Exercise 1:

1.1

Exercise 1:

1)

1 - (2, 3, 4)

2 - (1, 3, 4)

3 - (1, 2, 4)

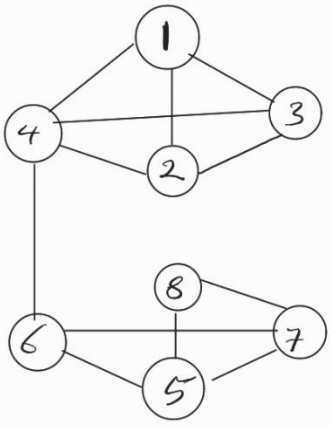
4 - (1, 2, 3, 6)

5 - (6, 7, 8)

6 - (4, 5, 7)

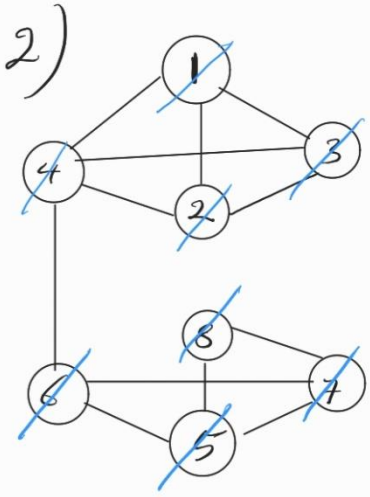
7 - (5, 6, 8)

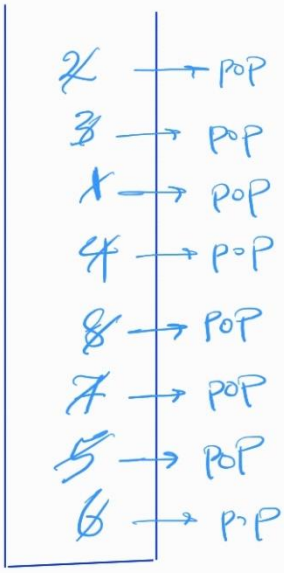
8 - (5, 7)



1.2

2)



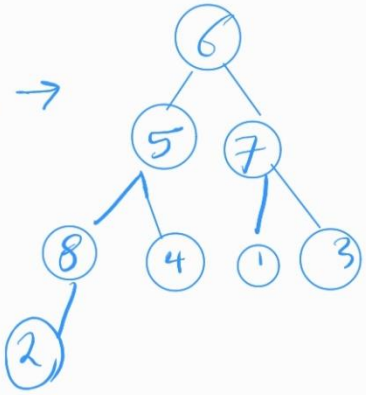


visited Array

1	2	3	4	5	6	7	8
false	false	false	false	false	false	false	false
True	True	True	True	True	True	True	True

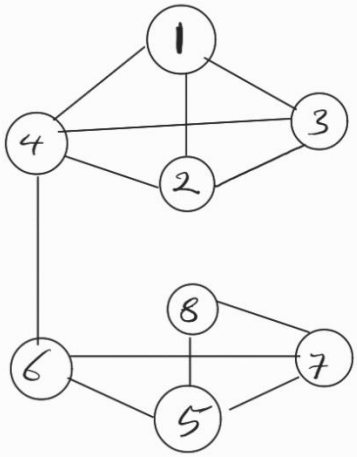
output: 6 5 7 8 4 1 3 2

possibility tree →



1.3

3)



visited Array :

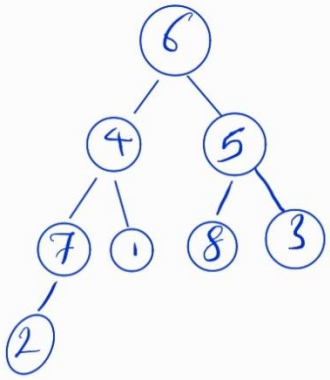
1	2	3	4	5	6	7	8
~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1

deque deq deq deq deq deq deq deq

Queue : ~~6~~, ~~4~~, ~~5~~, ~~7~~, ~~1~~, ~~8~~, ~~3~~, 2

print output : 6, 4, 5, 7, 1, 8, 3, 2

possibility tree →



Exercise 2: The three mentioned algorithms are designed for connected, weighted graphs with only positive labels (edge weights). Fore weighted graphs with negative labels we need to modify the algorithms otherwise it will end up to wrong results.

1. Kruskal: it is a greedy algorithm designed for connected, undirected graph with non-negative labels (edge weights) and in this algorithm we sort the edges in non-decreasing (ascending) order. If the weights' values be negative, the sorting part will be challenging, because it is possible that negative cycles happen, and the algorithm probably won't guarantee creating a correct MST.
2. Prims: a greedy algorithm designed for connected, undirected graph with non-negative edge weights. If we have a graph with negative weights, algorithm might get trapped in the negative cycles, because this algorithm is based on choosing the smallest edge, so it might not end up to a correct MST.
3. Dijkstra: is a shortest-path algorithm designed for graphs with positive values, which uses a priority queue to greedily choose the vertex with the smallest distance from the goal (target, source). so, when we have negative weights in the graph, there is possibility of (negative) infinite loops, so the algorithm might get stuck in infinitive decreasing loop and never reach the goal.

Exercise 3:

Exercise 3 :

Step	Selected islands	Unselected islands	Selected Bridge (Edge)
0	{1}	{2,3,4,5,6,7,8}	none

1 {1,8} {2,3,4,5,6,7} (1,8) – Distance: 120

2 {1,2,8} {3,4,5,6,7} (2,8) – Distance: 155

3 {1,2,4,8} {3,5,6,7} (2,4) – Distance: 175

4 {1,2,4,5,8} {3,6,7} (4,5) – Distance: 160

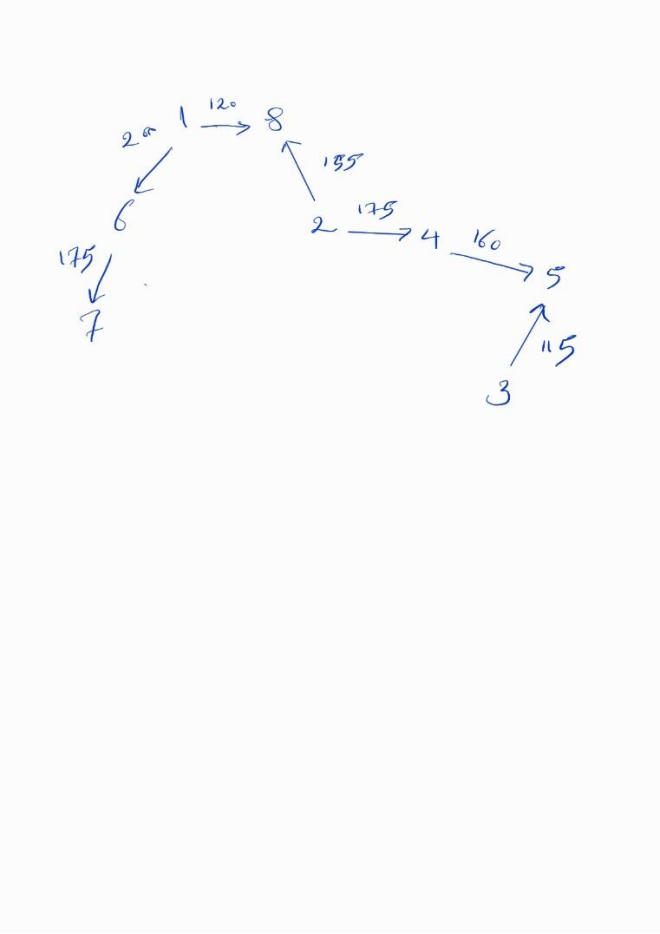
5 {1,2,3,4,5,8} {6,7} (3,5) – Distance: 115

6 {1,2,4,5,6,8} {7} (1,6) – Distance: 200

7 {1,2,4,5,6,7,8} {} (6,7) – Distance: 175

path = 120 + 155 + 175 + 160 + 115 + 200

+ 175 = 1100



Exercise 4: Finding the longest possible path in the graph is a little bit challenging, especially if the graph has cycle. Finding the longest possible path questions are usually considered as NP-hard, in other words there is no known polynomial-time algorithm to find this path for all types of paths. If the graph is a directed acyclic graph (DAGs) finding the longest possible path is less challenging and more manageable.

1. Undirected graph: to find the longest possible path we need to convert the graph to a directed acyclic graph (DAG). To do the conversion, for each undirected edge (u,v) with weight w, we can make two directed edges such as, (u,v) with weight w and (v,u) with weight w. After the conversion, we can apply the longest path algorithm.
   Step1: was the converting the undirected graph to a directed acyclic graph.
   Step2: create the topological ordering of all vertices, we can use depth-first search.
   Step3: initialize an array of dist[] which its length is equal to number of vertices and set all the values in the array to negative infinity, but for the first value (start) initialize it to 0, dist[start]=0

Step4: for each vertex u, assign values to the distances of its neighbours v as following: dist[v]= max(dist[v],dist[u]+weight(u,v)), we use max because we are trying to find the longest path, so we need to consider the max distance

Step5: when the above iteration is done for all u vertices, the value of dist[goal] is in fact equal to the length of the longest acyclic path from start to goal.

Time complexity is O(|V|+|E|)

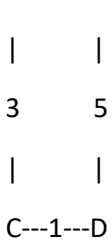
2. Directed graph: for this case, to find the longest path we don't need to do the conversion that we did in the undirected graph, and we can simply apply the longest path algorithm directly on the graph, which includes the steps are mentioned above (excluding the conversion step)

Exercise 5:

No, this greedy strategy does not always find a shortest path from start to goal. If between start and goal there are more than one shortest path, this greedy algorithm might find only one of them, not necessarily the shortest path. Or if we have a graph which contains negative weight cycles, the greedy algorithm might probably not terminate or produce wrong results, because in fact negative weight cycles cause infinite decreasing path weights and the algorithm may keep cycling. So having a graph with negative weight cycles shows this algorithm might not work.

in the below example, the greedy algorithm will choose A-B-D as the shortest path for A to D with the total weight of 7 , but the actual shortest path from A to D is A-C-D with the total weight of 4

Counterexample:



Exercise 6:

Representation 1 of Graph G:

- deleteEdge(i,j): deleting an edge in this representation means changing the value of (entries of)A[i,j] to false, because it is only true if the edge(i,j) exists, since we are deleting it we need to change it to the false. To do so, we need to find the specific element A[i,j] in the matrix, which is a 2D Boolean array and modify it to false. Because finding and modifying a specific element in a 2D Boolean array is constant time, this operation takes a constant time O(1).

- delteIncidentEdges(i): to do this operation, we need to modify (update) the values of all entries in i-th row and i-th column of the matrix to the false, because in fact all the entries in the i-th row and all the entries in i-th column are incidents to the vertex i, so we need to modify all of them to false (because the edges won't exists after deletion). To do so, we need to modify n entries (element) in i-th row and n entries in i-th column, so this operation takes O(n) time.

Representation 2 of Graph G:

- deleteEdge(i,j): to do this operation, we need to find and remove the node that represents vertex j in the list of neighbours of vertex i. The worst-case scenario for vertex j from the list of neighbours of vertex i we need to go through all the list of neighbours of i and find j, which will take O(d(i)) time.
- delteIncidentEdges(i): we need to find and remove all the d(i) nodes that are neighbours of vertex i. To do so, we need to traverse through the entire list and remove each node that is neighbours of i, which takes O(d(i)) time for each node. So, the total time complexity of this operation is $O(d(i)^2)$.