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

Problem 1.

Since we know that the number of edges is $|V|+5$, we also know that in order to create a tree, we need to remove 6 edges because there will be 6 circles in the initial graph. Thus, we iterate through the graph (vertices) once until we detect a cycle by *a simple Depth First Search(DFS)*. DFS can find cycles in $O(|V|)$ because at most $n-1$ edges can be tree edges. After detecting a cycle, we check the edges we iterated through in order to get the most expensive one and remove it (we write it this way since it doesn't increase complexity but it can also be done by traversing). We repeat the above procedure 6 times. Searching for a circle will cost $O(|V|)$ each time, since we have a sparse graph. Also the search for the most expensive edge is bounded by $O(|V|)$ since not all edges will be part of the circle except for the last iteration. Since we have a certain finite number of searches(6), total cost is linear in the number of vertices:

$$\begin{aligned} Cost &= 6 \cdot (\textit{find cycle by DFS} + \textit{find highest cost edge in cycle}) \\ &= 6 \cdot (O(|V|) + O(|V|)) \\ &= 6 \cdot (O(|V|)) \leq O(|V|) \end{aligned}$$

Proceeding to the proof of correctness and optimality:

After the algorithm terminates our tree has exactly $|V|-1$ edges and since all removed edges belonged to a circle the graph has no more circles now, it is a tree by definition.

As for optimality, suppose there is a different optimal solution. Let O denotes optimal solution and G be our solution. Suppose that O and E differ in one edge only and that $cost(O) < cost(G)$ (Notice that equality gives a different tree but doesn't violate optimality). We also know that since all nodes are included in both minimum spanning trees adding any edge creates a cycle. Therefore, suppose we add to the tree O the edge that exists in G and does not exist in O. Then a cycle is created and by definition this is a cycle contained in the original graph, meaning that this edge has not been removed in G. Then since G is removing the most expensive edges in all cycles remained in the original tree, the only valid assumption is that this edge has the same cost as one or more other edge in the cycle. Otherwise O cannot be optimal since interchanging the edges will create a better solution. This must hold for any edge added to O and thus all edges differing between O and G must be of equal cost, meaning G is equal to the optimal solution.

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	...	a_k	...	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.

Apparently, allocation of each file is an independent event, so the problem could be reduced to figuring out the allocation scheme of one file.

Suppose we have the file F_1, n nodes and a known sequence of retrieval and update requests for this file. As far as retrieval requests are concerned, the order of retrievals isn't of interest to us. What really matters is the frequency with which each node requests this particular file. Moreover, the sequence of update requests, or the node that issues the request does not matter since we know that in an update request all copies of the requested file should be updated in every node. Thus, we are only interested in the sum of the bytes we would have to update.

Therefore, the total cost of an allocation scheme of F_1 would be the sum of the *total retrieval cost*, *the total update cost* minus *the gain in reliability*.

- Total retrieval cost: let the total bytes (of all retrieval requests for file F_1) requested by node i be B_i,then retrieval cost for file F_1 when F_1 is assigned to no node is $\sum_{i=1}^{n} B_i$
- Total update cost: let the total bytes (of all update requests for F_1) requested be U then retrieval cost for F_1 when F_1 is on C nodes is $U \cdot C$.
- The gain in reliability follows the rule of diminishing return; thus, we know that for C copies, $g(C) > g(C+1)$ Example of law of diminishing returns application: $\frac{1}{X}\sum_{i=1}^{X} \frac{1}{2^{i-1}} = D$

It is clear that for C copies of F_1, the total cost depends upon the retrieval requests, so we order the nodes based on the total number of bytes they request of a specific file in a descending order. Initially the *total update cost* is equal to zero and the *total retrieval cost* is maximum. Then we take the first node and since this is the node that requests most bytes of the file we put the file there. For the rest of the nodes the procedure is the following: We consider the next node in order. In order to decide whether to put the file in the next node we calculate the new total cost with the file in that node.

In case we decide to put the file in this node:
The *total retrieval cost* will be decreased by B_i where B_i the total number of bytes that this node will request, so:

$$new\ retrieval\ cost = old\ retrieval\ cost - B_i$$

and the *update cost* will increase by the number of the totally requested since a new copy will be added, so if *update cost* was $C \cdot U$ the *new update cost* will be $(C+1) \cdot U$. Furthermore, the *total reliability gain* will slightly increase.

As a result, we compare the new total cost to the previous one and if it is smaller we add that file in the node. Then we proceed with the next node in the order, until the new total cost is no longer smaller than the previous one. In that case we stop and the allocation scheme is complete.

Proceeding to the proof of correctness:
Let us suppose that the above algorithm is not the optimal. In this case, there should be an optimal allocation that is different than the one suggested by the greedy algorithm described above.

There are two cases: The optimal allocation contains copies that is not contained in the greedy one, or the greedy one contains copies that are not contained in the optimal one. We will prove that both cases are not valid.

1. Suppose there is an equal number of copies in both allocations but there is one copy of the file in a node a the optimal allocation (OPT) instead of the node b in the greedy one (GR).

   It is known that:

   $$cost(OPT) < cost(GR), copies(OPT) = copies(GR)$$

   So:

   $$\begin{aligned} Update\ cost(OPT) &= Update\ cost(GR) \\ Retrieval\ cost(OPT) &= Retrieval\ cost(GR) - B_a + B_b \\ gain(OPT) &= gain(GR) \end{aligned}$$

   Then for *cost(OPT)* to be greater than *cost(GR)* we would need B_a to be greater than B_b. That would mean that node a would be in a higher order than b and if that was the case our greedy algorithm would already have selected b.

2. Suppose there is one extra copy of the file in a node a in the optimal allocation (OPT) that does not exist in the greedy one (GR).

   It is known that:

   $$cost(OPT) < cost(GR), copies(OPT) = copies(GR) + 1$$

   So:

   $$\begin{aligned} Update\ cost(OPT) &= (\frac{Update\ cost(GR)}{C}) \cdot (C+1) \rightarrow Update\ cost(OPT) > Update\ cost(GR) \\ Retrieval\ cost(OPT) &= Retrieval\ cost(GR) - B_a \rightarrow Retrieval\ cost(OPT) < Retrieval\ cost(GR) \\ gain(OPT) &> gain(GR) \end{aligned}$$

   So $cost(OPT) < cost(GR)$ implies that the total cost when including a copy of F_1 in the node a is smaller than when it isn't.

   Then, we can suppose there is no other node with $B_i > B_a$ that is not contained in both allocations, thus our greedy algorithm considered adding the file to the node a and decided not to. That would mean that the total cost of including a copy of F_1 in the node a is greater than when not to, a fact that contradicts our previous assumption.

3. Suppose there is one extra copy of the file in a node b in the greedy allocation (GR) that does not exist in the optimal one (OPT).

It is known that:

$$cost(OPT) < cost(GR), copies(OPT) = copies(GR) - 1$$

So:

$$Update\ cost(OPT) = (\frac{Update\ cost(GR)}{C}) \cdot (C-1) \rightarrow Update\ cost(OPT) < Update\ cost(GR)$$
$$Retrieval\ cost(OPT) = Retrieval\ cost(GR) + B_b \rightarrow Retrieval\ cost(OPT) > Retrieval\ cost(GR)$$
$$gain(OPT) < gain(GR)$$

So $cost(OPT) < cost(GR)$ implies that the total cost when including a copy of F_1 in the node b is greater than when not.

Then we know that our greedy algorithm considered adding the file to the node a and decided to do so. That would mean that the total cost of including a copy of F_1 in the node b is smaller than when not, a fact that contradicts our previous assumption.

As a conclusion, we have proved that in any case our greedy algorithm succeeds in finding an optimal solution.

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

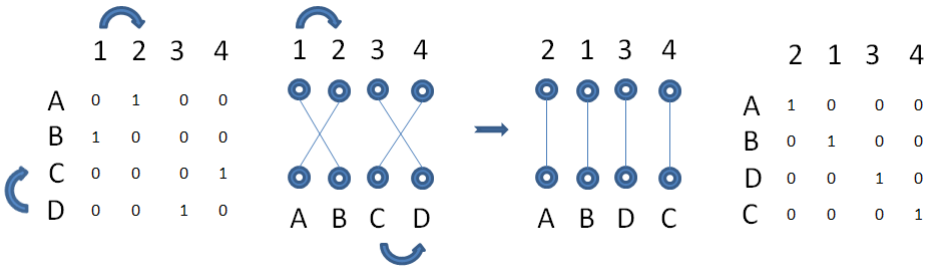

Figure 1: Illustration of swapping colums and rows

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.