Prob. 1	Prob. 2	Prob. 3	Prob. 4

Problem 1.

just checking degree if it is 1, take edge we are going over vertexes to check their degree, if degree is 1, we take its edge then we go over higher degree vertices, if there aren't connected, we take their smallest cost edge

we can have a set S with unexplored vertices take one vertice, if it is degree=1 put the vertice + the edge in the graph

but we have to connect the vertices of degree 1 first

for proof: we need to show that in the MST, we have 6 unused edges from the original graph.

for vertices, their degree is bigger than 1, we could add other edges but algorithm chooses their min edge so if we add another edges, total cost would be higher.

Problem 2.

This question is well-known as degree sequence problem.

Firstly, we need to be sure the degrees sum up to an even number due to *handshaking lemma* because every edge has two end points and each edge contributes to total degree by 2. Summation of even numbers is even. Therefore,

$$if\ sum(a_1, a_2, \ldots, a_n) = 2 \cdot k + 1\ where\ k \geq 0 \rightarrow No$$

Otherwise, we continue analysis and check another claim to be hold. We sort degrees in descending order to make comparison between high degree vertices and small degree vertices. For n vertices, this can be achieved by a regular sorting algorithm in $O(n \log n)$ time.

a_1	a_2	$\ldots$	a_k	$\ldots$	a_n

Table 1: Degree Sequence after sorting in descending order

For the analysis, we will call vertices from 1 to k, including k, *high degree* vertices, S_H and the rest will be called *small degree* vertices, S_S. Since we have chosen k, it can be at anywhere from 1 to n, more precisely $1 \leq k \leq n$.

We can devise an inequality between S_H and S_S. Each edge that has one end point in S_H, can totally be between two high degree vertices so its two end points in S_H. Since we have k vertices in S_H, we can have $\frac{k \cdot (k+1)}{2}$ edges at most which contributes to degree sum by $k \cdot (k+1)$ via *handshaking lemma*. Moreover, an edge that has one end point in S_H, may have its other end point in S_S so vertices in S_S can contribute to degree sum by their degrees or k if their degrees are larger than k because we have k nodes in the S_H and it must have some of its adjacent vertices in S_S. Moreover, since this degree sum is the maximum possible, sum of degrees of vertices in S_H must be equal or less than above two summation. More precisely , this discussion results:

$$\sum_{i=1}^{k} d_i \leq k \cdot (k+1) + \sum_{i=k+1}^{n} min(d_i, k) \text{ where } 1 \leq k \leq n$$

These inequality can be calculated by two nested loops, one for k and one for i so it gives us $O(n^2)$ algorithm since sorting cost is dominated by this inequality check.

If this inequality doesn't hold, we can immediately say that simple graph from given sequence is impossible because right part is maximum and nodes have degree sum more than possible maximum. However, the other way if this is a sufficient condition is more involved.

For this part, we can use induction in that way: a graph with no edges is a simple graph and for higher degrees, we find last index l of a degree d such that $d_l > d_{l+1}$. Here, we can add or remove one edge between these two vertices without changing degrees of other vertices and coming closer to desired degree by 1. If it is a simple graph before doing addition or deletion, it will still be a simple graph because chosen vertices are different that prevents self loops and their degrees differ that prevents multi-edges.

Problem 3.

here, I don't really get how to calculate the gain of reliability do we have to construct our own function to calculate it? in some sense, we need to bound it but I think it would be easier after asking

for example, we can assume only nodes access file can update them

Problem 4.

So for this problem, we can represent the matrix $n \times n$ as a bipartite graph where we have n nodes on one side, representing the rows and n other nodes on the other side, representing the column. A "1" represent an edge linking its column and row.
For a bipartite graph $G = (\{U,V\},E)$, nodes in U representing the columns and nodes in V representing the rows. Let's denote a node U_i the one representing the i-th column and V_i the one representing the i-th row.
As we can see on Figure 1, to have a diagonalized matrix we need to have an edge linking

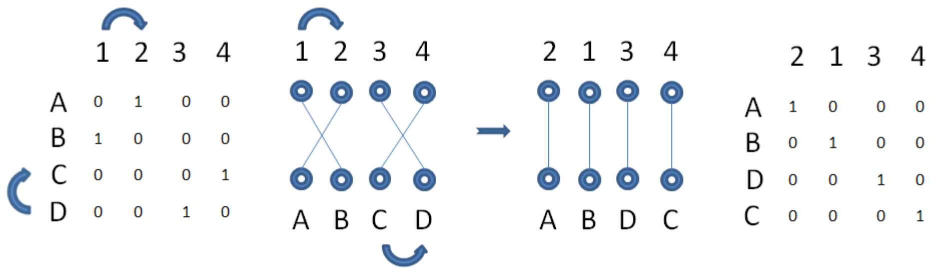

Figure 1: Illustration of swapping colums and rows

U_i and V_i, $\forall i \in \{1,2,\ldots,n\}$. Plus swapping column i with column j means that U_i becomes U_j and U_j becomes U_i. Same logic can be apply to swap rows. So if we have n matching, it means we can diagonalize the matrix by swapping the columns and rows so we end up with n edges linking U_i with V_i, $\forall i \in \{1,2,\ldots,n\}$. Thus our algorithm would just check if the matrix has n matchings.