```
# Copyright (C) 2025 J V Ramani
# I release this file under the terms of the GNU General Public
# License.

# The programs in this file are distributed in the hope that it
# will be useful, but WITHOUT ANY WARRANTY; without even the implied
# warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

# See the GNU General Public License  for more details at
# http://www.gnu.org/copyleft/gpl.html


# For comments, suggestions and the like, feel free to contact the
# author at
#  ramaniji AT yahoo DOT com


"""




            partial_order.py
            ________________






This module provides functions related to partially ordered
sets and lattices.

Functions:
- cartesian_product(A, B)
- findRelation(A)
- checkReflexive(A, R)
- checkSymmetric(A, R)
- checkAntiSymmetric(A, R)
- checkTransitive(A, R)
- findCoveringRelation(A, R)
- maximalElements(A, R)
- minimalElements(A, R)
- maximalElementsList(A, R)
- minimalElementsList(A, R)
- gElement(A, R)
- sElement(A, R)
- checkUpperBound(A, R, P, e)
- checkLowerBound(A, R, P, e)
- findUpperBounds(A, R, P)
- findLowerBounds(A, R, P)
- lub(A, R, P)
- glb(A, R, P)
- checkLattice(A, R)

Example usage :


    print(cartesian_product([1, 2, 3], [1, 2, 3]))

    print(findRelation([1, 2, 3, 4]))

    A = [1, 2, 3, 4]
    R1 = [[1, 1], [1, 2], [1, 3], [1, 4], [2, 2], [2, 4], [3, 3], [4, 4]]
    result = checkReflexive(A, R1)
    print(result)
```

```
A = [1, 2, 3, 4]
R1 = [[1, 1], [1, 2], [1, 3], [1, 4], [2, 2], [2, 4], [3, 3], [4, 4]]
result = checkSymmetric(A, R1)
print(result)


A = [1, 2, 3, 4]
R1 = [[1, 1], [1, 2], [1, 3], [1, 4], [2, 2], [2, 4], [3, 3], [4, 4]]
result = checkAntiSymmetric(A, R1)
print(result)



A = [1, 2, 3, 4]
R1 = [[1, 1], [1, 2], [1, 3], [1, 4], [2, 2], [2, 4], [3, 3], [4, 4]]
result = checkTransitive(A, R1)
print(result)




A = [1, 2, 3, 4, 6, 8, 12]
R = findRelation(A)
result = findCoveringRelation(A, R)
print(result)


A = [2, 3, 4, 6, 8]
R = findRelation(A)
result = maximalElements(A, R)
print(result)

A = [2, 3, 4, 6, 8]
R = findRelation(A)
result = minimalElements(A, R)
print(result)

A2 = [3, 5, 9, 15, 24, 45]
R2 = findRelation(A2)
result = maximalElementsList(A2, R2)
print(result)
result = minimalElementsList(A2, R2)
print(result)


A = [2, 3, 4, 6, 8]
R = [[8,8],[6,6],[4,8],[4,4],[3,6],[3,3],[2,8],[2,6],[2,4],[2,2]]
result = gElement(A, R)
print(result)

A = [2, 3, 4, 6, 8]
R = [[8,8],[6,6],[4,8],[4,4],[3,6],[3,3],[2,8],[2,6],[2,4],[2,2]]
result = sElement(A, R)
print(result)


A1 = [2, 4, 5, 10, 12, 20, 25]
R1 = findRelation(A1)
P1 = [2, 4]
print(checkUpperBound(A1, R1, P1, 10))
print(checkUpperBound(A1, R1, P1, 12))


A1 = [2, 4, 5, 10, 12, 20, 25]
R1 = findRelation(A1)
```

```
    P1 = [2,  4]
    print(checkLowerBound(A1,   R1,   P1,   5))
    print(checkLowerBound(A1,   R1,   P1,   2))

    A1 = [2,  4,  5,  10,  12,  20,  25]
    R1 = findRelation(A1)
    P1 = [2,  4]
    print(findUpperBounds(A1,   R1,   P1))
    P2 = [2,  5]
    print(findUpperBounds(A1,   R1,   P2))


    A1 = [2,  4,  5,  10,  12,  20,  25]
    R1 = findRelation(A1)
    P1 = [2,  4]
    print(findLowerBounds(A1,   R1,   P1))
    P2 = [2,  5]
    print(findLowerBounds(A1,   R1,   P2))
    P3 = [4,   10,  25]
    print(findLowerBounds(A1,   R1,   P3))
    P4 = [4,   20]
    print(findLowerBounds(A1,   R1,   P4))



    print(lub(A1,  R1,  P1))
    print(lub(A1,  R1,  P2))
    print(lub(A1,  R1,  P3))
    print(lub(A1,  R1,  P4))


    print(glb(A1,  R1,  P1))
    print(glb(A1,  R1,  P2))
    print(glb(A1,  R1,  P3))
    print(glb(A1,  R1,  P4))


    A1 = [1, 2, 3, 4, 5]
    R1 = findRelation(A1)
    print(checkLattice(A1,  R1))


    A2 = [1, 2, 4, 8, 16]
    R2 = findRelation(A2)
    print(checkLattice(A2,  R2))


    A3 = [1, 3, 6, 9, 12]
    R3 = findRelation(A3)
    print(checkLattice(A3,  R3))


    Reference :

    Kenneth H Rosen :: Discrete Mathematics and Its Applications,
                     Eigth Edition, 2019, McGraw-Hill Education.



"""


# finds the cartesian product of two sets
def cartesian_product(A,  B):
    C = [ [a,  b]   for a in A for b in B]
```

```
    return(C)



# finds the relation R = {(a,  b) | a | b where a, b  in A}
# to find other relations change the condition in if( )
def findRelation(A):
    A1 = cartesian_product(A,   A)
    R = []
    for i in range(len(A1)):
        temp = A1[i]
        if((temp[1] % temp[0]) == 0):
            R.append(temp)
    return(R)


# check reflexive
def checkReflexive(A,  R):

    s = 0
    for i in  range(len(A)):
        if([A[i],  A[i]] in R):
            s = s + 1
    if(s == len(A)):
        return(s,   'reflexive')
    else:
        return(s,  'Not reflexive')



# check symmetry
def checkSymmetric(A,  R):

    s = 0
    for i in range(len(R)):
        temp = R[i]
        if([temp[1],  temp[0]] in R):
            s = s + 1
    if(s == len(R)):
        return(s,   'symmetric')
    else:
        return(s,  'Not symmetric')



# check antisymmetry
def checkAntiSymmetric(A,  R):

    s = 0
    for i in range(len(R)):
        temp = R[i]
        if(temp[1] != temp[0]):
            if([temp[1],  temp[0]] in R and [temp[0],  temp[1]] in R):
                s = s + 1
    if(s == 0):
        return(s,   'Anti symmetric')
    else:
        return(s,  'Not Anti symmetric')





# check transitivity
def checkTransitive(A,  R):
```

```
    s = 0
    for i in range(len(A)):
        for j in range(len(A)):
            for k in range(len(A)):
                if([A[i],  A[j]] in R and [A[j],  A[k]] in R):
                    if([A[i],  A[k]] in R):
                        s = 0
                    else:
                        s = 1
                        break
            if(s == 1):
                break
        if(s == 1):
            break
    if(s == 1):
        return(s,   'Not transitive')
    else:
        return(s,   'transitive')






# find the covering relation
def findCoveringRelation(A,   R):

    C = []
    nR = []
    for k in range(len(R)):
        if(R[k][0] != R[k][1]):
            nR.append([R[k][0],  R[k][1]])
    for j in range(len(nR)):
        e = nR[j][0]
        f = nR[j][1]
        s = 0
        for i in range(len(A)):
            if([e,  A[i]] in nR and [A[i],  f] in nR):
                s = s + 1
        if(s == 0):
            C.append([e,  f])
    return(C)



# find maximal elements
def maximalElements(A,   R):

    for k in range(len(A)):
        s = 0
        for i in range(len(A)):
            if( (k != i) and ([A[k],  A[i]] in R)):
                s = s + 1
        if(s == 0):
                print(A[k], " is maximal ")




# find minimal elements
def minimalElements(A,   R):

    for k in range(len(A)):
        s = 0
        for i in range(len(A)):
            if( (k != i) and ([A[i],  A[k]] in R)):
```

```python
                s = s + 1
        if(s == 0):
                print(A[k], " is minimal ")



# in a list
def maximalElementsList(A,   R):

    maxElt = []
    for k in range(len(A)):
        s = 0
        for i in range(len(A)):
            if( (k != i) and ([A[k],  A[i]] in R)):
                s = s + 1
        if(s == 0):
                maxElt.append(A[k])
    return(maxElt)


# in a list
def minimalElementsList(A,   R):

    minElt = []
    for k in range(len(A)):
        s = 0
        for i in range(len(A)):
            if( (k != i) and ([A[i],  A[k]] in R)):
                s = s + 1
        if(s == 0):
                minElt.append(A[k])
    return(minElt)





# find the greatest element
def gElement(A,  R):

    t = 0
    for j in range(len(A)):
        s = 0
        for i in range(len(A)):
            if([A[i],  A[j]] in R):
                s = s + 1
        if(s == len(A)):
            t = j
            break
    if(t == 0):
        return(t,  'no greatest element')
    else:
        return(t,  A[t],  'is the greatest element')



# find the smallest element
def sElement(A,  R):

    t = -1
    for i in range(len(A)):
        s = 0
        for j in range(len(A)):
            if([A[i],  A[j]] in R):
                s = s + 1
        if(s == len(A)):
```

```python
            t = i
            break
    if(t == -1):
        return(t,  'no least element')
    else:
        return(t,  A[t],  'is the least element')



# check for upper bound
def checkUpperBound(A,   R,   P,   e):

    s = 0
    for i in range(len(P)):
        if([P[i],   e] in R):
            s = s + 1
    if(s == len(P)):
        return(True)
    else:
        return(False)



# check for lower bound
def checkLowerBound(A,   R,   P,   e):

    s = 0
    for i in range(len(P)):
        if([e,   P[i]] in R):
            s = s + 1
    if(s == len(P)):
        return(True)
    else:
        return(False)


# find upper bounds
def findUpperBounds(A,   R,   P):

    C = []
    for k in range(len(A)):
        s = 0
        temp = A[k]
        for i in range(len(P)):
            if( [P[i],  temp] in R):
                s = s + 1
        if(s == len(P)):
            C.append(temp)
    return(C)



# find lower bounds
def findLowerBounds(A,   R,   P):

    C = []
    for k in range(len(A)):
        s = 0
        temp = A[k]
        for i in range(len(P)):
            if( [temp,   P[i]] in R):
                s = s + 1
        if(s == len(P)):
            C.append(temp)
    return(C)
```

```python
# find the least upper bound
def lub(A, R, P):

    U = findUpperBounds(A, R, P)
    if(U == []):
        return(U)
    else:
        temp = U[0]
        for i in range(1, len(U)):
            if([U[i], temp] in R):
                temp = U[i]
        return(temp)



# find the greatest lower bound
def glb(A, R, P):

    L = findLowerBounds(A, R, P)
    if(L == []):
        return(L)
    else:
        temp = L[0]
        for i in range(1, len(L)):
            if([temp, L[i]] in R):
                temp = L[i]
        return(temp)



# check for lattice
def checkLattice(A, R):

    s = 0
    for i in range(len(A)):
        for j in range(len(A)):
            if( (lub(A, R, [A[i], A[j]]) != []) and (glb(A, R, [A[i], A[j]]) != []) ):
                s = s + 1
    if(s == (len(A))**2):
        return('Lattice')
    else:
        return('Not a Lattice')
```