1. Sorting Array

The sortByColours function sorts an array:

- Black, red, and yellow values 0,1 and 2.
- The array order is specified (black first, then red, then yellow), so all 0s come first, all 1s, and all 2s.
- No built-in sort functions or libraries.
- Modify the original array directly.

sortByColours function:

```
def sortByColours(numColour):
        low = 0
    mid = 0
    high = len(numColour) - 1


    while mid <= high:
      if numColour[mid] == 0:
        numColour[low], numColour[mid] = numColour[mid], numColour[low]
        low += 1
        mid += 1
      elif numColour[mid] == 1:
        mid += 1
```

```
else:
        numColour[high], numColour[mid] = numColour[mid], numColour[high]
        high -= 1
```

1. Minimum Height Tree

```
FUNCTION findMHTs(n, edges):
  IF n == 1 THEN
    RETURN [0]  // Special case: single node is the MHT root

  // Build the graph as an adjacency list
  INITIALIZE graph as an empty dictionary of lists
  FOR each edge (u, v) in edges DO
    ADD v to graph[u]
    ADD u to graph[v]

  FUNCTION bfs(start):
    INITIALIZE distance as a list of -1 with size n
    INITIALIZE queue as a deque with start
    SET distance[start] = 0
    INITIALIZE farthest_node = start
    WHILE queue is not empty DO
        SET node = queue.popleft()
        FOR each neighbor of node DO
            IF distance[neighbor] == -1 THEN
                SET distance[neighbor] = distance[node] + 1
                ADD neighbor to queue
                IF distance[neighbor] > distance[farthest_node] THEN
                    SET farthest_node = neighbor
    RETURN farthest_node, distance

// Find the farthest node from an arbitrary node (e.g., node 0)
  SET farthest_node_1, _ = bfs(0)
  // Find the farthest node from farthest_node_1
  SET farthest_node_2, dist_from_1 = bfs(farthest_node_1)
  // Find distances from farthest_node_2
  SET _, dist_from_2 = bfs(farthest_node_2)
  // Compute the diameter of the tree
  SET diameter = dist_from_1[farthest_node_2]
```

```
    // Find all nodes that are at the center of the tree
INITIALIZE centers as an empty list
    FOR i from 0 to n-1 DO
        IF (dist_from_1[i] == diameter // 2 OR dist_from_1[i] == (diameter + 1) // 2) AND
           (dist_from_2[i] == diameter // 2 OR dist_from_2[i] == (diameter + 1) // 2) THEN
            ADD i to centers
    RETURN centers


// Example usage
FUNCTION main():
    READ n from input
    INITIALIZE edges as an empty list
    PRINT "Enter n - 1 edges in the format 'a b' (one edge per line):"
    FOR i from 1 to n - 1 DO
        READ a, b from input
        ADD [a, b] to edges
    // Find Minimum Height Trees
    SET mhts = findMHTs(n, edges)
    PRINT "Minimum Height Trees roots:", mhts
```

```
// Call the main function to execute

CALL main()
```

Minimum Height Tree (MHT) functions:

Number of nodes (n): 4

Edges (n-1) :[1,0],[1,2],[1,3]

It requires connecting nodes 0 to 3 by constructing a tree with four edges.

Node 1 is linked to nodes 0, 2, and 3. The diameter can be determined by executing Backwards Finite Fields (BFS) starting with node 1.

Node 1, located in the middle of the diameter path, is the center of the tree.

As a result, node 1 is the only root with a height of one, the minimum possible height.

3. Clique Problem

3.1

A clique is defined as a complete subgraph of an undirected graph in graph theory, where each pair of different vertices in the subset is nearby and connected by an edge, ensuring that every pair of vertices in the clique is connected directly.

A graph with a 3-clique (also called a triangle) means a set of three vertices where each vertex is connected to the other two by an edge because a clique is called a k-clique if it has k vertices.

Example of a 3-clique:

A graph with 4 vertices: A, B, C, and D

Edges:

- (A,B)
- (A,C)
- (B,C)
- (C,D)

In this graph, the vertices A, B, and C form a **3-clique**, because:

- There is an edge between A and B
- An edge between A and C,
- An edge between B and C.

This forms a complete subgraph (a triangle).

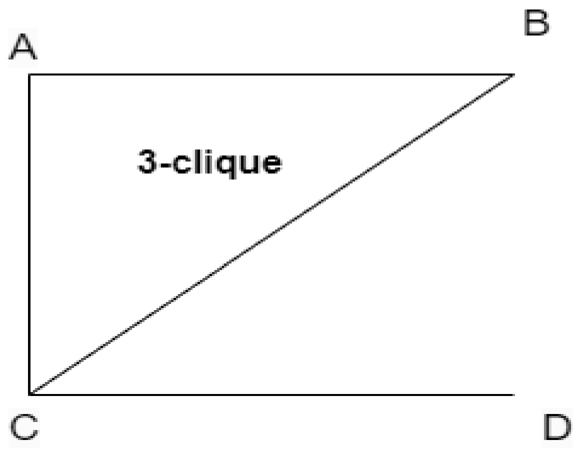

Figure : graph with a 3-clique.

3.2 Pseudo code for graph contains a k-clique

```
 Algorithm ContainsKClique(Graph G, Integer k):

   Input: Graph G with vertices V and edges E, Integer k



Output: True if G contains a k-clique, False otherwise

   Function IsClique(VertexSubset subset):

       for each pair of vertices (u, v) in subset:

           if (u, v) not in E:

               return False

       return True

   // Main procedure

   V = GetVertices(G) // Get all vertices from graph G

   for each subset in Combinations(V, k): // Directly generate k-sized subsets

       if IsClique(subset):

           return True

   return False
```

4. Finite State Machine

In 3x3 square board Tic Tac Toe game:

Initial state S0 and End state S3 and S4.

From S0 to S1:

Condition: Player O makes the first move.

O		

From S1 to S1:

Condition: Players alternate turns, making moves.

X	O	

From S1 to S2:

Condition: Player X achieves three in a row.

X	O	X
	X	

From S1 to S3:

Condition: Player O achieves three in a row.

X	O	X

O		O
X	O	X

From S1 to S4:

Condition: The board is full, and no one has won.

X	O	X
O	X	O
X	O	X

The transition diagram of a Tic-Tac-Toe game

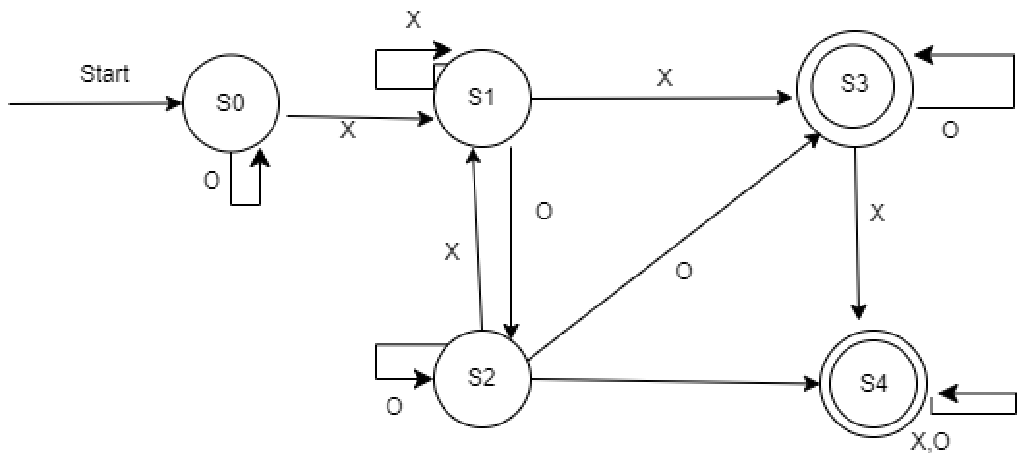

Figure: the transition diagram of a Tic-Tac-Toe game

The transitions table of a Tic-Tac-Toe game

State	**Input X**	**Input O**	**Row of three**	**Full**

S0	**S1**	**S0**		
S1	**S1**	**S2**	**S3**	**S4**
S2	**S1**	**S2**	**S3**	**S4**
S3	**S3**	**S3**	**S3**	**S4**
S4	**S4**	**S4**		**S4**