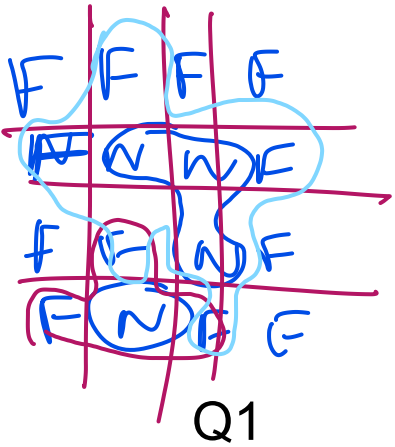


DSA Assignment 4

Siddhant Bali (2022496)

Q1

This code implements flood filling on a 2D board. It defines a structure for the board (BOard) and provides functions to fill regions of 'N' characters with 'F'.

The BOard structure represents the 2D board with methods for accessing and modifying elements, checking validity, and checking if a cell is unfilled.

The floodFill function recursively fills neighboring cells with 'F' if they are valid and unfilled.

The floodreport function iterates through the board, finds the first 'N' cell, and starts flood filling from there.

The printBOard function prints the resulting board after flood filling.

Flood Fill: O(N*M) where N is the number of rows and M is the number of columns in the board.

Overall: O(NM) as the flood report function iterates through every cell of the board, and for each unfilled cell, it calls the flood fill function which takes O(NM) time in total.

Q2

This code finds pairs of elements from an array whose division results in a given remainder. It defines a structure tuplefind to find such pairs.

The tuplefind structure takes an array and a remainder as input.

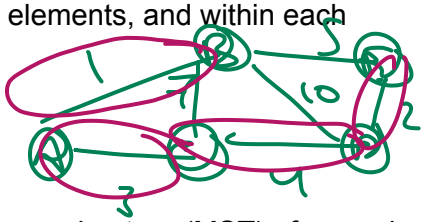
It finds unique elements in the array and iterates through each unique pair of elements.

If the elements are different and the division result has the desired remainder, it adds the pair to the result.

The printPairs function prints the pairs found.

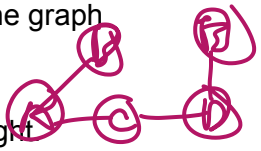
Finding Pairs: O(N^2) where N is the number of unique elements in the array because it iterates through each unique pair.

Overall: O(N^2) because the main loop iterates through unique elements, and within each iteration, it iterates again to find pairs.



Q3

This code implements Kruskal's algorithm to find the minimum spanning tree (MST) of a graph. It defines structures for nodes (enode) and disjoint sets (disJointSets), and a class for the graph (gph).



The enode structure represents an edge with its start and end points along with the weight.

The disJointSets class implements disjoint-set data structure operations like finding parent and merging sets.

The gph class represents the graph with methods to add edges and compute the MST using Kruskal's algorithm.

The make function creates a graph from a set of coordinates, calculating weights as the Manhattan distance between points.

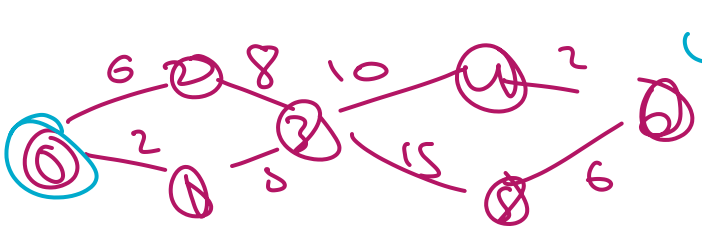
In the main function, a set of coordinates is provided, a graph is created, and then Kruskal's algorithm is applied to find the minimum fuel required.

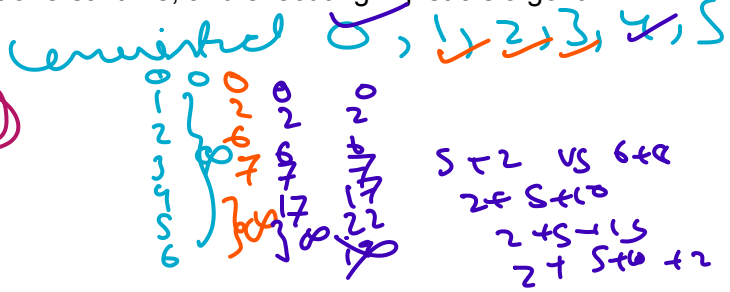
Kruskal's Algorithm: O(ElogE), where E is the number of edges. Sorting of edges takes O(ElogE) time, and for each edge, the find and gmrg operations on disjoint sets take nearly constant time.

Overall: O(ElogE), where E is the number of edges. This is because the time complexity of Kruskal's algorithm dominates the overall time complexity of the program.

Q4

This code implements Dijkstra's algorithm to find the minimum traversal time in a graph with weighted edges. It defines functions for adding edges to the graph, creating the graph, calculating traversal time, finding the minimum traversal time, and executing Dijkstra's algorithm.





The addEdge function adds an edge between two vertices with their associated weights to the adjacency list.

The makegph function creates a graph from a set of edges provided as input.

The Travtime function calculates the total traversal time based on the parameters a, b, and t.

The mintime function finds the minimum traversal time within a specified range of time.

The travmin function calculates the minimum traversal time for a given vertex and distance.

The dijkstra function implements Dijkstra's algorithm to find the minimum traversal time from the source to the destination vertex.

In the main function, the number of vertices and edges are taken as input, the edges are read, and Dijkstra's algorithm is applied to find the minimum traversal time.

Dijkstra's Algorithm: O((V + E)logV), where V is the number of vertices and E is the number of edges. Each vertex is visited once, and for each vertex, its adjacent edges are explored. The priority queue operations take O(logV) time.

Overall: O((V + E)logV). This is because the time complexity of Dijkstra's algorithm dominates the overall time complexity of the program.

Q5

This code performs topological sorting on a directed acyclic graph (DAG) using a modified queue data structure. It defines a custom queue (queue) with limited capacity and provides functions for pushing, popping, accessing the front element, and checking emptiness.

The queue structure is defined with methods to handle operations like pushing, popping, accessing the front element, checking emptiness, and retrieving the size.

The inputg function takes input for the number of vertices, edges, and the capacity of the queue, along with the required edges.

The gphmake function constructs the adjacency list representation of the graph and maintains an index of incoming edges for each vertex.

The topologicalSort function performs topological sorting using the modified queue. It iteratively removes vertices with no incoming edges, updating the index accordingly.

In the main function, input is taken, the adjacency list and index are constructed, and topological sorting is applied to find the minimum cycles required to process all vertices.

Topological Sorting: O(V + E), where V is the number of vertices and E is the number of edges. Each vertex is visited once, and for each vertex, its adjacent edges are explored.

Overall: O(V + E). This is because the time complexity of topological sorting dominates the overall time complexity of the program.