1. **Please mention group members names and UFID.**
   Name: Smrati Pandey
   UFID: ****-****

   Sachit Verma
   UFID: ****-****

   Steps to run the program -
   1. Download the file and unzip it
   2. Run the command: **mix run project3.exs 1000 5 3**

2. **What is working?**

   - In our assumption of tapestry algorithm, we are taking the input of numNodes and numRequests. Out of which we are creating the routing table for numNodes - 1 nodes. The last node is being inserted dynamically in the system.

   - We are using SHA1 function to generate our hash. Each node is represented by a unique hash containing 40-digit hexadecimal number.
     Example: '5B384CE32D8CDEF02BC3A139D4CAC0A22BB029E8'

     ```
     :crypto.hash(:sha, "#{nodeID}") |> Base.encode16()
     ```

   - For creating the routing table, we are comparing the numNode with its neighbour nodes by using the common substring formulae. In this formula, we compare the nodes using **prefix matching** and update the table where the numNodes do not match. Suppose, we took two nodes as “ABCD” and “ABCE”. As the node matches for first three characters, we increment to the third level and update the node “ABCE” in the E slot.

     “ABCD” -> {3, “E”} => “ABCE” (Level are starting from zero.)

     Suppose the table already has an entry at the {3, “E”} node then we perform comparison using the distance formulae. The closeness is calculated by taking the absolute value of the difference between hashes. The node that is nearest to the source numNode is stored in the table. The other node is discarded.

   - For the new node insertion in the network, we are first creating the routing table for the newNode by using the similar function used for creating the routing table for rest of the numNodes. Then, we iterate over its neighbour’s list to update the newNode in their table. The same scheme as above is followed to resolve conflict here. For example, if we already have an entry in the table, we check the nearest node and insert in the table. The table is updated with the new entry and we have prepared our network for hoping.

   - Our workers are being initialized using the supervisor. We are mapping all the hashKeys with the PID’s for hopping in the network. Once every hashKey has been associated with a PID, we start our hopping from the source ID to the destination ID. We have achieved parallelism by implementing hopping part of our algorithm in **GenServer.cast()** method which works asynchronously. We create a destination list for every source node by using the numRequests

in the input. Say the numRequests is 5, then for every numNode (Suppose 100 nodes) in the system we create 5 destination requests. Each destination request is chosen randomly from the network. The destination node can be any random node in the network except for the source node itself.

- First, we calculate the maximum of the hops for a node amongst the 5 numRequests calls.
  For example,
  Source Node
  -> Destination Node1 = Hops (3)
  -> Destination Node2 = Hops (3)
  -> Destination Node3 = Hops (1)
  -> Destination Node4 = Hops (2)
  -> Destination Node5 = Hops (4)

  We choose the maximum hops. So here we got the max hop as 4.
  Similarly, for all the 100 source numNodes we get the max hops.

  SourceNode1 - > maxHops(3)
  SourceNode2 - > maxHops(4)
  SourceNode3 - > maxHops(2)
  SourceNode4 - > maxHops(1)
  SourceNode5 - > maxHops(5)
  And so, on
  SourceNode100 - > maxHops(4)

  Over all the nodes we give the totalMaxhops = 5

  We are storing the maximum number of hops in the state of each worker and prints the maximum of all those using **GenServer.call()** method in main method.

- We have **preferred using ETS over Elixir Maps** for creating and storing the routing tables. The reason for this was that fetching data from Elixir Maps takes logarithmic time as compared to constant time in ETS. So, as the network grows fetching data becomes slow if we use maps. However, time taken in fetching data from ETS is independent of the network size.

- **Each node will make only one request per second.** By putting this condition for each node, our program will go to sleep after every request the source node makes except for the last one. For example, if numRequests is 5 for one node then it will go to sleep for a total of (numRequests – 1) seconds i.e. 4 seconds in this case.

- **Hopping implementation** – While calculating the number of hops from a source node to its destination node, we first find a neighbouring node in its routing table using **prefix matching.** With prefix matching we check if the node is present in the routing table of the source node. if yes, we hop to the destination node's process id and our counter terminates as it found the destination node. Otherwise, through prefix matching we get to know the level and the slot where the nearest neighbour is stored. We hop to the nearest neighbour node's process ID and check in the routing table of the neighbour node if destination node is present and increment our hopping counter. If the destination is found, we terminate our recursion otherwise we keep hopping in the same way until we reach the destination. The counter keeps

on incrementing giving us the number of hops required to reach from source node to the destination node.

- At the end, when all nodes have hopped, we print the maximum number of hops.

3. The failure model implementation

For the failure model implementation, we are failing max of three nodes in the system. In this case, if a node encounters a killed node while hopping, it will route to new node that is built to handle the fail-case scenario. So, it will direct it to the destination node from the newly created node. This node has its link to all the nodes in the system. If anyone of the nodes encounters a killed node. It can route itself to new node to reach the destination node.

4. What is the largest network you managed to deal with?

Largest Network dealt with:
numNodes: 20000
numRequests: 1
Maximum Hops: 6

numNodes	numRequests	maxHops	Num_of_fail_nodes
1000	1	4	3
1000	5	4	3
2000	1	4	3
2000	5	5	3
3000	1	5	3
3000	5	5	3
4000	1	5	3
4000	5	5	3
5000	1	5	3
5000	5	5	3
6000	1	5	3
6000	5	5	3
10000	1	6	3
20000	1	6	3