exp-7(Implementation and analysis of Distance vector routing)

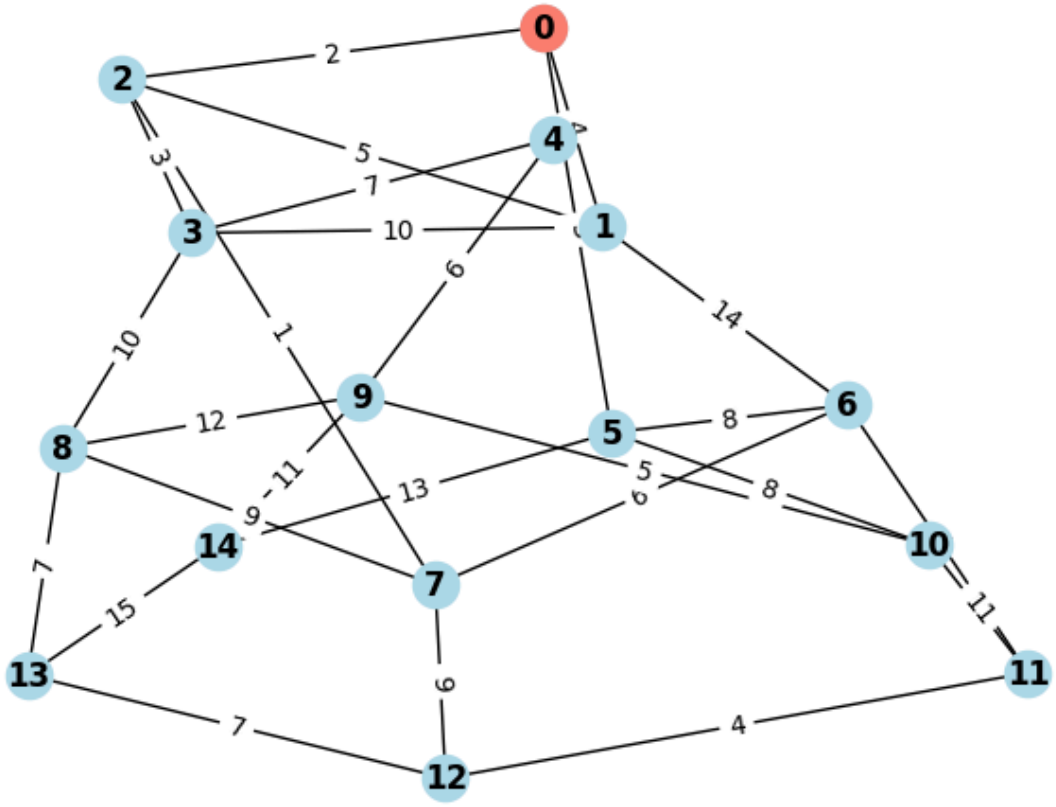
```
import networkx as nx
import matplotlib.pyplot as plt
from colorama import Fore, Style

def BellmanFord(edges, V, src):

  dist = [float("Inf")] * V   #Initialize distance from src to vertices as
INFINITE
  dist[src] = 0

  for _ in range(V): # Relax all edges |V| - 1 times. A shortest path can
have at-most |V| - 1 edges
    for u, v, w in edges:
      if dist[u] != float("Inf") and dist[u] + w < dist[v]: # Consider only
those vertices in queue
        dist[v] = dist[u] + w  # Update dist & parent index of adj vertices
of picked vertex.

      if dist[v] != float("Inf") and dist[v] + w < dist[u]: # Interchange u &
v - undirected graph
        dist[u] = dist[v] + w
      #print(u,v,w,"\n",dist)

  # Check for negative-weight cycles. The above step guarantees shortest
distances if graph doesn't contain
  # negative weight cycle. If we get a shorter path, then there is a cycle. -
Directed graph
  for u, v, w in edges:
    if dist[u] != float("Inf") and dist[u] + w < dist[v]:
      print("Graph contains negative weight cycle")
      return

  return dist

def plot(edges,src):
  G = nx.Graph()
  G.add_weighted_edges_from(edges)
  fig = plt.figure()
```

```
  pos = nx.spring_layout(G, seed= 42)
  nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue')
  nx.draw_networkx_nodes(G, pos, nodelist=[src], node_color='salmon') #
Highlight the current node
  edge_labels = {(i, j): G[i][j]['weight'] for i, j in G.edges()} # Draw edge
labels
  nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

# Inputs
V = 15
edges = [(0, 1, 4), (0, 2, 2), (1, 2, 5),
         (1, 3, 10), (2, 3, 3), (3, 4, 7),
         (5, 6, 8), (6, 7, 6), (7, 8, 9),
         (8, 9, 12), (9, 10, 5), (10, 11, 11),
         (11, 12, 4), (12, 13, 7), (13, 14, 15),
         (14, 5, 13), (0, 5, 3), (1, 6, 14),
         (2, 7, 1), (3, 8, 10), (4, 9, 6),
         (5, 10, 8), (6, 11, 5), (7, 12, 9),
         (8, 13, 7), (9, 14, 11)]
src = 0

plot (edges, src)
dist = BellmanFord(edges, V, src) # Find shortest distance from src to
vertices using Bellman-Ford algorithm

print(f"\nNode\t\tShortest distance from {src}") #Print
for i in range(V):
  print(Fore.GREEN+f"{i}\t\t{dist[i]}"+Style.RESET_ALL)

#plt.tight_layout()
plt.show()
ArithmeticError
```

Output:

```
Node        Shortest distance from 0
0           0
1           4
```

```
2           2
3           5
4           12
5           3
6           9
7           3
8           12
9           16
10          11
11          14
12          12
13          19
14          16
```



exp-8(Implementation and analysis of Link State Routing)

```
import networkx as nx
import matplotlib.pyplot as plt
import heapq
from colorama import Fore, Style
```

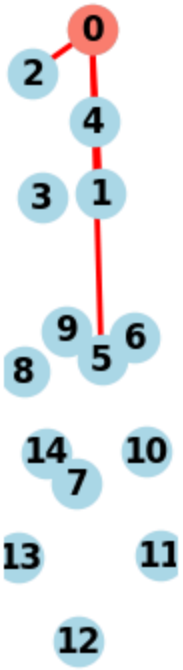
```
def dijkstra(G, source):
    distances = {node: float('infinity') for node in G}     #
Initialisations
    predecessors = {node: None for node in G}
    distances[source] = 0  # Set the distance to the starting node
as 0
    priority_queue = [(0, source)]  # 1. Priority queue to track
nodes and their distances
    subplot_position = 0

    while priority_queue: # 2. Explore nodes in the priority queue
until it's empty
        current_distance, current_node =
heapq.heappop(priority_queue)
        for neighbor, weight in G[current_node].items(): # 3:
Explore neighbors of the current node
            distance = current_distance + weight['weight']
            if distance < distances[neighbor]: #4: Update distance
if shorter path is found
                distances[neighbor] = distance
                predecessors[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))
# Add to the priority queue

        show_step(predecessors, distances, current_node,
priority_queue) # Visualize current state
        subplot_position += 1
        if subplot_position == 1:
            fig = plt.figure()
        draw_step(G, predecessors, current_node, fig,
subplot_position)
        if subplot_position  == 5:
            subplot_position = 0

    return distances, predecessors

def draw_step(G, predecessors, current_node,fig, subplot_position):
    pos = nx.spring_layout(G, seed=42)
    fig.add_subplot(1, 5, subplot_position)  # Adjust the subplot
grid as needed
```

```
    nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue', edge_color='white')
    for node, pred in predecessors.items(): # Highlight the edges in
the shortest path
        if pred is not None:
            nx.draw_networkx_edges(G, pos, edgelist=[(pred, node)],
edge_color='red', width=2)
    nx.draw_networkx_nodes(G, pos, nodelist=[current_node],
node_color='salmon')#Highlight current node
    plt.title(f"Step: {current_node}")

def show_step(predecessors, distances, current_node,
priority_queue):
    print(f"Current Node {current_node}: ")
    print(Fore.BLUE+ f"Distances: {distances} \nPredecessors:
{predecessors}" + Style.RESET_ALL)
    print(Fore.YELLOW + f"Updated Elements: {priority_queue}" +
Style.RESET_ALL)
    print(Fore.RED + f"Shortest distance from {source} to
{current_node}: {distances[current_node]}" +Style.RESET_ALL)
    path = []
    node = current_node
    while node is not None:
        path.insert(0, node)
        node = predecessors[node]
    print(Fore.RED + "Path:", " -> ".join(path),"\n" +
Style.RESET_ALL)

def plot(G):
    fig = plt.figure()     # Plot
    pos = nx.spring_layout(G, seed= 42)
    nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue')
    nx.draw_networkx_nodes(G, pos, nodelist=[source],
node_color='salmon') # Highlight the current node
    edge_labels = {(i, j): G[i][j]['weight'] for i, j in G.edges()}
# Draw edge labels
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

#Inputs
```

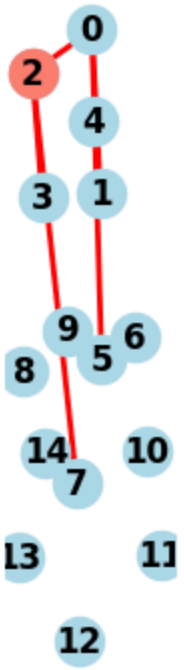
```
edges = [('0', '1', 4), ('0', '2', 2), ('1', '2', 5), ('1', '3',
10),
         ('2', '3', 3), ('3', '4', 7), ('5', '6', 8), ('6', '7', 6),
         ('7', '8', 9), ('8', '9', 12), ('9', '10', 5), ('10', '11',
11),
         ('11', '12', 4), ('12', '13', 7), ('13', '14', 15), ('14',
'5', 13),
         ('0', '5', 3), ('1', '6', 14), ('2', '7', 1), ('3', '8',
10),
         ('4', '9', 6), ('5', '10', 8), ('6', '11', 5), ('7', '12',
9),
         ('8', '13', 7), ('9', '14', 11)]
source = '0'

G = nx.Graph() # Example graph
G.add_weighted_edges_from(edges)

plot(G) #Plot

distances, predecessors = dijkstra(G, source)  # Run Dijkstra's
algorithm

print(f"\nNode\t\tPedecessor\tShortest distance from {source}\n") #
Print Distance and predecessor
for i,j in distances.items():
    print(Fore.GREEN + f"  {i}\t\t{predecessors[i]}\t\t{j}" +
Style.RESET_ALL)

#plt.tight_layout()
plt.show()
ArithmeticError
```

Output:

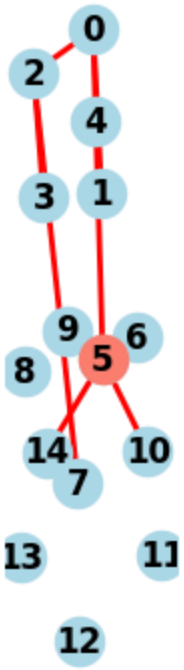
Step: 0



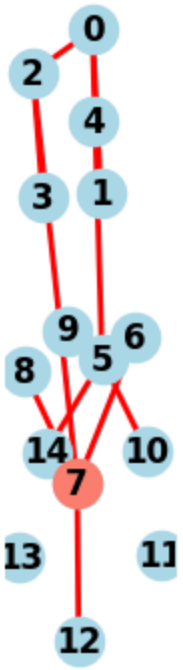
Step: 2



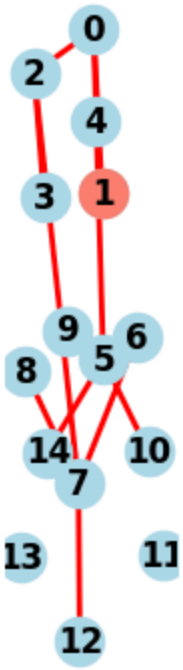
Step: 5



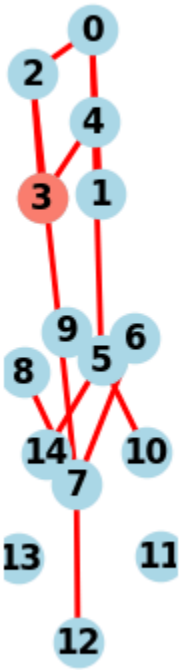
Step: 7



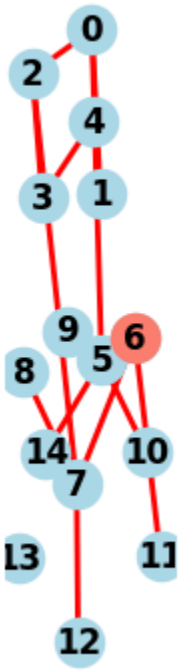
Step: 1



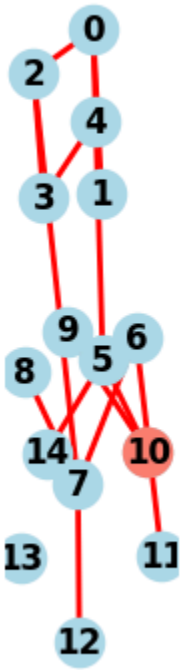
Step: 3



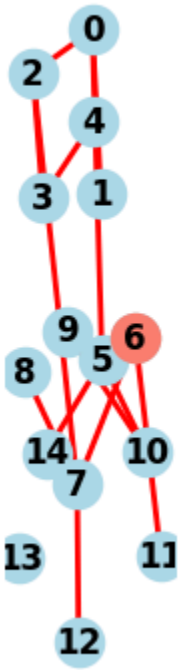
Step: 6



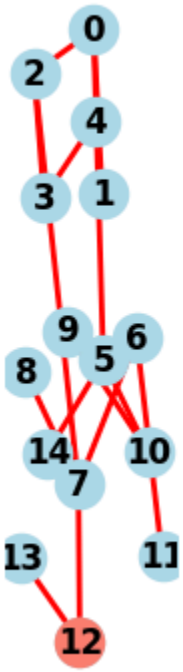
Step: 10



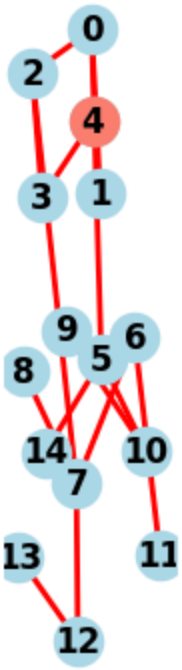
Step: 6



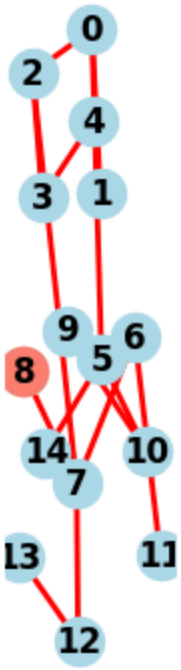
Step: 12



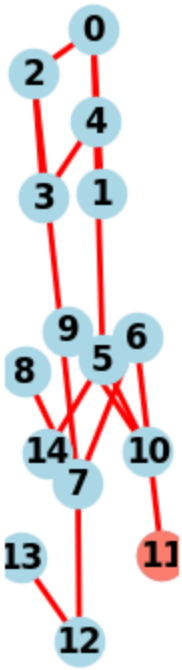
Step: 4



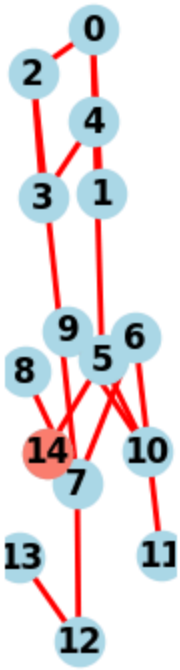
Step: 8



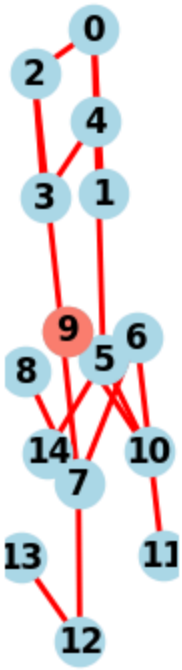
Step: 11



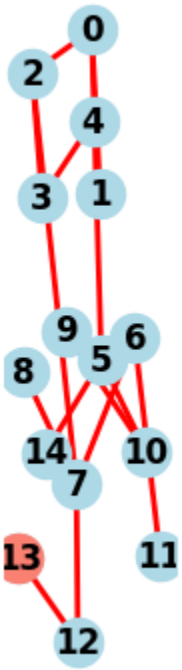
Step: 14



Step: 9



Step: 13



EXP-9(Implementation and analysis of Leaky bucket and Token bucket congestion control algorithms)

LeakyBucket:

```
import java.util.Scanner;
import java.io.*;
public class Leaky{
    public static void main(String[] args) {
        int bufferSize = 0;
        final int bucketSize = 10;
        final int inputPacketSize = 4;
        final int outputPacketSize = 1;
        final int numQueries = 4;

        System.out.println("Initial buffer size: " + bufferSize);
        System.out.println("Bucket size: " + bucketSize);
        System.out.println("Input packet size: " + inputPacketSize);
        System.out.println("Output packet size: " +
outputPacketSize);
        System.out.println("Number of queries: " + numQueries);

        for (int i = 0; i < numQueries; i++) {
            int spaceLeft = bucketSize - bufferSize;

            if (inputPacketSize <= spaceLeft) {
                bufferSize += inputPacketSize;
                System.out.println("\n\nPacket added to buffer. New
buffer size: " + bufferSize);
            } else {
                // when  there is no space in buffer
                int packetsLost = inputPacketSize - spaceLeft;
                System.out.println("Packet loss = " + packetsLost);
                bufferSize = bucketSize; // Reset buffer to bucket
size after overflow
                System.out.println("Buffer reset. New buffer size: "
+ bufferSize);
            }
```

```
            bufferSize -= outputPacketSize;
            System.out.println("Packet transmitted. New buffer size:
" + bufferSize);
            System.out.println("----------------------");
        }
    }
}
```

Output:

```
E:\java>javac Leaky.java

E:\java>java Leaky
Initial buffer size: 0
Bucket size: 10
Input packet size: 4
Output packet size: 1
Number of queries: 4


Packet added to buffer. New buffer size: 4
Packet transmitted. New buffer size: 3
--------------------


Packet added to buffer. New buffer size: 7
Packet transmitted. New buffer size: 6
--------------------


Packet added to buffer. New buffer size: 10
Packet transmitted. New buffer size: 9
--------------------
Packet loss = 3
Buffer reset. New buffer size: 10
Packet transmitted. New buffer size: 9
--------------------

E:\java>
```

Token Bucket:

```
import java.io.*;
import java.lang.*;
import java.util.*;

class Bucket{
 public int tokens, maxsize;

 Bucket(int max){
      tokens = 0;
      maxsize = max;
 }

 synchronized void addToken(int n){
      if(tokens >= maxsize) return;
      tokens += 1;
      System.out.println("Added a token. Total:" + tokens);
 }

 synchronized void sendPacket(int n){
      System.out.println("Packet of size " + n + " arrived");
      if(n > tokens){
           System.out.println("Packet is non comformant,
discarded");
      }
      else{
           tokens -= n;
           System.out.println("Forwarding packet");
      }
 }
}

class AddTokenThread extends Thread{
 Bucket b;
 AddTokenThread(Bucket b){
      this.b = b;
 }
 public void run(){
      while(true){
           b.addToken(1);
```

```
            try{
                  Thread.sleep(300);
            } catch(Exception e){}
      }
  }
}

class AddPacketThread extends Thread{
 Bucket b;
 AddPacketThread(Bucket b){
      this.b = b;
 }

 public void run(){
      while(true){
            try{
                  Thread.sleep(500 + (int) (Math.random()*3000));
            }
            catch(Exception e){}
            b.sendPacket(1 + (int) (Math.random()*9));
      }
 }
}

class TokenBucket{
 public static void main(String args[]){
      Bucket b = new Bucket(10);
      Thread tokens = new AddTokenThread(b);
      Thread packets = new AddPacketThread(b);
      try{
            tokens.start();
            packets.start();
      }
      catch(Exception e){}
 }
}
```

Output:

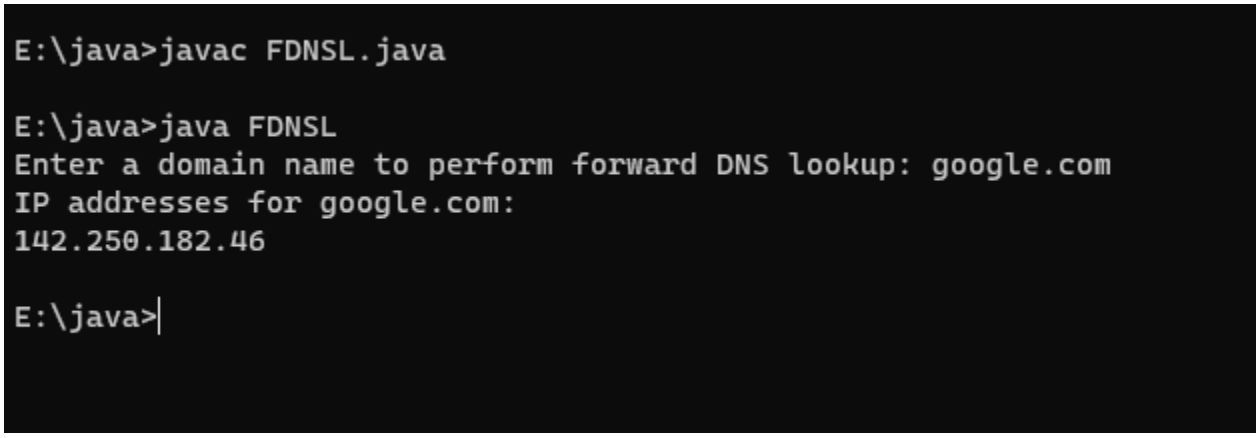
```
E:\java>javac TokenBucket.java

E:\java>java TokenBucket
Added a token. Total:1
Added a token. Total:2
Added a token. Total:3
Added a token. Total:4
Added a token. Total:5
Added a token. Total:6
Packet of size 6 arrived
Forwarding packet
Added a token. Total:1
Added a token. Total:2
Added a token. Total:3
Added a token. Total:4
Packet of size 9 arrived
Packet is non comformant, discarded
Added a token. Total:5
Added a token. Total:6
Added a token. Total:7
Added a token. Total:8
Added a token. Total:9
Added a token. Total:10
Packet of size 6 arrived
Forwarding packet
Added a token. Total:5
Added a token. Total:6
Added a token. Total:7
Added a token. Total:8
Added a token. Total:9
Added a token. Total:10
Packet of size 9 arrived
Forwarding packet
Added a token. Total:2
Added a token. Total:3
Added a token. Total:4
Added a token. Total:5
Added a token. Total:6
Added a token. Total:7
Added a token. Total:8
Added a token. Total:9
```

exp-10(Implementation of DNS lookup)

Forward DNS:

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

public class FDNSL {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a domain name to perform forward DNS
lookup: ");
        String domainName = scanner.nextLine();
        forwardDNSLookup(domainName);
        scanner.close();
    }

    public static void forwardDNSLookup(String domainName) {
        try {
            InetAddress[] addresses =
InetAddress.getAllByName(domainName);
            System.out.println("IP addresses for " + domainName +
":");
            for (InetAddress address : addresses) {
                System.out.println(address.getHostAddress());
            }
        } catch (UnknownHostException e) {
            System.out.println("Error: Unable to resolve the domain
name " + domainName);
        }
    }
}
```

Output:



Reverse DNS:

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

public class RDNS {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter an IP address to perform reverse DNS
lookup: ");
        String ipAddress = scanner.nextLine();
        reverseDNSLookup(ipAddress);
        scanner.close();
    }

    public static void reverseDNSLookup(String ipAddress) {
        try {
            InetAddress inetAddress =
InetAddress.getByName(ipAddress);
            String hostName = inetAddress.getCanonicalHostName();
            System.out.println("Domain name for " + ipAddress + ": "
+ hostName);
        } catch (UnknownHostException e) {
```

```
            System.out.println("Error: Unable to perform reverse DNS
lookup for IP address " + ipAddress);
        }
    }
}
```

Output: