1. **BitStuffing:**

```
public class Main {

    public static void main(String[] args) {
        // Input data to be bit-stuffed
        String inputData = "01111110 0011110 11111110";
        System.out.println("Original Data: " + inputData);

        String stuffedData = bitStuffing(inputData);//Calling
bitStuffing Function
        System.out.println("Stuffed Data: " + stuffedData);
    }

    public static String bitStuffing(String input) {
        StringBuilder stuffedData = new StringBuilder();//Creates a
mutable string object named "stuffedData"

        int count = 0;
        for (char bit : input.toCharArray()) {//Converts String to a
array of characters
            stuffedData.append(bit);//Add to stuffedData
            if (bit == '1') {
                count++;
            } else {
                count = 0;
            }

            // Inserting '0' after five consecutive '1's
            if (count == 5) {
                stuffedData.append('0');
                count = 0;
            }
        }

        return stuffedData.toString();
    }
}
```

Output

```
java -cp /tmp/DcRAApidpe Main
Original Data: 01111110 0011110 11111110
Stuffed Data: 011111010 0011110 111110110
```

2.MST and SPT:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <climits>
using namespace std;
// Data structure to represent an edge
struct Edge {
char src, dest;
int weight;
Edge(char s, char d, int w) : src(s), dest(d), weight(w) {}
};
// Data structure to represent a disjoint-set
struct DisjointSet {
vector<int> parent, rank;
DisjointSet(int n) {
parent.resize(n);
rank.resize(n, 0);
for (int i = 0; i < n; ++i) {
parent[i] = i;
}
}
int find(int x) {
if (parent[x] != x) {
parent[x] = find(parent[x]);
}
return parent[x];
```

```
}
void unionSets(int x, int y) {
int rootX = find(x);
int rootY = find(y);
if (rank[rootX] < rank[rootY]) {
parent[rootX] = rootY;
} else if (rank[rootX] > rank[rootY]) {
parent[rootY] = rootX;
} else {
parent[rootX] = rootY;
rank[rootY]++;
}
}
};
// Comparator for sorting edges by weight in ascending order
bool compareEdges(const Edge& e1, const Edge& e2) {
return e1.weight < e2.weight;
}
// Kruskal's algorithm to find Minimum Spanning Tree (MST)
vector<Edge> kruskalMST(const vector<Edge>& edges, int numVertices)
{
vector<Edge> resultMST;
DisjointSet ds(numVertices);
// Sort edges by weight
vector<Edge> sortedEdges = edges;
sort(sortedEdges.begin(), sortedEdges.end(), compareEdges);
for (const Edge& edge : sortedEdges) {
int srcSet = ds.find(edge.src - 'a');
int destSet = ds.find(edge.dest - 'a');
// If including this edge doesn't cause a cycle, add it to the MST
if (srcSet != destSet) {
resultMST.push_back(edge);
ds.unionSets(srcSet, destSet);
}
}
return resultMST;
}
// Dijkstra's algorithm to find Shortest Path Tree (SPT)
vector<int> dijkstraSPT(const vector<Edge>& edges, int numVertices,
char source) {
```

```cpp
vector<vector<pair<int, int>>> adjList(numVertices);
for (const Edge& edge : edges) {
adjList[edge.src - 'a'].emplace_back(edge.dest - 'a', edge.weight);
adjList[edge.dest - 'a'].emplace_back(edge.src - 'a', edge.weight);
}
priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
vector<int> dist(numVertices, INT_MAX);
dist[source - 'a'] = 0;
pq.push({0, source - 'a'});
while (!pq.empty()) {
int u = pq.top().second;
pq.pop();
for (const auto& neighbor : adjList[u]) {
int v = neighbor.first;
int weight = neighbor.second;
if (dist[v] > dist[u] + weight) {
dist[v] = dist[u] + weight;
pq.push({dist[v], v});
}
}
}
return dist;
}
int main() {
// Given graph represented by edges
vector<Edge> edges = {
{'a', 'c', 6},
{'a', 'b', 6},
{'a', 'd', 6},
{'b', 'd', 2},
{'c', 'd', 2}
};
const int numVertices = 4;
// Kruskal's algorithm to find Minimum Spanning Tree (MST)
vector<Edge> mst = kruskalMST(edges, numVertices);
// Print MST
cout << "Minimum Spanning Tree (MST):" << endl;
for (const Edge& edge : mst) {
```

```
cout << edge.src << " -- " << edge.dest << " Weight: " <<
edge.weight << endl;
}
// Dijkstra's algorithm to find Shortest Path Tree (SPT)
char sourceVertex = 'a'; // Change the source vertex as needed
vector<int> spt = dijkstraSPT(edges, numVertices, sourceVertex);
// Print SPT
cout << "\nShortest Path Tree (SPT) from vertex " << sourceVertex <<
":" << endl;
for (int i = 0; i < numVertices; ++i) {
cout << sourceVertex << " to " << char('a' + i) << " Distance: " <<
spt[i] << endl;
}
return 0;
}
```

Output

```
/tmp/O1p1nBEHQL.o
Minimum Spanning Tree (MST):
b -- d Weight: 2
c -- d Weight: 2
a -- c Weight: 6

Shortest Path Tree (SPT) from vertex a:
a to a Distance: 0
a to b Distance: 6
a to c Distance: 6
a to d Distance: 6
```