TW2: RSA Algorithm

**Instructions:** (Put Instruction)

|  |
| --- |
| import random  import math  # Function to check if a number is prime  def is\_prime(num):  if num <= 1:  return False  for i in range(2, int(math.sqrt(num)) + 1):  if num % i == 0:  return False  return True  # Function to generate random prime numbers  def generate\_prime(bits):  while True:  num = random.getrandbits(bits)  if is\_prime(num):  return num  # Function to compute the greatest common divisor (GCD)  def gcd(a, b):  while b:  a, b = b, a % b  return a |
| # Function to find the modular multiplicative inverse  def mod\_inverse(a, m):  m0, x0, x1 = m, 0, 1  while a > 1:  q = a // m  m, a = a % m, m  x0, x1 = x1 - q \* x0, x0  return x1 + m0 if x1 < 0 else x1  # Function to generate RSA key pairs  def generate\_key\_pair(bits):  p = generate\_prime(bits)  q = generate\_prime(bits)  n = p \* q  phi = (p - 1) \* (q - 1)  while True:  e = random.randint(2, phi - 1)  if gcd(e, phi) == 1:  break  d = mod\_inverse(e, phi)  public\_key = (n, e)  private\_key = (n, d)  return public\_key, private\_key  # Function to encrypt a message  def encrypt(public\_key, message):  n, e = public\_key  cipher\_text = [pow(ord(char), e, n) for char in message]  return cipher\_text |
| # Function to decrypt a message  def decrypt(private\_key, cipher\_text):  n, d = private\_key  decrypted\_message = ''.join([chr(pow(char, d, n)) for char in cipher\_text])  return decrypted\_message  # Main program  if \_\_name\_\_ == "\_\_main\_\_":  bits = 8 # Adjust the number of bits for your desired security level  public\_key, private\_key = generate\_key\_pair(bits)  print(f" Generated Public Key : {public\_key} \n Generated Private Key : {private\_key}")  message = eval(input(" Enter the Message to be Encrypted : "))  print(" Original message:", message)  encrypted\_message = encrypt(public\_key, message)  print(" Encrypted message:", encrypted\_message)  decrypted\_message = decrypt(private\_key, encrypted\_message)  print(" Decrypted message:", decrypted\_message) |
|  |

TW3: UDP Connectionless Transport

**Instructions:** (Put Instruction)

|  |
| --- |
| **#UDP Server Code**  import socket  # Create a UDP socket  server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  # Bind the socket to an address and port  server\_address = ('localhost', 12345)  server\_socket.bind(server\_address)  print('UDP server is waiting for messages...')  while True:  # Receive data from the client  data, client\_address = server\_socket.recvfrom(1024)  print(f'Received message from {client\_address}: {data.decode()}')  # Close the socket (this will never be reached in this example)  server\_socket.close() |
| **UDP Client Code**  import socket  # Create a UDP socket  client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  # Server address and port  server\_address = ('localhost', 12345)  while True:  message = input('Enter a message: ')  # Send data to the server  client\_socket.sendto(message.encode(), server\_address)  # Close the socket (this will never be reached in this example)  client\_socket.close() |

**Note: A separate Terminals need to be opened for Server and Client**

|  |
| --- |
|  |
|  |

TW4: TCP Connection-Oriented Transport

**Instructions:** (Put Instruction)

|  |
| --- |
| **# TCP Server Code**  import socket  # Define the server address and port  server\_address = ('127.0.0.1', 12345)  # Create a socket object  server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  # Bind the socket to the server address and port  server\_socket.bind(server\_address)  # Listen for incoming connections  server\_socket.listen(4)  print("Server is listening for incoming connections...")  # Accept a connection  client\_socket, client\_address = server\_socket.accept()  print(f"Connected to {client\_address}")  # Receive data from the client  data = client\_socket.recv(1024)  print(f"Received data from client: {data.decode('utf-8')}")  # Send a response to the client  response = "Hello, client!"  client\_socket.send(response.encode('utf-8'))  # Close the sockets  client\_socket.close()  server\_socket.close() |
| **# TCP Client Code**  import socket  # Define the server address and port  server\_address = ('127.0.0.1', 12345)  # Create a socket object  client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  # Connect to the server  client\_socket.connect(server\_address)  # Send a message to the server  message = "Hello, server!"  client\_socket.send(message.encode('utf-8'))  # Receive a response from the server  data = client\_socket.recv(1024)  print(f"Received response from server: {data.decode('utf-8')}")  # Close the socket  client\_socket.close() |

**Note: A separate Terminals need to be opened for Server and Client**

|  |
| --- |
|  |
|  |

Tw5

# DVR

import sys

class Network:

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

self.nodes = nodes

self.graph = {} # Dictionary to store network topology

self.distance\_vector = {} # Dictionary to store distance vectors

def add\_link(self, node1, node2, cost):

# Add a link between two nodes with a given cost

if node1 not in self.graph:

self.graph[node1] = {}

self.graph[node1][node2] = cost

if node2 not in self.graph:

self.graph[node2] = {}

self.graph[node2][node1] = cost

def initialize\_distance\_vector(self, node):

# Initialize the distance vector for a node

self.distance\_vector[node] = {node: 0}

for n in self.nodes:

if n != node:

self.distance\_vector[node][n] = sys.maxsize

def update\_distance\_vector(self, node):

# Update the distance vector for a node

for dest in self.nodes:

if dest != node:

min\_cost = sys.maxsize

for neighbor in self.graph[node]:

if dest in self.distance\_vector[neighbor]:

cost = self.distance\_vector[neighbor][dest] + self.graph[node][neighbor]

if cost < min\_cost:

min\_cost = cost

self.distance\_vector[node][dest] = min\_cost

def print\_routing\_table(self, node):

# Print the routing table for a node

print(f"Routing table for Node {node}:")

print("Destination\tCost")

for dest, cost in self.distance\_vector[node].items():

if dest != node:

print(f"{dest}\t\t{cost}")

print()

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

nodes = [1, 2, 3, 4, 5]

network = Network(nodes)

network.add\_link(1, 2, 2)

network.add\_link(1, 3, 2)

network.add\_link(1, 4, 1)

network.add\_link(2, 3, 1)

network.add\_link(2, 5, 1)

network.add\_link(3, 4, 1)

network.add\_link(3, 5, 1)

for node in nodes:

network.initialize\_distance\_vector(node)

num\_iterations = 6 # Number of iterations to update the distance vectors

for \_ in range(num\_iterations):

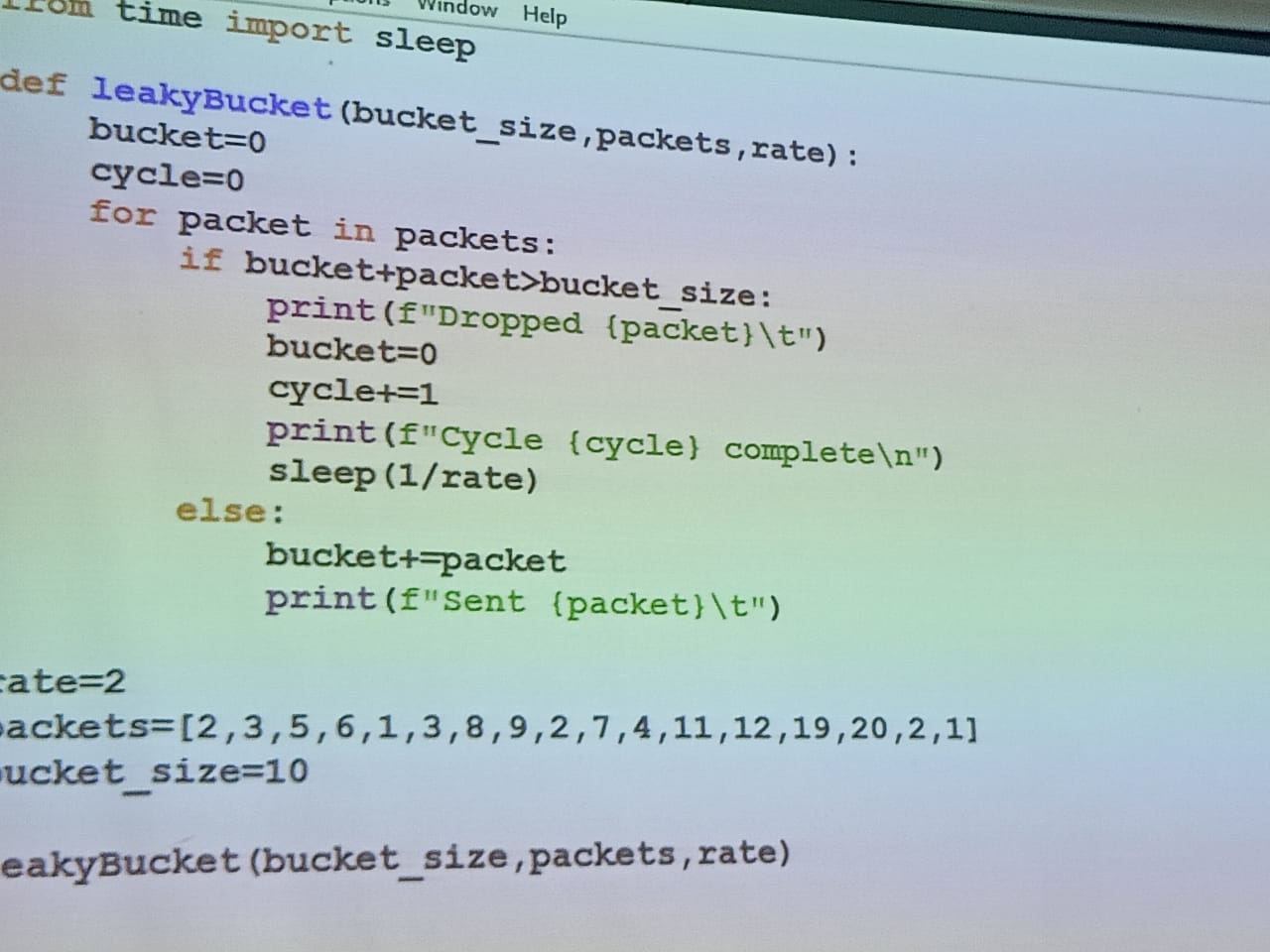
for node in nodes:

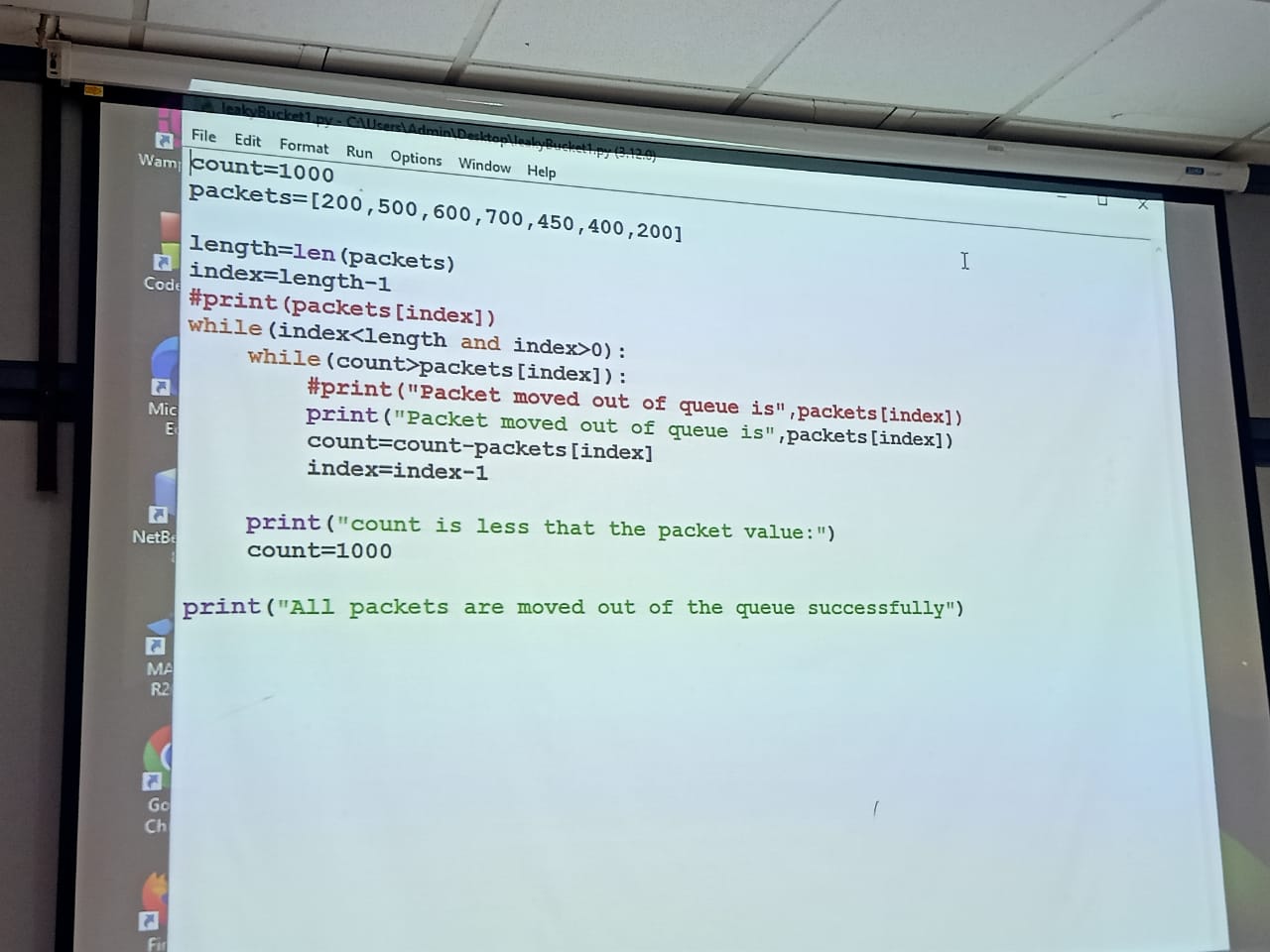
network.update\_distance\_vector(node)

for node in nodes:

network.print\_routing\_table(node)

Tw6





Tw7