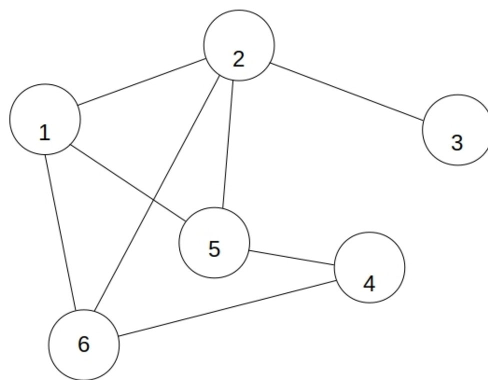
Question 1: (10 Marks)

Let G be a graph with n vertices and m edges.

What data structure would you choose to represent the graph, so that it is possible to quickly determine the number of neighbours shared by two arbitrarily selected vertices? Explain your choice.

For example, in this graph Vertex 3 and Vertex 6 share exactly one neighbour (Vertex 2).



Solution:

I would represent the graph by an adjacency matrix. We can determine the shared neighbourhood of two vertices in O(n) time by "OR"-ing together their rows of the adjacency matrix.

Using adjacency lists would take O(n^2) time if the lists are unordered, or O(n) time if each list were kept in sorted order – but that takes more time to maintain the lists.

Bonus observation: if each vertex has degree $\leq$ k where k is a fixed value, then the graph should be stored as a set of adjacency lists because this permits the shared neighbourhood of any two vertices to be computed in O(1) time.

Marking:

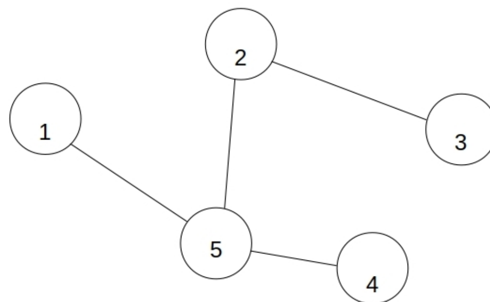
For choosing adjacency matrix and explaining why	**10/10**
For choosing adjacency matrix and not explaining	**7/10**
For choosing adjacency lists and giving an explanation	**7/10**
For choosing adjacency lists and not explaining	**5/10**
For choosing some other structure (heap, queue, etc)	**3/10**
For trying	**1/10**

Question 2 (15 marks):

As we know, a tree is a graph that contains exactly one connecting path for each pair of vertices in the graph.

Suppose T is a tree on the vertex set $\{1, 2, \ldots, n\}$ with the edges stored in a list E, where each edge is in the form of a pair of vertices. The edges are randomly ordered in the list.

For example , if T looks like this



Then the list E might look like this: Head $\rightarrow$(1,5) $\rightarrow$(3,2) $\rightarrow$(5,4) $\rightarrow$(2,5) $\rightarrow$ Nil

Describe how, given vertices x and y, you would find the path in T that connects x and y. You do not have to write a full pseudo-code algorithm (unless you want to!) but do give enough detail to make it clear how your solution works.

Be sure to identify any data structures that you use in your solution.

For full marks, your solution should run in O(n) time.

Write your solution on the next page.

Write your solution to Question 2 on this page.

Solution:

Since a tree on n vertices contains exactly n-1 edges, the length of E is n-1.

Step 1: Traverse E once and build adjacency lists for all vertices. This takes O(m) time, which for this graph is O(n) time.

Step 2: Use BFS to start at x and explore the tree. BFS uses a queue to keep track of the vertices. For each vertex added to the queue, record its "predecessor" in its path from x. This can be done with a field in each vertex object, or in a separate 1-dimensional array with one element for each vertex.

Using adjacency lists BFS takes O(m) time, which for this graph is O(n) time

Step 3: When y is added to the BFS tree, the algorithm terminates. The x-y path can be recovered in O(n) time by extracting the predecessor of y, then the predecessor of that vertex, etc.

Each of the 3 steps of the algorithm takes O(n) time, so the entire algorithm takes O(n) time.

Marking:

For a choice of d.s. that solves the problem in O(n) time, with explanation **15/15**

For a choice of d.s. that solves the problem in O(n) time, without explanation **13/15**

For a choice of d.s. that solves the problem in O(n log n) time or O(n^2) time, with explanation — 11/15

For a choice of d.s. that solves the problem in O(n log n) time or O(n^2) time, without explanation — 9/15

For a choice of d.s. that solves the problem in higher order time, with explanation — 8/15

For a choice of d.s. that solves the problem in higher order time, without explanation — 7/15

For a choice of d.s. that doesn't solve the probem — 4/15

For trying — 1/15

Question 3 (15 marks) :

Consider the following spanning tree algorithm, which is very similar to one of the algorithms we have studied. (The spanning tree it builds is not a minimum spanning tree, but that is not important for this question.) Examine the algorithm and then answer the questions stated below. **You can assume that in the graphs to which the algorithm will be applied,** $m \leq 4 * n$ (m is the number of edges, and n is the number of vertices). **You do not need to understand the purpose of the algorithm to answer this question**.

```
def test_alg(x):        # x is a vertex

        # initialization
        S = {}          # S is the set of edges that will be selected
        T = {x}         # The vertices we have connected to the tree
        R = V - {x}     # The rest of the vertices

        for each v in R:
             F[v] = 0                       # value of v
             N[v] = None                    # best neighbour of v

        for each neighbour y of x:
             F[y] = weight(x,y)             # weight(x,y) is the weight
                                            #  of the edge from x to y
             N[y] = x

        # main loop
        while |T| < n-1:
             Let v be the vertex in R with the largest F value
             S = S + {(N[v],v)}
             T = T + {v}
             R = R - {v}
             for each neighbour z of v:
                if (z is in R)  and (min(F[v],weight(v,z)) > F[z]):
                     # update F[z] and N[z]
                     F[z] = min(F[v],weight(v,z))
                     N[z] = v

        return S
```

The question continues on the next page.

a) [10 marks] What data structure would you use to store the graph G? Why?

Solution:

The best data structure for G would a set of adjacency lists, because of the "for each neighbour z of v:" loop. With an adjacency matrix this would always take O(n^2) total time. With adjacency lists, this loop takes O(m) time … which for the graphs we are given is O(n)

Marking:

For "adjacency lists" with explanation (the explanation does not need to include a comparison with an adjacency matrix)	**10**
For "adjacency lists" with invalid explanation (eg "easier to implement")	**7**
For "adjacency lists" with no explanation	**6**
For "adjacency matrix" with attempted justification	**5**
For "adjacency matrix" with no justification	**4**
For non-sensical data structure (hash table, etc.)	**3**
For trying	**1**

b) [5 marks] What data structure would you use to keep track of which vertices are in R? Why?

***Solution:* We should keep track of the vertices in R using a max-heap, using the F values computed by the algorithm to compare the vertices. The max-heap will have height $\simeq \log n$, so the n-1 "remove a vertex from R" actions will result in O($n * \log n$) operations. Since the number of edges is in O(n), the updates to the heap (based on changed F values) will also result in O($n * \log n$) operations. This choice of data structure results in a very efficient implementation of the algorithm.**

Marking:

For "max-heap" (students may just say "heap" - that's ok) with explanation	**5**
For "max-heap" (or "heap") without explanation	**4**
For a plausible alternative (such as "a 1-d array of booleans" or "a linked list of vertices") that results in higher complexity	**3**
For a non-sensical data structure, such as "stack"	**2**
For trying	**1**