```
graph = {
   'A': {'B','C'},
   'B': {'A','D','E'},
   'C': {'A','F'},
   'D': {'B'},
   'E': {'B','F'},
   'F': {'C','E'}
}

print("Ahmed Shaikh 323")

def dfs(g, n, v=None):
   v = v or []
   if n not in v:
      v.append(n)
      for x in g[n]: dfs(g, x, v)
   return v

print(dfs(graph, 'A'))
```

```
        from collections import deque

        def bfs(start, graph):
            visited, queue, order = set(), deque([start]), []
            while queue:
                node = queue.popleft()
                if node not in visited:
                    visited.add(node)
                    order.append(node)
                    queue.extend(n for n in graph.get(node, []) if n not in visited)
            return order

        print("Ahmed Shaikh 323")

        graph = {
            'A': ['B','C'],
            'B': ['A','D','E'],
            'C': ['A','F'],
            'D': ['B'],
            'E': ['B','F'],
            'F': ['C','E']
        }

print("BFS Order:", bfs('A', graph))
```

```
    def is_safe(b,r,c):
        for i in range(c):
            if b[r][i]==1: return False
        for i,j in zip(range(r,-1,-1),range(c,-1,-1)):
            if b[i][j]==1: return False
        for i,j in zip(range(r,len(b)),range(c,-1,-1)):
            if b[i][j]==1: return False
        return True

    print("Ahmed Shaikh 323")

    def solve(b,c):
        if c>=len(b): return True
        for i in range(len(b)):
            if is_safe(b,i,c):
                b[i][c]=1
                if solve(b,c+1): return True
                b[i][c]=0
        return False

    def print_board(b):
        for r in b: print(" ".join(map(str,r)))

    def solve_4q():
        b=[[0]*4 for _ in range(4)]
        print_board(b) if solve(b,0) else print("No solution exists")

solve_4q()
```

```
        print("Ahmed Shaikh 323")

        def hanoi(n,s,a,t):
            if n==1:
                print(f"Move disk 1 from {s} to {t}"); return
            hanoi(n-1,s,t,a)
            print(f"Move disk {n} from {s} to {t}")
            hanoi(n-1,a,s,t)

hanoi(3,'A','B','C')
```

```
        tree=[[[5,1,2],[8,-8,-9]],[[9,4,5],[-3,4,3]]]
        root=pruned=0

        def children(b,d,a,beta):
            global tree,root,pruned
            for i,c in enumerate(b):
                if isinstance(c,list):
                    na,nb=children(c,d+1,a,beta)
                    if d%2: beta=min(beta,na); b[i]=beta
                    else: a=max(a,nb); b[i]=a
                else:
                    if d%2==0 and a<c: a=c
                    if d%2==1 and beta>c: beta=c
                if a>=beta: pruned+=1; break
            if d==root: tree=a if root==0 else beta
            return a,beta

        print("Ahmed Shaikh 323")

        def alphabeta():
            global tree,pruned
            a,b=children(tree,root,-float('inf'),float('inf'))
            print("(alpha, beta):",a,b)
            print("Result:",tree)
            print("Times pruned:",pruned)
            return a,b,tree,pruned

if __name__=="__main__": alphabeta()
```

```
        def hill(f,x,g):
            x0=x
            while True:
                n=max(g(x0),key=f)
                if f(n)<=f(x0): return x0
                x0=n

        print("Ahmed Shaikh 323")

        def f(x): return -x**2+4*x+10
        def g(x): return [x-0.1,x+0.1]

        best=hill(f,0,g)
        print("Best solution:",best)
print("Maximum value of f(x):",f(best))
```

```
        from collections import defaultdict
        jug1,jug2,aim=4,3,2
        vis=defaultdict(lambda:False)

        def solve(a,b):
            if (a==aim and b==0)or(b==aim and a==0):
                print(a,b);return True
            if not vis[(a,b)]:
                print(a,b);vis[(a,b)]=1
                return (solve(0,b)or solve(a,0)or solve(jug1,b)or solve(a,jug2)or
                        solve(a+min(b,jug1-a),b-min(b,jug1-a))or
                        solve(a-min(a,jug2-b),b+min(a,jug2-b)))
            return False

        print("Ahmed Shaikh 323")
        print("Steps:")
solve(0,0)
```

```
        import itertools,random
        deck=list(itertools.product(range(1,14),['spade','heart','diamond','club']))
        random.shuffle(deck)
        print("Ahmed Shaikh 323\nYou got :")
for i in range(4): print(deck[i][0],"of",deck[i][1])
```

```
import tkinter as tk,random

class Puzzle:
   def __init__(s,r):
      s.r=r; s.n=3
      s.t=list(range(1,s.n*s.n))+[0]; random.shuffle(s.t)
      s.b=[]; [s.mk(i) for i in range(s.n*s.n)]; s.up()

   def mk(s,i):
      b=tk.Button(s.r,text=s.t[i] if s.t[i]!=0 else "",font=("Helvetica",24),width=4,height=2,
            command=lambda i=i:s.mv(i))
      b.grid(row=i//s.n,column=i%s.n); s.b.append(b)

   def up(s):
      for i in range(s.n*s.n):
         t=s.t[i]
         s.b[i].config(text="" if t==0 else str(t),bg="gray" if t==0 else "lightblue")

   def mv(s,i):
      e=s.t.index(0)
      if s.adj(i,e):
         s.t[e],s.t[i]=s.t[i],s.t[e]; s.up()
         if s.ok(): tk.Label(s.r,text="Puzzle
Solved!",font=("Helvetica",24),fg="green").grid(row=s.n,columnspan=s.n)

   def adj(s,i,e):
      r,c=divmod(i,s.n); er,ec=divmod(e,s.n)
      return abs(r-er)+abs(c-ec)==1

   def ok(s): return s.t==list(range(1,s.n*s.n))+[0]

if __name__=="__main__":
r=tk.Tk(); r.title("Number Puzzle"); Puzzle(r); r.mainloop()
```

```
        a = int(input("Enter the first number: "))
        b = int(input("Enter the second number: "))
        c = int(input("Enter the third number: "))

        print("Associative Law")
        print("A + (B + C) =", a + (b + c))
print("(A + B) + C =", (a + b) + c)
```

```
        a = int(input("Enter the first number: "))
        b = int(input("Enter the second number: "))
        c = int(input("Enter the third number: "))
        print("Distributive Law")
        print("A * (B * C) =", a * (b * c))
print("(A * B) * C =", (a * b) * c)
```

correct version

```
        a = int(input("Enter the first number: "))
        b = int(input("Enter the second number: "))
        c = int(input("Enter the third number: "))
        print("Distributive Law")
        print("A * (B + C) =", a * (b + c))
print("(A * B) + (A * C) =", (a * b) + (a * c))
```

```
        # Define the relationships
        sachin_predicate = "batsman"
        batsman_predicate = "cricketer"

        # Derive the predicate
        if sachin_predicate == "batsman":
            derived_predicate = "cricketer"
            result = f"Sachin is {derived_predicate}."
        else:
            result = "No derivation found."

        # Print the result
print(result)
```