Problem 1: Short Answer

(a) What is the running time of Dijkstras algorithm (with min-heap implementation) on a graph with n vertices and m edges?

> Solution.
>
> $O((n+m)\log n)$

(b) Dijkstra's algorithm is guaranteed to work correctly if the graph is directed with edge lengths that are non-negative (so they can be positive or zero, but not negative):

> Solution.
>
> **TRUE**
> If there are negative length edges then $\text{dist}(v)$ can decrease after v has been explored, but if the lengths are non-negative then this cannot occur and hence Dijkstra's algorithm works correctly.

(c) Consider a MST T for undirected $G = (V, E)$. Now suppose for every edge e in G, we replace its edge weight $w(e)$ by $w(e) + 1$, so we increase every edge weight by $+1$. The tree T is guaranteed to still be a minimum spanning tree for this new weighted graph:

> Solution.
>
> **TRUE**
>
> A spanning tree has exactly $n-1$ edges, so increasing each edge weight by 1 increases the weight of all spanning trees by $n-1$. It will not change the relative weights of two spanning trees, so a MST in the original graph will still be a MST in the new one.

(d) Consider a MST T for undirected $G = (V, E)$. Now suppose for every edge e in G, we replace its edge weight $w(e)$ by:

$$\hat{w}(e) = \begin{cases} 2w(e) & \text{if } w(e) > 100 \\ 0 & \text{if } w(e) \leq 100 \end{cases}$$

The tree T is guaranteed to still be a minimum spanning tree for this new weighted graph:

> Solution.
>
> **TRUE**
>
> If an edge e was minimum across some cut $(S, \overline{S})$ in the original weighting scheme then it is still minimum across this same cut in the new weighting scheme (if it has weight ≤ 100 then there may be additional edges of the same new weight but e is still minimum).

Problem 2: MST

For an undirected graph $G = (V, E)$ with weights $w(e) > 0$ for each edge $e \in E$, you are given a MST T. Unfortunately one of the edges $e^* = (u, z)$ which is in the MST T is deleted from the graph G (no other edges change). Give an algorithm to build a MST for the new graph. Your algorithm should start from T.

Note: G is connected, and $G - e^*$ is also connected.

Explain your algorithm in words and use the algorithms from class as black-box subroutines. Clearly state and explain the running time of your algorithm. To receive credit, your algorithm should be correct and faster in $O()$ than building a MST from scratch (so dont use Prim's or Kruskal's, even part of Kruskal's).

Solution.

Let u and v be the two endpoints of e^*. Look at $T - e^*$. Let T_u and T_v be the two subtrees formed by removing e^* from T, such that $u \in T_u$ and $v \in T_v$. We can find the vertices in these 2 components by running DFS on $T - e^*$. We then go through all of the edges of the graph and take the minimum weight edge e' that has one endpoint in T_u and the other in T_v. We add e' to $T - e^*$ to get the new MST.

Running DFS takes $O(|V|)$ time since the graph $T - e^*$ has only $O(|V|)$ edges. Checking each edge takes $O(|E|)$ time. Therefore the algorithm takes $O(|V| + |E|)$ time.

Problem 3: Max-flow

Consider the following problem `Bipartite-`k`-Perfect`:

Input : Undirected bipartite graph $G = (V, E)$
and integer k where $1 \leq k \leq n$.

Output : A k-perfect subgraph if one exists, and NO if none exist in G.

Denote the 2 sides of the bipartite graph as $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$; note $|A| = |B| = n$.

Explain how to reduce the `Bipartite-`k`-Perfect` problem to the max-flow problem. In other words, explain how to solve the bipartite k-perfect problem using the algorithm for solving max-flow as a black-box.

(a) Given an input $G = (V, E)$ and k to the Bipartite-k-Perfect problem, explain how you create the input to the max-flow problem (specify the graph G' and the edge capacities in this new graph). Do not do it for the above example, do it in general.

Solution.

To create G', first add a source vertex s and a sink vertex t to G. Direct all edges of G to go from A to B and assign capacity 1 to each of these edges. Add edges of capacity k from s to each vertex in A, and from each vertex in B to t.

(b) Given a max-flow f^* for the flow network that you defined in part (a), explain how you use f^* to determine if G has a k-perfect subgraph.

Solution.

To check if G has a k-perfect subgraph, check if the max-flow in G' has size kn. If it does, then G has a k-perfect subgraph. The k-perfect subgraph is the set of edges between A and B with non-zero flow in f^*.

(c) What is the running time of your algorithm in terms of the original graph G where $n = |V|$ and $m = |E|$. State whether you are using the Ford-Fulkerson or Edmonds-Karp algorithm (only consider these two which we saw in class); faster is better.

Solution.

This algorithm uses Ford-Fulkerson. Constructing G' takes linear time ($O(n + m)$). Running Edmonds-Karp takes $O(nm^2)$, whereas running Ford-Fulkerson takes $O(mC)$ time. Since $C \leq kn$ then Ford-Fulkerson takes $O(knm)$ time which is better in this case.

Problem 4: Graph algorithms

You are given a directed graph $G = (V, E)$ where each edge e has a length $\ell(e) > 0$. Some of the vertices are marked special. Let $A = \{a_1, \ldots, a_k\}$ be the subset of vertices marked special. The set A may be large, e.g., it may be that $|A| = O(n)$.

Devise an algorithm to compute the minimum total length from every vertex to the nearest special vertex. In other words, we want, for every $v \in V$ the minimum over all $a_i \in A$ of the distance from v to a_i.

Use the algorithms from class as a black-box subroutine. Get the best running time in $O()$ notation in terms of $n = |V|$ and $m = |E|$.

(a) There is **only 1 special vertex** so $|A| = 1$. Find the minimum distance from every vertex to this one special vertex a_1.

Solution.

First, create the reverse graph $G^R = (V, E^R)$ $(E^R = \{(v, u) : (u, v) \in E\})$. Then, run Dijkstra's algorithm on G^R from a_1. This finds the shortest path from a_1 to all other vertices. Reverse these paths to find the paths with minimum distance from every vertex to a_1. The running time is $O((n + m) \log n)$ since 1 run of Dijkstra's algorithm is used.

(b) There are **many special vertices**.

Hint: transform the graph G so that you need just 1 run of Explore, DFS, BFS, or Dijkstra's algorithm.

Solution.

First, create the reverse graph $G^R = (V, E^R)$ $(E^R = \{(v, u) : (u, v) \in E\})$. Add a new vertex v to this graph, and add edges with 0 weight from v to each special vertex. Then, run Dijkstra's algorithm on this graph from v. Since the added edges have 0 weight, they do not affect the minimum distance paths. For a vertex z, its distance from v in this next graph will be the minimum distance from the closest special vertex. The running time is $O((n + m) \log n)$ since 1 run of Dijkstra's algorithm is used.