RSA ALGORITHM CODE:

```
def gcd(a, b): # calculates GCD of a and d
   while b != 0:
      c = a % b
      a = b
      b = c
   return a

def modinv(a, m): # calculates modulo inverse of a for mod m
   for x in range(1, m):
      if (a * x) % m == 1:
         return x
   return None

def coprimes(a): # calculates all possible co-prime numbers with a
   l = []
   for x in range(2, a):
      if gcd(a, x) == 1 and modinv(x, phi) != None:
         l.append(x)
   for x in l:
      if x == modinv(x, phi):
         l.remove(x)
   return l
def encrypt_block(m): # encrypts a single block
   c = m ** e % n
   return c

def decrypt_block(c): # decrypts a single block
   m = c ** d % n
   return m

def encrypt_string(s): # applies encryption
   return ''.join([chr(encrypt_block(ord(x))) for x in list(s)])

def decrypt_string(s): # applies decryption
   return ''.join([chr(decrypt_block(ord(x))) for x in list(s)])
if __name__ == "__main__":
```

```
p = int(input('Enter prime p: '))
q = int(input('Enter prime q: '))
print("Choosen primes:\np=" + str(p) + ", q=" + str(q) + "\n")
n = p * q
print("n = p * q = " + str(n) + "\n")
phi = (p - 1) * (q - 1)
print("Euler's function (totient) [phi(n)]: " + str(phi) + "\n")
print("Choose an e from a below coprimes array:\n")
print(str(coprimes(phi)) + "\n")
e = int(input())
d = modinv(e, phi) # calculates the decryption key d
print("\nYour public key is a pair of numbers (e=" + str(e) + ", n=" + str(n) +
").\n")
print("Your private key is a pair of numbers (d=" + str(d) + ", n=" + str(n) +
").\n")
s = input("Enter a message to encrypt: ")
print("\nPlain message: " + s + "\n")
enc = encrypt_string(s)
print("Encrypted message: ", enc, "\n")
dec = decrypt_string(enc)
print("Decrypted message: " + dec + "\n")
```

OUTPUT:

```
Enter prime p: 11
Enter prime q: 13
Choosen primes:
p=11, q=13

n = p * q = 143

Euler's function (totient) [phi(n)]: 120

Choose an e from a below coprimes array:

[7, 13, 17, 23, 31, 37, 43, 47, 53, 61, 67, 73, 77, 83, 91, 97, 103, 107, 113]

7

Your public key is a pair of numbers (e=7, n=143).

Your private key is a pair of numbers (d=103, n=143).

Enter a message to encrypt: hello

Plain message: hello

Encrypted message:  [>-

Decrypted message: hello


...Program finished with exit code 0
Press ENTER to exit console.
```

A * ALGORITHM CODE:

```
class Node:
   def __init__(self, data, level, fval):
   # Initialize the node with the data, level of the node and the calculated
fvalue
      self.data = data
      self.level = level
      self.fval = fval
   def generate_child(self):
   # Generate child nodes from the given node by moving the blank space
   # either in the four directions {up,down,left,right}
      x, y = self.find(self.data, '_')
   # val_list contains position values for moving the blank space in either of
   # the 4 directions [up,down,left,right] respectively.
      val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
      children = []
      for i in val_list:
         child = self.shuffle(self.data, x, y, i[0], i[1])
         if child is not None:
            child_node = Node(child, self.level + 1, 0)
            children.append(child_node)
      return children
   def shuffle(self, puz, x1, y1, x2, y2):
   # Move the blank space in the given direction and if the position value are
out
   # of limits the return None
      if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
         temp_puz = []
         temp_puz = self.copy(puz)
         temp = temp_puz[x2][y2]
         temp_puz[x2][y2] = temp_puz[x1][y1]
         temp_puz[x1][y1] = temp
         return temp_puz
      else:
         return None
   def copy(self, root):
   # Copy function to create a similar matrix of the given node
```

```
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp
    def find(self, puz, x):
    # Specifically used to find the position of the blank space
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j
class Puzzle:
    def __init__(self, size):
    # Initialize the puzzle size by the specified size,open and closed lists to empty
        self.n = size
        self.open = []
        self.closed = []
    def accept(self):
    # Accepts the puzzle from the user
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
    def f(self, start, goal):
    # Heuristic Function to calculate hueristic value f(x) = h(x) + g(x)
        return self.h(start.data, goal) + start.level
    def h(self, start, goal):
    # Calculates the different between the given puzzles
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp
```

```
 def process(self):
 # Accept Start and Goal Puzzle state
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()
    start = Node(start, 0, 0)
    start.fval = self.f(start, goal)
    # Put the start node in the open list
    self.open.append(start)
    print("\n\n")
    while True:
       cur = self.open[0]
       print("")
       print("  | ")
       print(" \\\'/\n")
       for i in cur.data:
          for j in i:
             print(j, end=" ")
          print("")
    # If the difference between current and goal node is 0 we have reached
the goal node
       if (self.h(cur.data, goal) == 0):
          break
       for i in cur.generate_child():
          i.fval = self.f(i, goal)
          self.open.append(i)
       self.closed.append(cur)
       del self.open[0]
    # sort the opne list based on f value
    self.open.sort(key=lambda x: x.fval, reverse=False)
puz = Puzzle(3)
puz.process()
```

OUTPUT:

```
Enter the start state matrix

1 2 3
4 _ 6
7 5 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _




  |
 \'/

1 2 3
4 _ 6
7 5 8
```

```
  |
 \'/

1 2 3
4 _ 6
7 5 8

  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
 \'/

1 2 3
4 5 6
7 8 _


...Program finished with exit code 0
Press ENTER to exit console.
```

BFS CODE:

```
from collections import deque


# A class to represent a graph object
class Graph:
    # Constructor
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]

        # add edges to the undirected graph
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)


# Function to perform BFS recursively on the graph
def recursiveBFS(graph, q, discovered):
    if not q:
        return

    # dequeue front node and print it
    v = q.popleft()
    print(v, end=' ')

    # do for every edge (v, u)
    for u in graph.adjList[v]:
        if not discovered[u]:
            # mark it as discovered and enqueue it
            discovered[u] = True
            q.append(u)

    recursiveBFS(graph, q, discovered)


if __name__ == '__main__':

    # List of graph edges as per the above diagram
    #edges = [
        # Notice that node 0 is unconnected
        #(1, 8), (1, 5), (1, 2), (8, 6), (8, 4), (8, 3),
        #(6, 10), (6, 7), (2, 9)
```

```
    #]
    edges = list(tuple(map(int,input().split())) for r in
range(int(input("Enter edges:"))))
    print(edges)

    # total number of nodes in the graph
    #n = 11
    n = int(input("Enter value of n:"))

    # build a graph from the given edges
    graph = Graph(edges, n)

    # to keep track of whether a vertex is discovered or
not
    discovered = [False] * n

    # create a queue for doing BFS
    q = deque()

    # Perform BFS traversal from all undiscovered nodes
    print("\nFollowing is Breadth First Traversal: ")
    for i in range(n):
        if not discovered[i]:
            # mark the source vertex as discovered
            discovered[i] = True

            # enqueue source vertex
            q.append(i)

            # start BFS traversal from vertex i
            recursiveBFS(graph, q, discovered)
```

OUTPUT:

```
Enter edges:9
1 8
1 5
1 2
8 6
8 4
8 3
6 10
6 7
2 9
[(1, 8), (1, 5), (1, 2), (8, 6), (8, 4), (8, 3), (6, 10), (6, 7), (2, 9)]
Enter value of n:11

Following is Breadth First Traversal:
0 1 8 5 2 6 4 3 9 10 7

...Program finished with exit code 0
Press ENTER to exit console.
```

DFS CODE:

```
class Graph:
	# Constructor
	def __init__(self, edges, n):
		self.adjList = [[] for _ in range(n)]

		# add edges to the undirected graph
		for (src, dest) in edges:
			self.adjList[src].append(dest)
			self.adjList[dest].append(src)


# Function to perform DFS recursively on the graph
def recursive_DFS(graph, v, discovered):

	discovered[v] = True			# mark the current node as
discovered
	print(v, end=' ')			# print the current node

	# do for every edge (v, u)
	for u in graph.adjList[v]:
		if not discovered[u]:	# if `u` is not yet discovered
			recursive_DFS(graph, u, discovered)


if __name__ == '__main__':

	# List of graph edges as per the above diagram
	edges = list(tuple(map(int, input().split())) for r in range(int(input("Enter
edges:"))))
	print(edges)
	#edges = [
		# Notice that node 0 is unconnected
		#(1, 2), (1, 3), (2, 4), (2, 5), (4, 6), (6, 7), (3, 5), (5, 6)
	#]

	# total number of nodes in the graph
```

```
#n = 8
n = int(input("Enter value of n:"))

graph = Graph(edges, n)

# to keep track of whether a vertex is discovered or not
discovered = [False] * n

# Perform DFS traversal from all undiscovered nodes
print("\nFollowing is Depth First Traversal: ")
for i in range(n):
	if not discovered[i]:
		recursive_DFS(graph, i, discovered)
```

OUTPUT:

```
Enter edges:8
1 2
1 3
2 4
2 5
4 6
6 7
3 5
5 6
[(1, 2), (1, 3), (2, 4), (2, 5), (4, 6), (6, 7), (3, 5), (5, 6)]
Enter value of n:8

Following is Depth First Traversal:
0 1 2 4 6 7 5 3

...Program finished with exit code 0
Press ENTER to exit console.
```

TRANSPOSITION TECHNIQUE

```
import math

key = "HACK"

# Encryption
def encryptMessage(msg):
    cipher = ""

    # track key indices
    k_indx = 0

    msg_len = float(len(msg))
    msg_lst = list(msg)
    key_lst = sorted(list(key))

    # calculate column of the matrix
    col = len(key)

    # calculate maximum row of the matrix
    row = int(math.ceil(msg_len / col))
    fill_null = int((row * col) - msg_len)
    msg_lst.extend('_' * fill_null)

    # create Matrix and insert message
    matrix = [msg_lst[i: i + col]
              for i in range(0, len(msg_lst), col)]

    # read matrix column-wise using key
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        cipher += ''.join([row[curr_idx]
                           for row in matrix])
        k_indx += 1

    return cipher
```

```
# Decryption
def decryptMessage(cipher):
    msg = ""

    # track key indices
    k_indx = 0

    # track msg indices
    msg_indx = 0
    msg_len = float(len(cipher))
    msg_lst = list(cipher)

    # calculate column of the matrix
    col = len(key)

    # calculate maximum row of the matrix
    row = int(math.ceil(msg_len / col))

    # convert key into list and sort
    # alphabetically so we can access
    # each character by its alphabetical position.
    key_lst = sorted(list(key))

    # create an empty matrix to
    # store deciphered message
    dec_cipher = []
    for _ in range(row):
        dec_cipher += [[None] * col]

    # Arrange the matrix column wise according
    # to permutation order by adding into new matrix
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])

        for j in range(row):
            dec_cipher[j][curr_idx] = msg_lst[msg_indx]
```

```
            msg_indx += 1
        k_indx += 1

    # convert decrypted msg matrix into a string
    try:
        msg = ''.join(sum(dec_cipher, []))
    except TypeError:
        raise TypeError("This program cannot",
                        "handle repeating words.")

    null_count = msg.count('_')

    if null_count > 0:
        return msg[: -null_count]

    return msg


# Driver Code
msg = input("Enter your Message: ")

cipher = encryptMessage(msg)
print("Encrypted Message: {}".format(cipher))

print("Decryped Message: {}".format(decryptMessage(cipher)))
```

OUTPUT:

```
Enter your Message: Hello
Encrypted Message: e_l_Hol_
Decryped Message: Hello


...Program finished with exit code 0
Press ENTER to exit console.
```

N QUEENS PROBLEM

CODE

```
public class NQueenProblem {
	final int N = 4;

	/* A utility function to print solution */
	void printSolution(int board[][])
	{
		for (int i = 0; i < N; i++) {
			for (int j = 0; j < N; j++)
				System.out.print(" " + board[i][j]
								+ " ");
			System.out.println();
		}
	}

	/* A utility function to check if a queen can
	be placed on board[row][col]. Note that this
	function is called when "col" queens are already
	placed in columns from 0 to col -1. So we need
	to check only left side for attacking queens */
	boolean isSafe(int board[][], int row, int col)
	{
		int i, j;
```

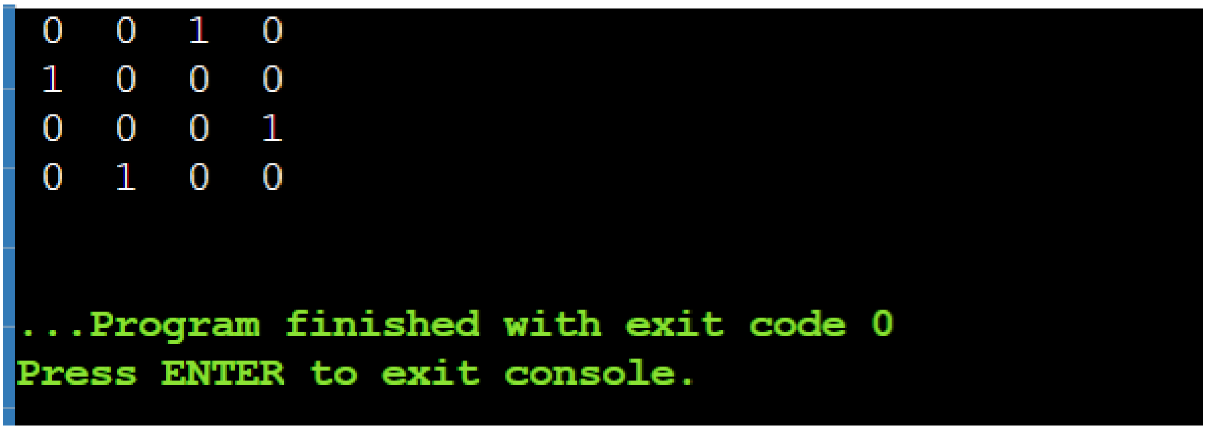
```
	/* Check this row on left side */
	for (i = 0; i < col; i++)
		if (board[row][i] == 1)
			return false;

	/* Check upper diagonal on left side */
	for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
		if (board[i][j] == 1)
			return false;

	/* Check lower diagonal on left side */
	for (i = row, j = col; j >= 0 && i < N; i++, j--)
		if (board[i][j] == 1)
			return false;

	return true;
}

/* A recursive utility function to solve N
Queen problem */
boolean solveNQUtil(int board[][], int col)
{
	/* base case: If all queens are placed
	then return true */
	if (col >= N)
		return true;
```

```
        /* Consider this column and try placing
        this queen in all rows one by one */
        for (int i = 0; i < N; i++) {
            /* Check if the queen can be placed on
            board[i][col] */
            if (isSafe(board, i, col)) {
                /* Place this queen in board[i][col] */
                board[i][col] = 1;

                /* recur to place rest of the queens */
                if (solveNQUtil(board, col + 1) == true)
                    return true;

                /* If placing queen in board[i][col]
                doesn't lead to a solution then
                remove queen from board[i][col] */
                board[i][col] = 0; // BACKTRACK
            }
        }

        /* If the queen can not be placed in any row in
        this column col, then return false */
        return false;
    }

    /* This function solves the N Queen problem using
    Backtracking. It mainly uses solveNQUtil () to
```

```
solve the problem. It returns false if queens
cannot be placed, otherwise, return true and
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.*/
boolean solveNQ()
{
	int board[][] = { { 0, 0, 0, 0 },
					{ 0, 0, 0, 0 },
					{ 0, 0, 0, 0 },
					{ 0, 0, 0, 0 } };

	if (solveNQUtil(board, 0) == false) {
		System.out.print("Solution does not exist");
		return false;
	}

	printSolution(board);
	return true;
}

// driver program to test above function
public static void main(String args[])
{
	NQueenProblem Queen = new NQueenProblem();
	Queen.solveNQ();
```

```
        }
}
```

CODE OUTPUT



```
 0   0   1   0
 1   0   0   0
 0   0   0   1
 0   1   0   0


...Program finished with exit code 0
Press ENTER to exit console.
```