**WHAT is Depth First Search ?**

Depth first search or depth first traversal is a recursive algorithm for

Searching all the vertices of a graph or tree data structure traversal

Means visiting all the nodes of a graph.

A standard DFS implementation plus each vertex of the graph into one of

Two categories:

Visited

Not visited

The purpose of the algorithm is to mark each vertex as visited while

Avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph’s vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex’s adjacent nodes. Add the ones which aren’t

In the visited list to the top of the stack.

1. Keep repeating steps 2 and 3 until the stack is empty.

For Example:-

graph1={

'A':set(['B','C']),

'B':set(['A','D','E']),

'C':set(['A','F']),

'D':set(['B']),

'E':set(['B','F']),

'F':set(['C','E']),

}

def dfs(graph,node,visited):

if node not in visited:

visited.append(node)

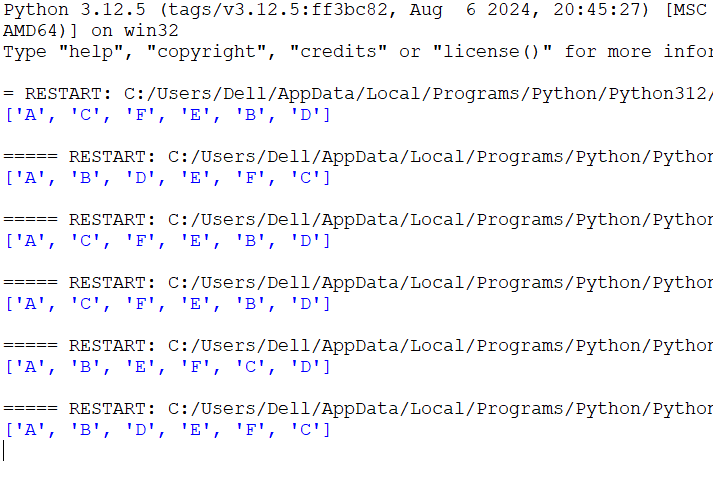
for n in graph[node]:

dfs(graph,n,visited)

return visited

visited=dfs(graph1,'A',[])

print(visited)



# 

**PRACTICAL:- 1**

graph= {

'B':set(['A','D','E']),

'A':set(['B','F','G']),

'F':set(['A']),

'G':set(['A']),

'D':set(['B']),

'E':set(['H','B']),

'H':set(['E','B']),

}

def dfs(graph,node,visited):

if node not in visited:

visited.append(node)

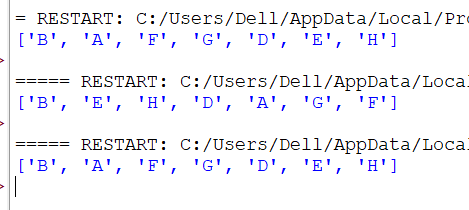
for n in graph[node]:

dfs(graph,n,visited)

return visited

visited=dfs(graph1,'B',[])

print(visited)



**PRACTICAL 1(B)**

**Breath First Search (BFS)**

graph = {

'A':set(['B','C','D']),

'B':set(['A','E']),

'C':set(['A',]),

'D':set(['A','F','G']),

'E':set(['B','H']),

'F':set(['D','H']),

'G':set(['D']),

'H':set(['E','F']),

}

def bfs(start):

queue = [start]

levels = {}

levels[start] = 0

visited = set([start])

while queue:

node = queue.pop(0)

neighbours = graph[node]

for neighbor in neighbours:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

levels[neighbor] = levels[node] + 1

print(levels)

return visited

print(str(bfs('A')))

def bfs\_paths(graph, start, goal):

queue = [(start, [start])]

while queue :

(vertex, path) = queue.pop(0)

for next in graph[vertex] - set(path):

if next == goal:

yield path + [next]

else:

queue.append((next, path + [next]))

result = list(bfs\_paths(graph, 'A', 'F'))

print(result)

def shortest\_path(graph, start, goal):

try:

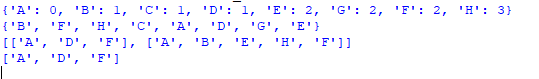
return next(bfs\_paths(graph, start, goal))

except StopIteration:

return None

result1 = shortest\_path(graph, 'A', 'F')

print(result1)



# 

**PRACTICAL:- 2**

**Write a program to simulate 4\_Queen / n\_Queen problem.**

def is\_safe(board, row, col, N):

for i in range(row):

if board[i][col] ==1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, N)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, row, N, solutions):

if row == N:

solution = []

for i in range(N):

sol\_row = ""

for j in range(N):

if board[i][j] == 1:

sol\_row += 'Q'

else:

sol\_row += '.'

solution.append(sol\_row)

solutions.append(solution)

return

for col in range(N):

if is\_safe(board, row, col, N):

board[row][col] = 1

solve\_n\_queens\_util(board, row + 1, N, solutions)

board[row][col] = 0

def solve\_n\_queens(N):

board = [[0] \* N for \_ in range(N)]

solutions = []

solve\_n\_queens\_util(board, 0, N, solutions)

return solutions

def print\_solutions(solutions):

for idx, solution in enumerate(solutions):

print(f"Solution {idx + 1}:")

for row in solution:

print(row)

print()

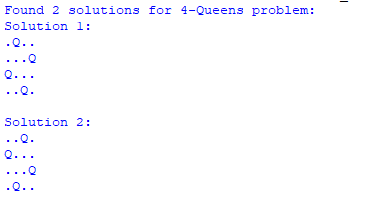
if \_\_name\_\_ == "\_\_main\_\_":

N=10

solutions = solve\_n\_queens(N)

print(f"Found {len(solutions)} solutions for {N}-Queens problem:")

print\_solutions(solutions)



**PRACTICAL 2(B)**

def tower\_of\_hanoi(n, source , target, auxiliary):

if n== 1:

print(f"move disk 1 from {source} to {target}")

return

tower\_of\_hanoi(n - 1, source, auxiliary, target)

print(f"move disk {n} from {source} to {target}")

tower\_of\_hanoi(n -1, auxiliary, target, source)

def tower\_of\_hanoi\_solver(num\_disks):

print(f"solving Tower of Hanoi problem for {num\_disks} disks:")

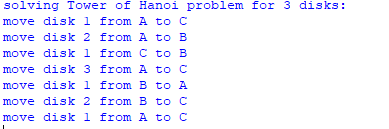
tower\_of\_hanoi(num\_disks,'A','C','B')

if \_\_name\_\_=="\_\_main\_\_":

num\_disks = 3 # change this to solve for different number of disks

tower\_of\_hanoi\_solver(num\_disks)

**OUTPUT:-**



**PRACTICAL 3**

Write a program of water jug problem

capacity = (12, 8, 5)

initial\_state = (12, 0, 0)

final\_state = (6, 6, 0)

visited = set()

solution\_path = []

def get\_all\_states(state):

if state == final\_state:

solution\_path.append(state)

return True

if state in visited:

return False

visited.add(state)

a, b, c = state

jugs = [a, b, c]

cap = capacity

for i in range(3):

for j in range(3):

if i !=j and jugs[i] > 0 and jugs[j] < cap[j]:

poured = min (jugs[i], cap[j] - jugs[j])

next\_state = list (jugs)

next\_state [i] -= poured

next\_state [j] += poured

if get\_all\_states(tuple(next\_state)):

solution\_path.append(state)

return True

return False

print("solving 3-jug problem...\n")

if get\_all\_states(initial\_state):

solution\_path.reverse()

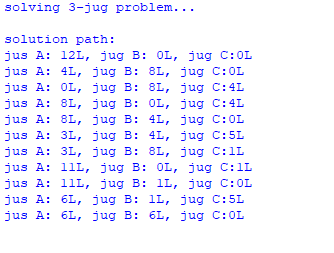
print("solution path:")

for state in solution\_path:

print(f"jus A: {state[0]}L, jug B: {state[1]}L, jug C:{state[2]}L")

else:

print("No solution is found.")



**PRACTICAL 3(B)**

import tkinter as tk

from tkinter import messagebox

root = tk.Tk()

root.title("Tic-Tac-Toe: Player vs AI")

root.resizable(False, False)

board = [""] \* 9

buttons = []

player\_symbol = "X"

ai\_symbol = "O"

game\_over = False

win\_conditions = [

(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)

]

def is\_winner(brd, sym):

return any(brd[a] == brd[b] == brd[c] == sym for a, b, c in win\_conditions)

def is\_draw(brd):

return "" not in brd

def minimax(brd, is\_ai):

if is\_winner(brd, ai\_symbol):

return 1

if is\_winner(brd, player\_symbol):

return -1

if is\_draw(brd):

return 0

scores = []

for i in range(9):

if brd[i] == "":

brd[i] = ai\_symbol if is\_ai else player\_symbol

score = minimax(brd, not is\_ai)

brd[i] = ""

scores.append(score)

return max(scores) if is\_ai else min(scores)

def get\_best\_move():

best\_score = -float('inf')

move = None

for i in range(9):

if board[i] == "":

board[i] = ai\_symbol

score = minimax(board, False)

board[i] = ""

if score > best\_score:

best\_score = score

move = i

return move

def check\_game():

global game\_over

if is\_winner(board, player\_symbol):

status\_label.config(text="You Win!")

messagebox.showinfo("Game Over", "You Win!")

game\_over = True

elif is\_winner(board, ai\_symbol):

status\_label.config(text="AI Wins!")

messagebox.showinfo("Game Over", "AI Wins!")

game\_over = True

elif is\_draw(board):

status\_label.config(text="It's a draw!")

messagebox.showinfo("Game Over", "It's a draw!")

game\_over = True

if game\_over:

for btn in buttons:

btn.config(state="disabled")

def ai\_move():

if not game\_over:

move = get\_best\_move()

if move is not None:

board[move] = ai\_symbol

buttons[move].config(text=ai\_symbol, state="disabled")

check\_game()

if not game\_over:

status\_label.config(text="Your Turn (X)")

def on\_click(i):

global game\_over

if board[i] == "" and not game\_over:

board[i] = player\_symbol

buttons[i].config(text=player\_symbol, state="disabled")

check\_game()

if not game\_over:

status\_label.config(text="AI is thinking...")

root.after(300, ai\_move)

def reset\_game():

global board, game\_over

board = [""] \* 9

game\_over = False

for btn in buttons:

btn.config(text="", state="normal", bg="SystemButtonFace")

status\_label.config(text="Your Turn (X)")

# GUI Setup

frame = tk.Frame(root)

frame.pack(pady=10)

for i in range(9):

btn = tk.Button(frame, text="", width=10, height=3, font=("Arial", 16),

command=lambda i=i: on\_click(i))

btn.grid(row=i//3, column=i%3, padx=5, pady=5)

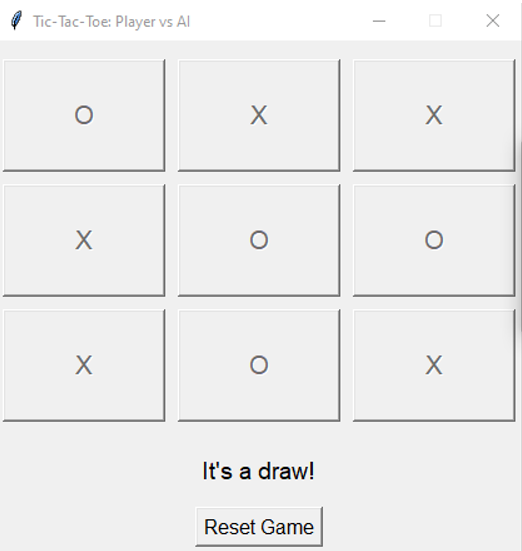
buttons.append(btn)

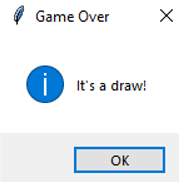
status\_label = tk.Label(root, text="Your Turn (X)", font=("Arial", 14))

status\_label.pack(pady=10)

tk.Button(root, text="Reset Game", font=("Arial", 12), command=reset\_game).pack(pady=5)

root.mainloop()





suits = ['Hearts','Diamonds','Clubs','Spades']

ranks = ['2','3','4','5','6','7','8','9','10',

'Jack','Queen','King','Ace']

deck=[f"{rank} of {suit}" for suit in suits for rank in ranks]

print("Original Deck:")

for card in deck:

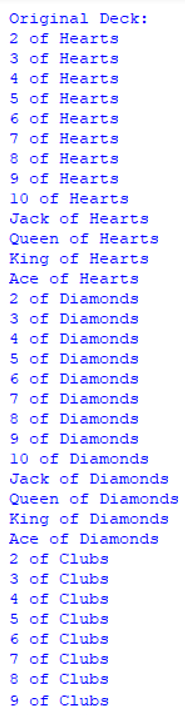
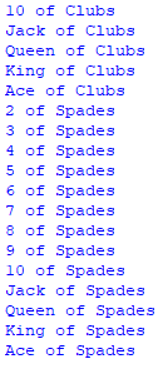
print(card)

random.shuffle(deck)

print("\nShuffled Deck:")

for card in deck:

print(card)



import random

import math

import matplotlib.pyplot as plt

class City:

def \_\_init\_\_(self):

self.x = random.randint(0, 100)

self.y = random.randint(0, 100)

def distance\_to(self, other):

return math.sqrt((self.x - other.x)\*\*2 + (self.y - other.y)\*\*2)

def \_\_repr\_\_(self):

return f"({self.x},{self.y})"

def create\_route(cities):

route = cities[:]

random.shuffle(route)

return route

def route\_distance(route):

distance = 0

for i in range(len(route)):

next\_city = route[(i + 1) % len(route)]

distance += route[i].distance\_to(next\_city)

return distance

def create\_population(cities, size):

return [create\_route(cities) for \_ in range(size)]

def rank\_population(population):

return sorted(population, key=route\_distance)

def select\_parents(population, num\_parents):

return population[:num\_parents]

def crossover(parent1, parent2):

start = random.randint(0, len(parent1) - 2)

end = random.randint(start + 1, len(parent1) - 1)

child = parent1[start:end]

for city in parent2:

if city not in child:

child.append(city)

return child

def mutate(route, rate=0.01):

for i in range(len(route)):

if random.random() < rate:

j = random.randint(0, len(route) - 1)

route[i], route[j] = route[j], route[i]

return route

def next\_generation(population, num\_parents, mutation\_rate):

ranked = rank\_population(population)

parents = select\_parents(ranked, num\_parents)

children = parents[:]

while len(children) < len(population):

p1 = random.choice(parents)

p2 = random.choice(parents)

child = crossover(p1, p2)

child = mutate(child, mutation\_rate)

children.append(child)

return children

def genetic\_algorithm(cities, pop\_size=100, generation=500, elite\_size=10, mutation\_rate=0.02):

population = create\_population(cities, pop\_size)

print("initial best distance:", round(route\_distance(rank\_population(population)[0]), 2))

for \_ in range(generation):

population = next\_generation(population, elite\_size, mutation\_rate)

best = rank\_population(population)[0]

print("Final best distance:", round(route\_distance(best), 2))

return best

def plot\_route(route):

x = [city.x for city in route] + [route[0].x]

y = [city.y for city in route] + [route[0].y]

plt.figure(figsize=(8, 6))

plt.plot(x, y, marker='o', linestyle='-', color='blue')

plt.title("Best TSP Route Found")

plt.xlabel("X Coordinate")

plt.ylabel("Y Coordinate")

for i, city in enumerate(route):

plt.text(city.x + 1, city.y + 1, str(i), fontsize=9)

plt.grid(True)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

num\_cities = 20

cities = [City() for \_ in range(num\_cities)]

best\_route = genetic\_algorithm(cities)

print("\nBest Route:")

for city in best\_route:

print(city)

plot\_route(best\_route)

**PRACTICAL 6(A)**

def alpha\_beta(node, depth, alpha, beta, maximizing\_player):

if depth == 0 or isinstance(node, int):

return node

if maximizing\_player:

max\_eval = float('-inf')

for child in node:

eval = alpha\_beta(child, depth - 1, alpha, beta, False)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = float('inf')

for child in node:

eval = alpha\_beta(child, depth - 1, alpha, beta, True)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

game\_tree = [

[3, 4, 6],

[3, 2, 9],

[0, 1, 4]

]

best\_value = alpha\_beta(game\_tree, depth=3, alpha=float('-inf'), beta=float('inf'), maximizing\_player=True)

print("Best value:", best\_value)



**PRACTICAL 6(B)**

import random

def hill\_climbing(function, x\_start, step\_size, max\_iterations):

current\_x = x\_start

current\_value = function(current\_x)

for i in range(max\_iterations):

neighbors = [current\_x + step\_size, current\_x - step\_size]

neighbor\_values = [function(x) for x in neighbors]

max\_value = max(neighbor\_values)

max\_index = neighbor\_values.index(max\_value)

best\_neighbor = neighbors[max\_index]

if max\_value <= current\_value:

print(f"Iteration {i} : No improvement found. Stopping.")

break

current\_x = best\_neighbor

current\_value = max\_value

print(f"Iteration {i}: x = {current\_x:.4f}, f(x) = {current\_value:.4f}")

return current\_x, current\_value

def objective\_function(x):

return -x\*\*2 + 10\*x

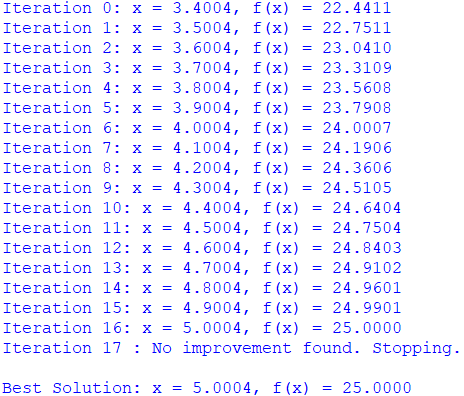
initial\_x = random.uniform(0,10)

step\_size = 0.1

max\_iters = 100

best\_x, best\_value = hill\_climbing(objective\_function, initial\_x,step\_size, max\_iters)

print(f"\nBest Solution: x = {best\_x:.4f}, f(x) = {best\_value:.4f}")



**PRACTICAL 7(A)**

import heapq

def astar(graph, heuristic, start, goal):

open\_list = [(heuristic[start], start, [start], 0)]

visited = set()

while open\_list:

f, node, path, g = heapq.heappop(open\_list)

if node == goal:

return path

if node in visited:

continue

visited.add(node)

for neighbor, cost in graph[node].items():

if neighbor not in visited:

g\_new = g + cost

f\_new = g\_new + heuristic[neighbor]

heapq.heappush(open\_list, (f\_new, neighbor, path + [neighbor], g\_new))

return None

graph = {

'A': {'B': 1, 'C': 3},

'B': {'D': 1, 'E': 5},

'C': {'F': 2},

'D': {'G': 3},

'E': {'G': 1},

'F': {'G': 2},

'G': {}

}

heuristic = {

'A': 7, 'B': 6, 'C': 5,

'D': 4, 'E': 2, 'F': 3,

'G': 0

}

path = astar(graph, heuristic, 'A', 'G')

print("Shortest path:", path)



**PRACTICAL 7(B)**

knowledge\_base = {

"Sachin": "Batsman",

"Batsman": "Cricketer",

"Cricketer": "Sportsperson",

"Sportsperson": "Human",

"Human": "Mammal",

"Mammal": "LivingBeing",

"Dog": "Mammal",

"Cat": "Mammal"

}

def derive\_predicate (kb, entity, target):

"""

Try to derive if an entity is a kind of target using transitive inference

"""

visited = set()

to\_visit = [entity]

while to\_visit:

current = to\_visit.pop()

if current == target:

return True

if current in kb and current not in visited:

visited.add(current)

to\_visit.append(kb[current])

return False

queries = [

("Sachin", "Cricketer"),

("Sachin", "Sportsperson"),

("Sachin", "LivingBeing"),

("Dog", "LivingBeing"),

("Cat", "Sportsperson"),

]

for entity,target in queries:

if derive\_predicate(knowledge\_base,entity,target):

print(f"{entity} is a {target}")

else:

print(f"Cannot derive that {entity} is a {target}")

**PRACTICAL 8(A)**

# Facts: Gender and Parent relationships

male = {"Ram", "Ravi", "Arjun", "Amit"}

female = {"Sita", "Reena", "Maya", "Neha"}

# parent(child, parent)

parent = {

("Ravi", "Ram"), ("Ravi", "Sita"),

("Reena", "Ram"), ("Reena", "Sita"),

("Arjun", "Ravi"), ("Arjun", "Maya"),

("Neha", "Reena"), ("Neha", "Amit")

}

# Predicates and Rules

def father(x, y):

return (x in male) and (y, x) in parent

def mother(x, y):

return (x in female) and (y, x) in parent

def grandfather(x, y):

return (x in male) and any(father(x, p) or mother(x, p) for p, q in parent if q== x and (y, p) in parent)

def grandmother(x, y):

return (x in female) and any(father(x, p) or mother(x, p) for p, q in parent if q== x and (y, p) in parent)

def siblings(x, y):

return x != y and any((x, p) in parent and (y, p) in parent for \_, p in parent)

def brother(x, y):

return (x in male) and siblings(x, y)

def sister(x, y):

return (x in female) and siblings(x, y)

def uncle(x, y):

return (x in male) and any(brother(x, p) for p in male | female if (y, p) in parent)

def aunt(x, y):

return (x in female) and any(sister(x, p) for p in male | female if (y, p) in parent)

def nephew(x, y):

return (x in male) and any((x, p) in parent and (p, y) in parent for p, q in parent)

def niece(x, y):

return (x in female) and any((x, p) in parent and (p, y) in parent for p, q in parent)

def cousin(x, y):

return any(siblings(p1, p2) for (x, p1) in parent for (y, p2) in parent if p1 != p2)

# --- Test cases ---

print("Father of Ravi:", [p for p in male if father(p, "Ravi")])

print("Mother of Arjun:", [p for p in female if mother(p, "Arjun")])

print("Siblings (Ravi, Reena):", siblings("Ravi", "Reena"))

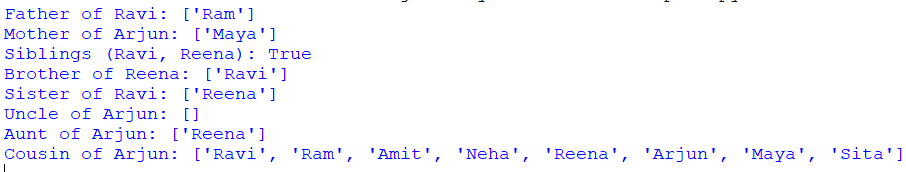
print("Brother of Reena:", [p for p in male if brother(p, "Reena")])

print("Sister of Ravi:", [p for p in female if sister(p, "Ravi")])

print("Uncle of Arjun:", [p for p in male if uncle(p, "Arjun")])

print("Aunt of Arjun:", [p for p in female if aunt(p, "Arjun")])

print("Cousin of Arjun:", [p for p in male | female if cousin(p, "Arjun")])



**PRACTICAL 8B**

**def check\_associative\_addition(a, b, c):**

**return (a + b) + c == a + (b + c)**

**def check\_associative\_multiplication(a, b, c):**

**return (a \* b) \* c == a \* (b \* c)**

**def check\_distributive(a, b, c):**

**return a \* (b + c) == a \* b + a \* c**

**def main():**

**a, b, c = 2, 3, 4 # Example values**

**print("=== Demonstrating Algebraic Laws (Numerical) ===\n")**

**print("Associative Law of Addition:")**

**print(f"({a} + {b}) + {c} = {(a + b) + c}")**

**print(f"{a} + ({b} + {c}) = {a + (b + c)}")**

**print(f"Are they equal? {check\_associative\_addition(a, b, c)}\n")**

**print("Associative Law of Multiplication:")**

**print(f"({a} \* {b}) \* {c} = {(a \* b) \* c}")**

**print(f"{a} \* ({b} \* {c}) = {a \* (b \* c)}")**

**print(f"Are they equal? {check\_associative\_multiplication(a, b, c)}\n")**

**print("Distributive Law:")**

**print(f"{a} \* ({b} + {c}) = {a \* (b + c)}")**

**print(f"{a} \* {b} + {a} \* {c} = {a \* b + a \* c}")**

**print(f"Are they equal? {check\_distributive(a, b, c)}\n")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**