax:

```
import math


def minimax(curDepth, nodeIndex,
```

```
            maxTurn, scores,
            targetDepth):
    # base case : targetDepth reached
    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))


# Driver code
scores = [2, 5, 11, 4, 14, 15, 3, 4]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end="")
print(minimax(0, 0, True, scores, treeDepth))
```

Waterjug:

```
from collections import deque


def BFS(a, b, target):

    m = {}
    isSolvable = False
    path = []


    q = deque()

    q.append((0, 0))

    while (len(q) > 0):
        u = q.popleft()# If this state is already visited
        if ((u[0], u[1]) in m):
```

```
    continue
if ((u[0] > a or u[1] > b or
    u[0] < 0 or u[1] < 0)):
    continue

# Filling the vector for constructing
# the solution path
path.append([u[0], u[1]])

# Marking current state as visited
m[(u[0], u[1])] = 1

# If we reach solution state, put ans=1
if (u[0] == target or u[1] == target):
    isSolvable = True

    if (u[0] == target):
        if (u[1] != 0):

            # Fill final state
            path.append([u[0], 0])
    else:
        if (u[0] != 0):

            # Fill final state
            path.append([0, u[1]])

    # Print the solution path
    sz = len(path)
    for i in range(sz):
        print("(", path[i][0], ",",
            path[i][1], ")")
    break

# If we have not reached final state
# then, start developing intermediate
# states to reach solution state
q.append([u[0], b]) # Fill Jug2
q.append([a, u[1]]) # Fill Jug1

for ap in range(max(a, b) + 1):

    # Pour amount ap from Jug2 to Jug1
    c = u[0] + ap
    d = u[1] - ap
```

```
            # Check if this state is possible or not
            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            # Pour amount ap from Jug 1 to Jug2
            c = u[0] - ap
            d = u[1] + ap

            # Check if this state is possible or not
            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        # Empty Jug2
        q.append([a, 0])

        # Empty Jug1
        q.append([0, b])

    # No, solution exists if ans=0
    if (not isSolvable):
        print("No solution")


# Driver code
if __name__ == '__main__':

    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
        "to solution state ::")

    BFS(Jug1, Jug2, target)

# This code is contributed by mohit kumar 29
```

Chatterbot:

```
spe=["Software Modelling &DevOps","Internet Of Things","Cloud &
EdgeComputing","Graphics,Gaming & UX Design","Cyber Security &
BlockchainTechnology","Artificial Intelligence & Intelligence
Process Automation","Data Science and BigData Analytics","Computer
Communications" ]
cer=["Professional scrum Master","None","Linux Essential(101-
160)","UnityDeveloper Advance Certification","ETHERIUM Developer
```

```
AdvanceCertification","PCAP|CertifiedAssociatePythonProgramming","C
100DEV:MongoDB certified DeveloperAssociate","None"]
print("Hello student, I am student advisor")
print("May I know your name?")
name=input()
print("Thank you", name)
print("I am here to help you explore through the specialisations
offered inCSE Department of K L University.")
print("Here are the list of specialisations")
for i in range(0,8):
 print((i),".",spe[i])
print("Choose any specialisation")
ch=int(input())
print("Your courses are:")
print("Specialisation
choose:",spe[ch],",","GlobalCertification:",cer[ch])
```