BFS

```
graph = {'A':['B', 'E', 'C'],
         'B':['A', 'D', 'E'],
         'D':['B', 'E'],
         'E':['A', 'D', 'B'],
         'C':['A', 'F', 'G'],
         'F':['C'],
         'G':['C']
         }
visited = []
queue = []

def bfs(visited, graph, start_node, goal_node):
   visited.append(start_node)
   queue.append(start_node)
   while queue:
      m = queue.pop(0)
      print(m)
      if m == goal_node:
         print("Node is Found !!! ")
         break
      else:
         for n in graph[m]:
            if n not in visited:
               visited.append(n)
               queue.append(n)

print("The BFS Traversal is : ")
bfs(visited, graph, 'A', 'D')
```

DFS

```
# graph = {
#    '0': ['1','3','4'],
#    '1': ['2'],
#    '2': [],
#    '3': ['5'],
#    '4': ['5'],
#    '5': []
# }

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

vis = set()

def dfs(vis, graph, node):
    if node not in vis:
        print(node, end=" ")
        vis.add(node)
        for adj in graph[node]:
            dfs(vis, graph, adj)

print("Following is the Depth-First Search")
dfs(vis, graph, 'A')

visited = []
queue = []

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        m = queue.pop(0)
        print(m, end=" ")
        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

print("\nFollowing is the Breadth-First Search")
bfs(visited, graph, 'A')
```

A*

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return None
        open_set.remove(n)
        closed_set.add(n)

def get_neighbors(v):
    if v in Graph_nodes:
```

```
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }
    return H_dist[n]

Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}

aStarAlgo('A', 'G')
```

N-Q

```
N = 8

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()

def isSafe(row, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):
    if (slashCodeLookup[slashCode[row][col]] or
            backslashCodeLookup[backslashCode[row][col]] or
            rowLookup[row]):
        return False
    return True

def solveNQueensUtil(board, col, slashCode, backslashCode, rowLookup, slashCodeLookup,
backslashCodeLookup):
    if col >= N:
        return True
```

```
    for i in range(N):
        if isSafe(i, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):
            board[i][col] = 1
            rowLookup[i] = True
            slashCodeLookup[slashCode[i][col]] = True
            backslashCodeLookup[backslashCode[i][col]] = True

            if solveNQueensUtil(board, col + 1, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):
                return True

            board[i][col] = 0
            rowLookup[i] = False
            slashCodeLookup[slashCode[i][col]] = False
            backslashCodeLookup[backslashCode[i][col]] = False

    return False

def solveNQueens():
    board = [[0 for i in range(N)] for j in range(N)]
    slashCode = [[0 for i in range(N)] for j in range(N)]
    backslashCode = [[0 for i in range(N)] for j in range(N)]
    rowLookup = [False] * N
    x = 2 * N - 1
    slashCodeLookup = [False] * x
    backslashCodeLookup = [False] * x

    for rr in range(N):
        for cc in range(N):
            slashCode[rr][cc] = rr + cc
            backslashCode[rr][cc] = rr - cc + 7

    if solveNQueensUtil(board, 0, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True

solveNQueens()
```

Pims

```
import sys
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

    def minKey(self, key, mstSet):
        min = sys.maxsize
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v
        return min_index

    def primMST(self):
        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = 0
        mstSet = [False] * self.V
        parent[0] = -1

        for cout in range(self.V):
            u = self.minKey(key, mstSet)
            mstSet[u] = True

            for v in range(self.V):
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u
        self.printMST(parent)
# Driver's code
if __name__ == '__main__':
    g = Graph(5)
    g.graph = [
        [0, 2, 0, 6, 0],
        [2, 0, 3, 8, 5],
        [0, 3, 0, 0, 7],
        [6, 8, 0, 0, 9],
        [0, 5, 7, 9, 0]
    g.primMST()
```

Transposition

```
import math
plaintext="transposition technique using python"
key=8
ciphertext=['']*key
for colum in range(key):
 pointer=colum
 while pointer<len(plaintext):
 ciphertext[colum]+=plaintext[pointer]
 # print(ciphertext)
 pointer+=key
cipher=' '.join(ciphertext)
print(cipher)
nC = math.ceil(len(cipher) / key)
print(nC )
nR = key
numOfShadedBoxes = (nC * nR) - len(cipher)
pt = [''] * nC
col=0
row=0
for sym in cipher:
 pt[col]+=sym
 col+=1
 if (col == nC) or (col == nC - 1 and row >= nR- numOfShadedBoxes):
 col=0
 row=row+1
print(''.join(pt))
```

DES

```
from Crypto.Cipher import DES
def pad(text):
 n = len(text) % 8
 print(b"text to encrypt:"+text + (b' ' * n))
 return text + (b' ' * n)
key = b'hello123'
text1 = b'Python is the Best Language!'
des = DES.new(key, DES.MODE_ECB)
padded_text = pad(text1)
encrypted_text = des.encrypt(padded_text)
print(encrypted_text)
print(des.decrypt(encrypted_text)
```

AES

```
# importing AES
from Crypto.Cipher import AES
# encryption key
#key = b'C&F)H@McQfTjWnZr'
key= b'1234455fghdhdfrs'
# create new instance of cipher
cipher = AES.new(key, AES.MODE_EAX)
# data to be encrypted
data = "This is experiment 4 AES".encode()
# nonce is a random value generated each time we instantiate the cipher using new()
nonce = cipher.nonce
# encrypt the data
ciphertext = cipher.encrypt(data)
# print the encrypted data
print("Cipher text:", ciphertext)
# generate new instance with the key and nonce same as encryption cipher
cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
# decrypt the data
plaintext = cipher.decrypt(ciphertext)
print("Plain text:", plaintext)
```

RSA

```
import random
def gcd(a, b):
 while b != 0:
 a, b = b, a % b
 return a
def multiplicative_inverse(e, phi):
 d = 0
 x1 = 0
 x2 = 1
 y1 = 1
 temp_phi = phi
 while e > 0:
 temp1 = temp_phi // e
 temp2 = temp_phi - temp1 * e
 temp_phi = e
 e = temp2
 x = x2 - temp1 * x1
 y = d - temp1 * y1
 x2 = x1
 x1 = x
 d = y1
 y1 = y
```

```
 if temp_phi == 1:
 return d + phi
def generate_keypair(p, q):
 n = p * q
 phi = (p-1) * (q-1)
 e = random.randrange(1, phi)
 g = gcd(e, phi)
 while g != 1:
 e = random.randrange(1, phi)
 g = gcd(e, phi)
 d = multiplicative_inverse(e, phi)
 return ((e, n), (d, n))
def encrypt(pk, plaintext):
 key, n = pk
 cipher = [(ord(char) ** key) % n for char in plaintext]
 return cipher
def decrypt(pk, ciphertext):
 key, n = pk
 plain = [chr((char ** key) % n) for char in ciphertext]
 return ''.join(plain)
if __name__ == '__main__':
 p = int(input("Enter a prime number (p): "))
 q = int(input("Enter another prime number (q): "))
 public, private = generate_keypair(p, q)
 print("Public key: ", public)
 print("Private key: ", private)
 message = input("Enter a message to encrypt: ")
 encrypted_message = encrypt(public, message)
 print("Encrypted message: ", ''.join(map(lambda x: str(x), encrypted_message)))
 decrypted_message = decrypt(private, encrypted_message)
 print("Decrypted message: ", decrypted_message)
```

Hellman

```
import random

def main():
    # Step 1: Agree on a prime number (p) and a base number (g)
    p = get_prime(128)     #128, 192, 256, 512...
    g = 2

    # Step 2: Alice generates a random number (a) and computes A = g^a mod p
    a = random.randint(2, p-2)
    A = pow(g, a, p)
    ##print("A : ",A)

    # Step 3: Bob generates a random number (b) and computes B = g^b mod p
```

```
    b = random.randint(2, p-2)
    B = pow(g, b, p)
    ##print("B : ",B)

    # Step 4: Both Alice and Bob compute the shared secret key (K)
    K1 = pow(B, a, p)
    ##print("K1 :",K1)
    K2 = pow(A, b, p)
    ##print("K2 :",K2)

    # Check if both keys are the same
    if K1 == K2:
        print("Shared Secret Key: " + hex(K1)[2:])
    else:
        print("Error: Keys do not match")

def get_prime(bits):
    while True:
        p = random.getrandbits(bits)
        if is_prime(p):
            return p

def is_prime(n, k=20):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Write n as 2^r * d + 1
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Witness loop
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
```

```python
    return True

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