Please read the instructions carefully:

1. There are two stages to each assignment:

   (a) For the first stage, you need to submit your homework solution as a single pdf file to NYU Classes. It can be either a scanned copy of your written solution or photos joined into a pdf. You are also allowed to typeset the solutions in MS Word/LaTeX. In case of scanned copies it is your responsibility that the solution is readable. There are no late submissions accepted, as sample solution outlines (to be used for the second stage of the homework) will be given out on the day of the submission of the first stage of the homework. You should leave around 5 to 6 cm of space between each part of the answers for the second stage of the homework.

   (b) For the second stage of the assignment, you are required to critique your own solutions and analyze the mistakes that you made. Based on the outlines provided, you need to write the self criticism in the empty space that you left in your homework. Do not rewrite your solution, you need to understand and point out the mistakes that you made when writing the original solutions.

   A good analysis of your solution should summarize the key steps, ideas or techniques that you previously missed. It should prompt you to reorganize your thoughts and think about what you can do to improve the solutions. Even in the case where you completely missed the solution it is insightful to point out why the approach adopted by you does not work and identify the step where it fails. This will allow you to get full credit on the self evaluation part of the homeworks and of course help you in understanding the question.

   Mark correct solutions in green, incorrect solutions in red and if you are unsure about any solutions write it in blue.

2. For each question where you are required to design an algorithm, you need to clearly explain the algorithm and prove the running time complexity as asked in the question.

3. You can assume $O(1)$ complexity of arithmetic of numbers (e.g., multiplication/division of numbers and calculations of $n!$) unless otherwise stated in the question.
Question 1. (10 points) DFS mechanics

1. Start: 1 2 3 17 9 4 11 10 5
   End: 16 15 8 14 7 12 13 6

2. Start: 1 2 3 17 4 5 6 18 7 12 8
   End: 22 21 16 20 15 14 11 19 10 13 9

Question 2. (10 points) SCC mechanics
Along with the DFS on the original graph, we get the following on the reverse graph

1. Start: 3 5 13 1 7 15 11 9 17
   End: 4 6 14 2 8 16 12 10 18
   Which gives us that the all components are singular vertices

2. Start: 1 2 3 7 11 13 17 9 18 12 17
   End: 6 5 4 8 16 14 21 10 19 15 20
   Which gives us that the components are (A, B, C), (D), (E, F, J), (G, I, K), (H)

Question 3. (10 points) Dijkstra mechanics

- Vertex: 0 1 2 3 4 5
- Parent: 0 0 0 5 0 0
- Distance: 0 8 6 16 3 5

Question 4. (20 points) MST mechanics
Edges are in the order of addition to the tree/forest.

1. (0 − 1), (0 − 9), (9 − 2), (9 − 4), (4 − 7), (3 − 7), (4 − 5), (4 − 8), (4 − 6)
2. (0 − 1), (3 − 7), (0 − 9), (9 − 2), (9 − 4), (4 − 7), (4 − 5), (4 − 8), (4 − 6)

Question 5. (35 points) Installing fire stations

1. (a) Look at the vertices with in-degree 0. To keep these neighbourhoods safe, we need to have a fire station installed at these stations. And by installing a fire station at these nodes, we have kept each neighbourhood safe.

   (b) For a general city structure, which is not a DAG, we first find the SCC DAG of the graph. Now we can see that on the SCC DAG, we should install a fire station at any one of the nodes in the components with in-degree 0 of the SCC DAG. This is $O(n + m)$ as we are only doing a SCC algorithm which is linear.

2. As in the previous part, we need to install a fire station in one of the nodes of the components of the SCC DAG with in-degree 0. But now instead of choosing any random vertex, we choose the vertex with the minimum cost in the component. This is also $O(n + m)$ as we are only doing a simple SCC algorithm.
3. Now we need to install a fire station in every SCC component of the graph. Similar to the previous part, we find the SCC DAG and in each component, we find the vertex with the minimum weight. This is also $O(n + m)$.

**Question 6. (15 points) Nearest fire station**

Instead of doing a Dijkstra from each vertex with a fire station installed, we can do a Dijkstra where we initialize the starting queue with each of these neighbourhoods, with distance 0. This is similar to finding the *boss* of a vertex in the code presented in class.

Which means that we are only doing a single Dijkstra, which has complexity $O((n + m) \log n)$.

**Question 7. (15 points) Fulfilling the network**

Here we can modify Kruskal’s algorithm to suit our needs. At the start of Kruskal’s algorithm, instead of starting with a disconnected set of nodes; do a `join(x, y)`, for every edge $(x, y)$, such that $\beta(x, y) = 1$.

This will give us a set of connected components, effectively giving us a smaller graph (where each vertex is a connected component in the original graph).

Now we can find the MST of this smaller graph using Kruskal’s algorithm in $O((n + m) \log n)$.

**Question 8. (15 points) All shortest paths**

Modify Dijkstra’s algorithm, to store two integers for each vertex

1. Shortest distance from $s$ to the vertex
2. Number of shortest paths from $s$ to a vertex

Now to calculate the number of shortest paths for a new vertex which is not currently in the Dijkstra tree, we can just add the shortest paths for each of its predecessors, if the shortest path for the predecessor + edge from predecessor to vertex is the same as the current shortest path to the vertex.

This is the same as doing a Dijkstra with just one more calculation per step, giving the same complexity, $O((n + m) \log n)$.

**Question 9. (20 points) Minimum Dijkstra tree**

We now need to modify dijkstra’s algorithm to keep track of the total weight of the tree. Whenever we have a choice of a predecessor for a vertex, which gives us two different shortest paths to the vertex, we chose the predecessor with the shorter edge weight.

E.g. in the case of the given graph, after we have made the minimum weight tree for the vertices, $A, B, C$, when we want to chose the predecessor to $D$, which could be either $B$ or $C$, we chose $C$ as the edge $CD$ has weight 5, which is smaller than $BD$ which has weight 6.

This has the same complexity as Dijkstra as we can keep the current edge weight for each node and just update it when we see another shortest path. Hence giving a time complexity of $O((n + m) \log n)$.

**Question 10. (20 points) A dangerous gamble**

Let us call the two parts $A$ and $B$, and let the shady neighbourhoods that sell the two parts be $a_1, a_2, \ldots, a_x$ and $b_1, b_2, \ldots, b_y$, respectively. We can pick either part first and then go to the second part in time less than $T$. After which we can go to the garage.

The algorithm works as follows:

1. First run Floyd-Warshall to find all pair shortest paths between all pairs of vertices, to obtain $D[1 \ldots n][1 \ldots n]$
2. Create a new graph of only the house, garage and two copies of each neighbourhood, which sells any of the items. Now add the following edges in this graph, as shown in fig. 1:

   (a) Add an edge from \( s \) to \( a_i \) and \( b_j \) for all \( i \) and \( j \), with weight \( D[s][a_i] \), \( D[s][b_j] \), respectively.
   (b) Add an edge from \( a_i \) and \( b_j \) to \( t \) for all \( i \) and \( j \), with weight \( D[a_i][t] \), \( D[b_j][t] \), respectively.
   (c) Add an edge from \( a_i \) to \( b'_j \) with weight \( D[a_i][b_j] \) if and only if \( D[a_i][b_j] \leq T \). If it is greater than \( T \), then don’t add this edge.
   (d) Similarly, add an edge from \( b_j \) to \( a'_i \) with weight \( D[b_j][a_i] \) if and only if \( D[b_j][a_i] \leq T \). If it is greater than \( T \), then don’t add this edge.

3. Now run Dijkstra from \( s \) to \( t \) in this super graph and return the distance.

We create the super graph because we need the distance between the two shady neighbourhoods that we visit to be less than \( T \) but we don’t care how much smaller than \( T \) they are. We only care about minimizing the total distance but keeping the middle path less than \( T \).

**Question 11. (25 points) Super independent**

First thing to notice is that even if we not given a root in this tree, we can choose an arbitrary root and direct the edges away from it, to get a rooted tree.

Now the idea is to do dynamic programming on the tree in a bottom up manner.

We will store a \( DP[1 \ldots n] \) array, where \( DP[i] \) will store the max cost independent set in the subtree of node \( i \).

To find the max cost independent set at \( i \), we have two choices, we either include \( i \), or we do not include \( i \). If we do include \( i \), then we can’t include any of the children of \( i \), so we then would look at the max cost independent sets of the grandchildren of \( i \) and add them up. Else if we don’t include \( i \), then we just add the max cost independent sets of all children of \( i \). Which gives us the following
DP

\[ DP[i] = \max \left\{ \sum_{j \in \text{Children}(i)} DP[j], \text{cost}(i) + \sum_{j \in \text{Grand-Children}(i)} DP[j] \right\} \]

We haven’t included \( i \) in the independent set
We have included \( i \)

**Question 12. (30 points - Extra credit) Collecting flowers**

Look at the attached code on the website...

Simple idea: First collapse the graph into components of the same color. Next do a DFS/UnionFind for each pairs of colors which share an edge. For each pair of colors we get the max component we can obtain using only these two colors. Now take the max over all pairs of colors that occur in the graph.