Problème 1.

Consider we want to build McMaster Maps, the best online route planning tool in existence. The developers of McMaster maps have decided to represent road information in terms of a a massive graph in which nodes are crossing and the edges are the roads between crossings. The developers of McMaster Maps will be optimised toward computing the directions to McMaster University is the *only destination that matters*. Hence, McMaster Maps will be optimised toward computing the directions to McMaster University. To do so, McMaster Maps maintains *single-sink shortest path index* that maintains the shortest path from any node to McMaster University. This index is represented by the typical *path* and *cost* arrays as computed by either `Dijkstra` or `Bellman-Ford`. Once in a while, an update to the road network happens: The weight of a single edge changes (this can represent adding and removing edges: addition changes the weight of the edge from ∞ to a numeric value and removing changes the weight of the edge from a numeric value to ∞)

ⓘ P1.1

Given the road network as a graph $\mathcal{G}$, the *shortest path index*, and the edge that was changes, write an algorithm that determines whether the shortest path index is still valid. You may assume the graph is already updated. Argue why your algorithm is correct. What is the complexity of your algorithm?

```
**Algorithm** CheckShortestPathIndexValidity($\mathcal{G}, path, cost, u, v, w$)
**Require:** $\mathcal{G}$, $path$ and $cost$, edge $(u, v)$ with new weight $w$
**Ensure:** Returns true if the shortest path index is still valid, false otherwise
    **if** $cost[u] + w < cost[v]$ **then**
        **return** false
    **end if**
    **if** $path[v] = u$ and $cost[u] + w > cost[v]$ **then**
        **return** false
    **end if**
    **return** true
```

The shortest path index is invalid in two cases:

- If the new shortest path from u via edge (u, v) is shorter than the current shortest path from v to the destination. This is checked by the condition $cost[u] + w < cost[v]$.
- If v's current shortest path to the destination goes through u and the new weight w makes this path longer than v current shortest path cost. This is checked by $cost[u] + w > cost[v]$. If both of these conditions met, shortest path index remains valid.

Therefore the algorithm is correct □

The algorithm performs a constant number of comparison and array lookups. Thus it is $O(1)$

P1.2

Assume the shortest path index is no longer valid: provide a modification of `Dijkstra` algorithm that restores the index to a valid state without recomputing all shortest paths. You may assume the graph is already updated. Argue why your algorithm is correct.

```
**Algorithm** UpdateSingleSinkShortestPath($G, w, e = (u, v)$, path, cost)
**Require:** Graph $G = (V, E)$, weight function $w : E \to \mathbb{R}^+$, updated edge $e = (u, v) \in E$
    with new weight $w'(e)$, path$[1..|V|]$ and cost$[1..|V|]$
**Ensure:** Updated shortest path tree arrays path$[1..|V|]$ and cost$[1..|V|]$
    Initialize min-priority queue $Q$ with $(v, \text{cost}[u] + w'(e))$
    **while** $Q$ is not empty **do**
        Extract vertex $x$ with minimum cost from $Q$
        **for** each neighbor $y$ of $x$ **do**
            **if** $\text{cost}[x] + w(x, y) < \text{cost}[y]$ **then**
                $\text{cost}[y] \leftarrow \text{cost}[x] + w(x, y)$
                $\text{path}[y] \leftarrow x$
                **if** $y$ is in $Q$ **then**
                    Decrease key of $y$ in $Q$ to $\text{cost}[y]$
                **else**
                    Insert $y$ into $Q$ with key $\text{cost}[y]$
                **end if**
            **end if**
        **end for**
    **end while**
    **return** path, cost
```

Correctness: Let T be the shortest path tree rooted at the sink vertex t before the edge update, and let T' be the tree true shortest path tree after the edge update. We define a following boundary function:

$$B(i) = \{v \in V : \text{dist}_{T'}(v, t) \leq i \ \wedge \ \text{dist}_{T'}(v, t) < \text{dist}_{T}(v, t)\}$$

or $B(i)$ is the set of vertices whose true shortest path to t is at most i and strictly less than their distance in T. The following invariant will hold of the while loop:

$$\forall v \in B(i) : \text{cost}[v] = \text{dist}_{T'}(v, t) \ \wedge \ \text{path}[v] \text{ is correct}$$

For base case, $i = 0$, $B(0)$ is empty, thus invariant holds

Suppose the invariant holds for k, that is at k-th iteration $B(k)$ and invariant holds. Let x be the vertex extracted from Q in the $k + 1$-th iteration. We argue that $x \in B(k + 1) \setminus B(k)$, or

$\text{dist}_{T'}(x,t) = k \wedge \text{dist}_{T'}(x,t) < \text{dist}_T(x,t)$

First, note that x must have been inserted into Q by some previous iteration, say when visiting a vertex $y \in B(j)$ for some $j < i$. By the induction hypothesis, $\text{cost}[y] = \text{dist}_{T'}(y,t)$ at that time, so the tentative distance $\text{cost}[y] + w(y,x)$ used to insert x into Q equals $\text{dist}_{T'}(x,t)$. Since x is extracted in the $k+1$-th iteration, we have $\text{dist}_{T'}(x,t) = k+1$. Moreover, we must have $\text{dist}_{T'}(x,t) < \text{dist}_T(x,t)$, for otherwise x would have been visited before via a shorter path in T, contradicting the fact that $\text{dist}_{T'}(x,t) = i$. Thus, $x \in B(k+1) \setminus B(k)$. The algorithm then correctly updates $\text{cost}[x]$ to $\text{dist}_{T'}(x,t)$ and $\text{path}[x]$ to its parent y in T'. Furthermore, for each neighbor z of x, if $\text{cost}[x] + w(x,z) < \text{cost}[z]$, then the path to z through x is shorter than its current path, so the algorithm correctly updates $\text{cost}[z]$ and $\text{path}[z]$ and inserts z into Q with the updated distance.

Therefore, after the $k+1$-th iteration, the invariant holds for all vertices in $B(k+1)$, including the newly added vertex x and possibly some of its neighbors. By induction, the invariant holds for all i. $\square$

P1.3

Explain which graph representation you used for your algorithm and what the complexity of your modified-`Dijkstra` algorithm is using this graph representation.

note: Express the complexity in terms of the number of nodes affected by the change. For example, use a notation in which C is the number of nodes affected by the edge change, $\text{incoming}(C)$ the incoming edges of C, and $\text{outgoing}(C)$ the outgoing edges of C.

The algorithm use an adjacency list representation, as each vertex maintains a list of its outgoing edges.

The overall complexity is $O((|\text{outgoing}(C)||C|) \log |C|)$

The main loop run until Q is empty. Per iteration, it extracts the vertex x with minimum distance from Q, which takes $O(\log |C|)$ time, where $|C|$ is the number of affected vertices.

For each outgoing edge (x,y) of x, update the distance and parent of y and either decrease its keep in Q or insert into Q. The operation takes $O(\log |C|)$ time worst-case.

The total number of iterations of the inner loop is bounded by $|\text{outgoing}(C)|$ as each outgoing edge of an affected vertex is processed at most once.

Space complexity is $O(|C|)$, as min-priority queue Q stores at most one entry per affected vertex.

P1.4

What is the worst-case complexity of your solution if you use the other graph representation?

If an adjacency matrix representation is used, the graph is represented by a $|V| \times |V|$ matrix, where $|V|$ is the number of vertices in the graph.

With this representation, the main difference in the algorithm is in the loop that iterates over the neighbors of the current vertex x. In an adjacency matrix, finding the neighbors of x requires scanning the entire row corresponding to x in the matrix, which takes $O(|V|)$ time. Therefore, the overall time complexity of the modified Dijkstra's algorithm using an adjacency matrix becomes:

$$O(|C| \cdot |V| \log |C|)$$

If all vertices are affected ($|C| = |V|$), the complexity becomes $O(|V|^2 \log |V|)$.

Space complexity is $O(|V|^2)$, as the matrix requires storing $|V|^2$ entries.

Problème 2.

Consider a company managing many servers placed all over the world. The company wants to add network connections between a minimal amount of servers to ensure that there is a path of communication between all pairs of servers. While researching this problem, the company was advised that some connections can be built more reliable than others: according to the consulted contractors, the probability that a connection (m, n) between server m and n will work at any given time is $p(m, n)$ (We have $p(m, n) = p(n, m)$). The company wants to *minimize* the number of connects, which *maximising* the probability that all servers are connected to each other at any given time. We will help the company out in their challenge to figure out which connections they need to built.

P2.1

Model the above problem as a graph problem: what are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

Nodes: Each server is represented by a node (vertex) in the graph. Denote the set of all servers as V. **Edges:** potential network connections between servers are represented by edges in the graph. An edge (m, n) exists between node m and n if a connection can be built. Denote set of all possible connections as E. **Weights**: Each edge (m, n) has a weight $-\log(p(m, n))$ representing the probability that the connection will work at any given time.

The problem is to find a *minimum spanning tree* (MST) of the graph that connects all servers, such that the sum of the weights of the edges in the MST is maximized. Or it can be stated as:

> Find a subset of edges $E' \subseteq E$ such that the graph (V, E') is a minimum spanning tree and the sum of the weights of the edges in E' is maximized.

P2.2

Provide an algorithm `NetworkPlan` to find the network connections to build. Explain why your algorithm is correct.

```
Algorithm NetworkPlan
Require: $G$
Require: $p$
  $E \leftarrow$ //Set of edges in the graph
  for each edge $(m, n)$ in $G$ do
    $w(m, n) \leftarrow -\log(p(m, n))$//Compute edge weight
    $E \leftarrow E \cup (m, n)$
  end for
  $E \leftarrow$ SortEdges($E$)//Sort edges by weight in ascending order
  $MST \leftarrow \emptyset$
  $DS \leftarrow$ MakeSet($G.V$)//Initialize disjoint sets
  for each edge $(m, n)$ in $E$ do
    if FindSet($DS, m$) $\neq$ FindSet($DS, n$) then
      $MST \leftarrow MST \cup (m, n)$//Add edge to MST
      UnionSets($DS, m, n$)//Union sets containing $m$ and $n$
    end if
  end for
  return $MST$
```

It follows Kruskal's algorithm to find MST. Since we are using negative logarithm of the probabilities as edge weights, the MST found by Kruskal will maximise the probability.

P2.3

Explain which graph representation you used for your algorithm and what the complexity of your algorithm is using this graph representation.

It uses an adjacency list representation, where each node maintains a list of its neighboring nodes and the corresponding edge weights. Using an adjacency list, the time complexity of Kruskal's algorithm is $O(E \log E)$, where E is the number of edges in the graph. This is because sorting the edges takes $O(E \log E)$ time, and the main loop of Kruskal's algorithm takes $O(E)$ time using a disjoint set data structure to efficiently check for cycles.

? **P2.4**

What is the worst-case complexity of your solution if you use the other graph representation? Explain your answer.

Constructing the adjacency matrix representation of the graph takes $O(V^2)$ time, where V is the number of vertices in the graph. Sorting the edges takes $O(V^2 \log V^2) = O(V^2 \log V)$ time. The main loop of Kruskal's algorithm takes $O(E \log V)$ time, where E is the number of edges. This is because we need to iterate over all edges ($O(E)$) and perform the `UnionSets` operation, which takes $O(\log V)$ time using an efficient disjoint set data structure like union-by-rank and path compression.

Therefore, the worst case scenario would be $O(V^2 + V^2 \log V + E \log V) = O(V^2 \log V)$