Algoritham Desiging:
Designed Algoritham using Greedy Approach and Back Tracking.
And Travesing Techings BFS(Breadth First Search) and DFS (Depth First Search).

Naming Conventions:
Given a sutable name for the star Graphs based on the Structure.

Time Complexity Analysis:
A thorough analysis of the algorithm's time complexity has been conducted. The star graph construction algorithm has a time complexity of O(n), where n is the number of peripheral nodes. This efficient performance is achieved through the strategic use of data structures and optimized operations.

Custom Data Structure:
To support efficient graph operations and analysis, I have designed a custom data structure tailored specifically for star graphs. This data structure, called StarGraph, encapsulates the central node and peripheral nodes, as well as their connections. It provides methods for graph construction, traversal, and property computation. The StarGraph class leverages java object-oriented programming capabilities and follows the principles of encapsulation and abstraction, ensuring code reusability and extensibility.
This algorithm and its associated data structure contribute to the ongoing research efforts in the field of star graphs, enabling more efficient analysis, exploration of new properties, and potential applications in various domains, such as computer networks, social network analysis, and graph theory.

Problem-1

Homogeneous amalgamated Star: $S_{n,3}$:

Output for n = 8, m = 3

```
n: 8     m: 3
k: 13
Root: 10
  └── Star: 1 E<10,1> l = 11
      └── Leaf: 1 E<1,1> l = 2
      └── Leaf: 2 E<1,2> l = 3
  └── Star: 2 E<10,2> l = 12
      └── Leaf: 2 E<2,2> l = 4
      └── Leaf: 3 E<2,3> l = 5
  └── Star: 3 E<10,3> l = 13
      └── Leaf: 3 E<3,3> l = 6
      └── Leaf: 4 E<3,4> l = 7
  └── Star: 4 E<10,4> l = 14
      └── Leaf: 4 E<4,4> l = 8
      └── Leaf: 5 E<4,5> l = 9
  └── Star: 5 E<10,5> l = 15
      └── Leaf: 5 E<5,5> l = 10
      └── Leaf: 11 E<5,11> l = 16
  └── Star: 7 E<10,7> l = 17
      └── Leaf: 11 E<7,11> l = 18
      └── Leaf: 12 E<7,12> l = 19
  └── Star: 10 E<10,10> l = 20
      └── Leaf: 11 E<10,11> l = 21
      └── Leaf: 12 E<10,12> l = 22
  └── Star: 13 E<10,13> l = 23
      └── Leaf: 11 E<13,11> l = 24
      └── Leaf: 12 E<13,12> l = 25
```

Output for n = 9, m = 3

```
n: 9     m: 3
k: 14
Root: 12
  └── Star: 1 E<12,1> l = 13
      └── Leaf: 1 E<1,1> l = 2
      └── Leaf: 2 E<1,2> l = 3
  └── Star: 2 E<12,2> l = 14
      └── Leaf: 2 E<2,2> l = 4
      └── Leaf: 3 E<2,3> l = 5
  └── Star: 3 E<12,3> l = 15
      └── Leaf: 3 E<3,3> l = 6
      └── Leaf: 4 E<3,4> l = 7
  └── Star: 4 E<12,4> l = 16
      └── Leaf: 4 E<4,4> l = 8
      └── Leaf: 5 E<4,5> l = 9
  └── Star: 5 E<12,5> l = 17
      └── Leaf: 5 E<5,5> l = 10
      └── Leaf: 6 E<5,6> l = 11
  └── Star: 6 E<12,6> l = 18
      └── Leaf: 6 E<6,6> l = 12
      └── Leaf: 13 E<6,13> l = 19
  └── Star: 8 E<12,8> l = 20
      └── Leaf: 13 E<8,13> l = 21
      └── Leaf: 14 E<8,14> l = 22
  └── Star: 11 E<12,11> l = 23
      └── Leaf: 13 E<11,13> l = 24
      └── Leaf: 14 E<11,14> l = 25
  └── Star: 14 E<12,14> l = 26
      └── Leaf: 13 E<14,13> l = 27
      └── Leaf: 14 E<14,14> l = 28
```

Explanation:

1. For a given n, k can be calculated as **k = ceil (n*3 + 1 / 2)**. Therefore, for n = 8, k = 13 and for n = 9, k = 14, which can be seen in the above figures.
2. Adjacency List data structure is used for storing the graph.
3. In the above figures, star node stores the edge values from root to star and, leaf node stores the edge value from its star to leaf.

   E.g.: Star: 1 E<10,1> l = 11

   1 => label of star

   E<10, 1> => Edge with vertex values 10(root) and 1(star)

   l = 11 => Edge weight

   E.g.: Leaf: 1 E<1,1> l = 2

   1 => label of leaf

   E<1, 1> => Edge with vertex values 1(star) and 1(leaf) l = 2 =>

Edge weight **Algorithms:**

Algorithm 1 HomogeneousAmalgamatedStar() method

Require: n

$k = ((n * 3) + 1) / 2$
EdgeLabelSet = [ ]
for $r = 1$ **to** k **do**
  Set the label of the root vertex as r
  for all stars in root's children **do**
    for $i = 1$ **to** k **do**
      edgeValue = root.vertexValue + i
      if edgeValue is not in EdgeLabelSet **then**
        star.vertexValue = i
        star.edgeValue = edgeValue
        Add edgeValue to EdgeLabelSet
        if labelStarChildren(star) is successful **then**
          Break the loop and move to the next star
        else
          Backtrack - remove edgeValue from EdgeLabelSet. Reset star's Vertex and Edge labels
        end if
      end if
    end for
    if star's value is still 0 i.e., no value is assigned **then**
      Break the loop (start from root with next possible value for root.)
    end if
  end for
  if all edges are labeled **then**
    Successfully completed labelling.
    Exit the loop
  else
    Backtrack - clear EdgeLabelSet. Clear Vertex and Edge labels for all nodes.
  end if
end for

Algorithm 2 labelStarChildren() method

Require: *star*

StarEdgeLabelSet = [ ]
for all node in star's children **do**
  for $i = 1$ **to** k **do**
    edgeValue = star.vertexValue + i
    if edgeValue is **not** in EdgeLabelSet **then**
      Add edgeValue to EdgeLabelSet
      Add edgeValue to StarEdgeLabelSet
      node.vertexValue = i
      node.edgeValue = edgeValue
      Break the loop
    end if
  end for
  if node's value is still 0 i.e., no value is assigned **then**
    Backtrack - Remove StarEdgeLabelSet from EdgeLabelSet.
    return false
  end if
end for
All nodes are successfully labelled
return true

Problem-2

Homogeneous amalgamated Star: $S_{n, m}$:

Output for n = 6, m = 4

```
n: 6     m: 4
k: 13
Root: 9
  └── Star: 1 E<9,1> l = 10
      └── Leaf: 1 E<1,1> l = 2
      └── Leaf: 2 E<1,2> l = 3
      └── Leaf: 3 E<1,3> l = 4
  └── Star: 2 E<9,2> l = 11
      └── Leaf: 3 E<2,3> l = 5
      └── Leaf: 4 E<2,4> l = 6
      └── Leaf: 5 E<2,5> l = 7
  └── Star: 3 E<9,3> l = 12
      └── Leaf: 5 E<3,5> l = 8
      └── Leaf: 6 E<3,6> l = 9
      └── Leaf: 10 E<3,10> l = 13
  └── Star: 5 E<9,5> l = 14
      └── Leaf: 10 E<5,10> l = 15
      └── Leaf: 11 E<5,11> l = 16
      └── Leaf: 12 E<5,12> l = 17
  └── Star: 9 E<9,9> l = 18
      └── Leaf: 10 E<9,10> l = 19
      └── Leaf: 11 E<9,11> l = 20
      └── Leaf: 12 E<9,12> l = 21
  └── Star: 13 E<9,13> l = 22
      └── Leaf: 10 E<13,10> l = 23
      └── Leaf: 11 E<13,11> l = 24
      └── Leaf: 12 E<13,12> l = 25
```

Output for n = 3, m = 7

```
n: 3     m: 7
k: 11
Root: 5
  └── Star: 1 E<5,1> l = 6
      └── Leaf: 1 E<1,1> l = 2
      └── Leaf: 2 E<1,2> l = 3
      └── Leaf: 3 E<1,3> l = 4
      └── Leaf: 4 E<1,4> l = 5
      └── Leaf: 6 E<1,6> l = 7
      └── Leaf: 7 E<1,7> l = 8
  └── Star: 4 E<5,4> l = 9
      └── Leaf: 6 E<4,6> l = 10
      └── Leaf: 7 E<4,7> l = 11
      └── Leaf: 8 E<4,8> l = 12
      └── Leaf: 9 E<4,9> l = 13
      └── Leaf: 10 E<4,10> l = 14
      └── Leaf: 11 E<4,11> l = 15
  └── Star: 11 E<5,11> l = 16
      └── Leaf: 6 E<11,6> l = 17
      └── Leaf: 7 E<11,7> l = 18
      └── Leaf: 8 E<11,8> l = 19
      └── Leaf: 9 E<11,9> l = 20
      └── Leaf: 10 E<11,10> l = 21
      └── Leaf: 11 E<11,11> l = 22
```

Explanation:

1. For a given n and m, k can be calculated as **k = ceil (n*m + 1 / 2).** Therefore, for n = 6 and m = 4, k = 13 and for n = 3 and m = 7, k = 11, which can be seen in the above figures.

2. Adjacency List data structure is used for storing the graph.
3. We can observe that for dynamic n and m values, labelling is possible with the given k value.
4. In the above figures, star node stores the edge values from root to star and, leaf node stores the edge value from its star to leaf.

   E.g.: Star: 1 E<9,1> l = 10

   9 => label of star

   E<9, 1> => Edge with vertex values 9(root) and 1(star)

   l = 10 => Edge weight

   E.g.: Leaf: 1 E<1,1> l = 2

   1 => label of leaf

   E<1, 1> => Edge with vertex values 1(star) and 1(leaf) l = 2 =>

Edge weight **Algorithm:**

Algorithm 1 HomogeneousAmalgamatedStar() method

Require: n, m

```
1:  k = ((n * m) + 1) / 2
2:  EdgeLabelSet = [ ]
3:  for r = 1 to k do
4:    Set the label of the root vertex as r
5:    for all stars in root's children do
6:      for i = 1 to k do
7:        edgeValue = root.vertexValue + i
8:        if edgeValue is not in EdgeLabelSet then
9:          star.vertexValue = i
10:         star.edgeValue = edgeValue
11:         Add edgeValue to EdgeLabelSet
12:         if labelStarChildren(star) is successful then
13:           Break the loop and move to the next star
14:         else
15:           Backtrack - remove edgeValue from EdgeLabelSet. Reset star's
              Vertex and Edge labels
16:         end if
17:       end if
18:     end for
19:     if star's value is still 0 i.e., no value is assigned then
20:       Break the loop (start from root with next possible value for root.)
21:     end if
22:   end for
23:   if all edges are labeled then
24:     Successfully completed labelling.
25:     Exit the loop
26:   else
27:     Backtrack - clear EdgeLabelSet. Clear Vertex and Edge labels for all
        nodes.
28:   end if
29: end for
```

labelStarChildren () algorithm is used again from problem 1.

Problem-3

Cyclic Homogeneous amalgamated Star: $S_{n, m}$:

Output for n = 6

```
n: 6     m: 3
k: 14
Root: 9
  └── Star: 13 E<9,13> l = 22
      └── CE<13,14> l = 27
      └── 1 E<13,1> l = 14
      └── 2 E<13,2> l = 15
  └── Star: 4 E<9,4> l = 13
      └── CE<4,13> l = 17
      └── 1 E<4,1> l = 5
      └── 2 E<4,2> l = 6
  └── Star: 3 E<9,3> l = 12
      └── CE<3,4> l = 7
      └── 1 E<3,1> l = 4
      └── 5 E<3,5> l = 8
  └── Star: 7 E<9,7> l = 16
      └── CE<7,3> l = 10
      └── 2 E<7,2> l = 9
      └── 4 E<7,4> l = 11
  └── Star: 11 E<9,11> l = 20
      └── CE<11,7> l = 18
      └── 8 E<11,8> l = 19
      └── 10 E<11,10> l = 21
  └── Star: 14 E<9,14> l = 23
      └── CE<14,11> l = 25
      └── 10 E<14,10> l = 24
      └── 12 E<14,12> l = 26
```

Output for n = 7

```
n: 7     m: 3
k: 16
Root: 11
  └── Star: 15 E<11,15> l = 26
      └── CE<15,16> l = 31
      └── 1 E<15,1> l = 16
      └── 2 E<15,2> l = 17
  └── Star: 4 E<11,4> l = 15
      └── CE<4,15> l = 19
      └── 1 E<4,1> l = 5
      └── 2 E<4,2> l = 6
  └── Star: 7 E<11,7> l = 18
      └── CE<7,4> l = 11
      └── 1 E<7,1> l = 8
      └── 2 E<7,2> l = 9
  └── Star: 3 E<11,3> l = 14
      └── CE<3,7> l = 10
      └── 1 E<3,1> l = 4
      └── 4 E<3,4> l = 7
  └── Star: 9 E<11,9> l = 20
      └── CE<9,3> l = 12
      └── 4 E<9,4> l = 13
      └── 12 E<9,12> l = 21
  └── Star: 13 E<11,13> l = 24
      └── CE<13,9> l = 22
      └── 10 E<13,10> l = 23
      └── 12 E<13,12> l = 25
  └── Star: 16 E<11,16> l = 27
      └── CE<16,13> l = 29
      └── 12 E<16,12> l = 28
```

Explanation:

1. For a given n and m, k can be calculated as **k = ceil ((n*m + n + 2) / 2) + lg n**. Therefore, for n = 6 and m = 3, k = 14 and for n = 7 and m = 3, k = 16, which can be seen in the above figures.
2. Adjacency List data structure is used for storing the graph.
3. In the above figures, star node stores the edge values from root to star and, leaf node stores the edge value from its star to leaf.
4. Each star also maintains pointers to its previous and next star nodes. Cycle Edge from previous start to current star is maintained at current star.

   E.g.: Star: 13 E<9,13> l = 22

   13 => label of star

   E<9, 13> => Edge with vertex values 9(root) and 13(star)

   CE<13, 14> => Cycle Edge with vertex value 14(curr star) and 14(prev star) l = 22 => Edge weight

   E.g.: Leaf: 1 E<13,1> l = 14

   1 => label of leaf

   E<13, 1> => Edge with vertex values 13(star) and 1(leaf) l = 14 => Edge weight

Algorithms:

Algorithm 1 CyclicHomogeneousAmalgamatedStar() method

Require: n

```
k = ((n * 3) + n + 2) / 2 + log n
for r = 1 to k do
    Set the label of the root vertex as r
    if labelStars(root.children[0]) == True then
        Successfully labelled graph.
        Exit the loop
    else
        Backtrack - Clear edge labels.
    end if
end for
```

labelStarChildren ()algorithm can be reused from probem1.

Algorithm 2 labelStars() recursion method

Require: *star*

```
if star.isVisited == True then
    lastEdgeValue = star.vertexValue + star.prev.vertexValue
    if lastEdgeValue is not in EdgeLabelSet then
        edgeLabels.add(lastEdgeValue)
        return true
    else
        return false
    end if
end if

star.isVisited = True
for i = 1 to k do
    edgeValue = root.vertexValue + i
    if edgeValue is not in EdgeLabelSet then
        Add edgeValue to EdgeLabelSet
        star.vertexValue = i
        if labelStarChildren() == True and labelStars(star.next) == True
        then
            star.edgeValue = edgeValue
            star.cycleEdgeValue = star.value + star.prev.value
            return true
        else
            Backtrack - EdgeLabelSet.remove(edgeValue). Reset star's and it's
            children's Vertex and Edge labels.
        end if
    end if
end for
star.isVisited = false
return false
```

Tasks – Problem 1 & 2

1. **Find out the best data-structure to represent / store the graph in memory.**

   The best data structure is Adjacency list.

   1. Root node maintains the list of star nodes.
   2. Each star node maintains the list of its children (leaf) nodes. Each star also stores the edge from root to star. For cyclic graph, each star nodes also maintains pointers for its previous and next star nodes. Each star also stores the cycle edge connecting to its previous star.
   3. Each leaf node maintains the edge from its star to the leaf node.

```java
class Root extends Node{
    ArrayList<Star> children; // Level1 nodes.

    Root(int n, int m) {
        children = new ArrayList<>();
        for(int i = 0; i < n; i++) {
            children.add(new Star(m));
        }
    }
}
```

```java
class Star extends Node{
    ArrayList<Node> children; // Level2 nodes.

    Star(int m) {
        children = new ArrayList<>();
        for(int i = 0; i < m-1; i++) {
            children.add(new Node());
        }
    }
}
```

```
class Leaf extends Node {

}
```

```
class Node {
    int value = 0;
    // Star stores the edge value of Root to Star.
    // Leaf stores the edge value of Star to Leaf.
    int edgeValue = 0;
}
```

2. **Devise an algorithm to assign the labels to the vertices using vertex k-labeling definition. (Main Task)**

   Algorithms were designed for each of the problems 1, 2 and 3 and mentioned in the respective sections above.

3. **What design strategy you will apply, also give justifications that selected strategy is most appropriate.**

   Backtracking is the design strategy that we chose for all 3 problems.

   Justification:

   1. For the given graph labelling problem, multiple solutions are possible. The most common approach will be brute force approach. But the complexity will be very high. For brute force the complexity will be $\mathbf{k}^{n*m+1}$.
      Where, k is the maximum possible value for the vertex which is (n*m + 1 / 2) and n*m + 1 = number of vertices.
      For higher values of n and m, the complexity for brute force approach increases and it is not feasible.
   2. In backtracking, we label the root node first and then label the star node and its children (lead nodes). If the current star node and its children are labelled, then we proceed to the next star node and so on. At any node, if we cannot label it, then we do not proceed further. We move to the previously labelled nodes and check if any other value is possible for the previous node and assign the value to it and then come to the current node and so on. In this way, most of the combinations that we see in brute force approach can be avoided.
   3. Divide and Conquer approach cannot be applied here, as the labels of the one subgraph depend on the labels of the other subgraphs.
   4. Applying dynamic programming might not be a feasible approach as the current substructure values cannot be same as the previously encountered substructure. For the above-mentioned reasons, backtracking is the most appropriate approach for labelling the given graphs.

4. **How traversing will be applied?**

We have applied DFS (Depth First Search) traversing.

1. Label the root node.
2. Label the first (star node) of the root.
3. Label the child (leaf node) of the star node.
4. Control goes back to the star node.
5. **Steps 3 to 4** will repeat until all the children (leaf nodes) of current star are labelled.
6. Control goes back to the root node.
7. **Steps 2 to 6** will repeat until all the children (star nodes) of root are labelled.
8. At any point, if a node cannot be labelled, the values set on any sibling nodes and its children will be reset and the control goes back to its parent node for relabeling.

5. **Store the labels of vertices and weights of the edges as an outcome.**

Each node stores the vertex label in vertexLabel variable.

1. Star node stores the edge weight from root to star in edgeLabel variable.
2. Leaf node stores the edge weight from start to leaf in edgeLabel variable.
3. Additionally, for cyclic graph, star node stores the edge weight from previous star node to the current star node in cycleEdgeLabel variable.

6. **Compare your results with mathematical property and tabulate the outcomes for comparison.**

m = 3

n (stars)	**Total no. vertices**	**Max vertex val using formula**	**Max vertex value using the designed algorithm**
3	10	5	5
4	13	7	7
5	16	8	8
6	19	10	10
7	22	11	11
8	25	13	13
9	28	14	14

10	31	16	16
50	151	76	76
100	301	151	151
500	15001	751	751
1000	3001	1501	1501
5000	15001	7501	7501
10000	30001	15001	15001

From the above result, we can say that the results of the designed algorithm match with the mathematical property.

7. Hardware resources supported until what maximum value of n, m.

Tried on MacBook Air with M1 chip.

1. **Higher n:** Successfully able to run for n = 10000 and m = 3. We tried for even higher n values, but it is taking very long time (more than an hour).
2. **Average n and m:** Successfully able to run for n = 300 and m = 300. Run time is indefinite for higher n and m values.
3. **Higher m:** Successfully able to run for n = 3 and m = 3000. We tried for even higher n values, but it is taking very long time (more than an hour).

8. Compute the Time Complexity of your algorithm T (V, E) or T(n).

As we are passing the values of n and m, the time complexity is calculated using n and m.

1. Acyclic graph creation – O (n*m + 1)
2. The outer loop for root runs k times – O ((n*m + 1)/2)
    1. The first nested loop runs for all stars – O(n)
    2. The second nested loop iterates from 1 to k - O ((n*m + 1)/2)
    3. labelChildren () method
        1. The outer loop iterates over the m-1 children of a star - O(m).
        2. Inside the outer loop, there is a nested loop that iterates from 1 to k - O(n*m).

        3. labelChildren () method is in the inner most for loop of the nested for.

From the above analysis complexity can be given as O $((n*m + 1)/2)^3$) * O(n) * O(m). Which can be simplified as **O (n^5 *m^4).**

Tasks - problem 3

1. **Suggest a suitable name.**

   Tri-nodal amalgamated cyclic star graph.

   Reason:
   1. Tri-nodal - Edges from root to stars and the cycle edges combined, look like a group of triangles and these triangles are formed by connecting 3 (tri) nodes.
   2. Amalgamated – All the stars are connected at the root. (Amalgamate - combine)
   3. Cyclic – Because of the cycles in the graph.
   4. Star – Nodes that are directly connected with the root, look like a star when looked with its children.

2. **Devise the formulae for calculating order and size of the graph.**

   For a given n and m
   - **a.** Order **– n*m + 1**
   - **b.** Size – **n*m + n**

   For a given n and constant m = 3
   - **a.** Order – **3n + 1**
   - **b.** Size – **3n + n = 4n**

3. **Data-structure to store the graph.**

   The best data structure is Adjacency list.
   1. Root node maintains the list of star nodes.
   2. Each star node maintains the list of its children (leaf) nodes. Each star also stores the edge from root to star. Each star nodes also maintains pointers for its previous and next star nodes. Each star also stores the cycle edge connecting to its previous star.
   3. Each leaf node maintains the edge from its star to the leaf node.

```java
class Root extends Node{
    ArrayList<Star> children; // Level 1 nodes.

    Root(int n, int m) {
        children = new ArrayList<>();
        for(int i = 0; i < n; i++) {
            children.add(new Star(m));
        }
        for(int i = 0; i < n; i++) {
            int prevIndex = (i - 1 + n) % n;
            int nextIndex = (i + 1) % n;
            children.get(i).prev = children.get(prevIndex);
            children.get(i).next = children.get(nextIndex);
        }
    }
}
```

```java
class Star extends Node {
    ArrayList<Leaf> children; // Level 2 nodes.
    Star prev; // for calculating the edge connected with the previous star.
    Star next; // for passing the recursion to the next star.
    boolean visited;
    int cycleEdgeValue;
    Set<Integer> starEdgeLabels;

    Star(int m) {
        children = new ArrayList<>();
        visited = false;
        starEdgeLabels = new HashSet<>();
        for (int i = 0; i < m - 1; i++) {
            children.add(new Leaf());
        }
    }
}
```

```java
class Leaf extends Node {

}
```

```java
class Node {
    int value = 0;
    // Star stores the edge value of Root to Star.
    // Leaf stores the edge value of Star to Leaf.
    int edgeValue = 0;
}
```

4. **Assign the labels using algorithm.**

   Labels are successfully assigned using the algorithm. Algorithm can be found in the problem-3 section above.

5. **Store the labels of vertices and weights of the edges as an outcome.**

   Vertex labels and edge weights are successfully stored in the graph. Execution results (graph visuals) can be found in the problem-3 section above.