This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be re-marked under any circumstances.

The test will be marked out of 50.

<table>
<thead>
<tr>
<th>Question 1</th>
<th>/15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 2</td>
<td>/10</td>
</tr>
<tr>
<td>Question 3</td>
<td>/10</td>
</tr>
<tr>
<td>Question 4</td>
<td>/15</td>
</tr>
</tbody>
</table>

**TOTAL** /50
General marking philosophy: a student who gives enough of an answer to show they understood what they were supposed to do, even if they couldn't do it (or made lots of errors while doing it) should get about 50% on that question.

Full marks should be given if a solution is sound and not missing anything important.

Many of these questions can be answered in different ways. It is not required that the students use exactly the same algorithms as I do.

A student should only get 0/10 on a question if they made no attempt to answer it at all.
Question 1 (15 marks)

Not all graphs are connected. Suppose G is a graph that may or may not be connected.

Create a modified version of either BFS or DFS that will find the connected components of G (i.e. the sets of vertices that are connected).

For example, if G looks like this

```
1  2  3
6  7
4
5
```

then your algorithm should print something like this (precise wording and vertex order are not important):

1, 7, 6 are connected

2, 4 are connected

3, 5 are connected

Please use the next page to write your algorithm. Your algorithm can contain high-level operations such as “for each neighbour of x” and “mark x visited”
a) [10 marks] Your algorithm, in pseudo-code:

Solution:

Suppose we modify BFS. One way is to run BFS starting from each vertex, testing first to see if the vertex has already been found.

```python
Modified_BFS():
    # all vertices are initially marked "unvisited"
    for each x in V:
        if x is unvisited:
            create queue Q
            Q.append(x)
            mark x "visited"
            print x
        while Q is not empty:
            v = Q.remove()
            for each y that is a neighbour of v:
                if y is unvisited:
                    Q.append(y)
                    mark y "visited"
                    print y
            print "are connected"
```
OR ... suppose we modify DFS. One way is to run standard DFS starting from each vertex of the graph, checking to see if the vertex has already been found.

Modified_DFS():
    # all vertices are initially marked "unvisited"
    for each x in V:
        if x is unvisited:
            Standard_DFS(x):
            print “are connected”

Standard_DFS(v):
    print v
    for each y that is a neighbour of v:
        if y is unvisited:
            mark y “visited”
            Standard_DFS(y)
Marking:

The key concept is that if the graph is not connected, we have to “re-start” the algorithm at another vertex. My implementations above do this by trying to start at each vertex of the graph. Some students may use more sophisticated methods such as keeping a list or set of unvisited vertices, and starting the next iteration of the search at a vertex that drawn directly from the unvisited set.

The printing or returning of the vertices in each connected component of the graph may be handled in many different ways.

Mark breakdown:
Understand the idea of starting the search again when the current search can reach no more vertices: 4 marks
Understand the fundamental structure of either BFS or DFS algorithm: 4 marks
Print or return the connected sets of vertices: 2 marks

b) [5 marks] What data structure would you use to store G, and why?

Solution: the best solution is to store G as a set of adjacency lists (or “use the adjacency list structure”) because the solution involves a loop of the form “for each y that is a neighbour of v”. The adjacency list structure guarantees optimal efficiency for this loop.

Marking: For “adjacency list” and explanation: 5 marks
     For “adjacency list” and no explanation: 3 marks
     For “adjacency matrix” and explanation: 3 marks
     For “adjacency matrix” and no explanation: 2 marks
     For other, such as “hash table” or “heap” : 1 mark
Let $G$ be a connected, edge-weighted graph.

(a) [5 marks] Suppose we wish to compute the total weight of all edges in $G$. What data structure would you choose to represent $G$ that would allow this calculation to be made most efficiently? Why?

**Solution:** An adjacency-list structure is preferable, because it allows us to compute the sum of all edge-weights in $O(m)$ time – for example, simply by iterating through all the adjacency lists and summing the weights, then dividing this total by 2 (since each edge is counted twice). Since we clearly have to look at each edge at least once, this problem is in $\Omega(m)$ so this algorithm based on adjacency lists is of optimal complexity.

An adjacency matrix representation would take $O(n^2)$ time to sum the edges.

**Marking:** For “adjacency list” and explanation: 5 marks  
For “adjacency list” and no explanation: 3 marks  
For “adjacency matrix” and explanation: 3 marks  
For “adjacency matrix” and no explanation: 2 marks  
For other, such as “hash table” or “heap”: 1 mark

They are not required to discuss more than one possible solution. They are not required to mention the $\Omega$ complexity of the problem.
(b) [5 marks] Let T be a MST of G, found by applying Prim’s Algorithm. If we add a constant value k to the weight of every edge in G, will the edges of T form a MST of the modified graph? Either explain why they will, or give an example of a graph where this does not hold.

(Hint: What would Prim’s Algorithm do when applied to the modified graph?)

**Solution:** Prim’s Algorithm would select exactly the same set of edges, since if $w(e) \leq w(f)$ for edges $e$ and $f$, then $w(e) + k \leq w(f) + k$

It is also acceptable to answer the question without reference to Prim’s Algorithm, along these lines:

For any spanning tree $X$ of $G$, let $WB(X)$ be the total weight before the modification, and let $WA(X)$ be the total weight after the modification. Since every spanning tree contains exactly $n-1$ edges and the weight of each edge is increased by $k$, we see that

$$WA(X) = WB(X) + (n-1)*k$$

Let $X$ be any spanning tree of $G$.

We know $WB(T) \leq WB(X)$

Therefore $WB(T) + (n-1)*k \leq WB(X) + (n-1)*k$ (adding the same quantity to each side)

Therefore $WA(T) \leq WA(X)$

Therefore $T$ is a MST of the modified graph.
Marking: For a correct explanation (not necessarily identical to mine):
10 marks

For an explanation that is mostly correct but has one or two small errors:
8 marks

For an explanation that has the right idea but has significant parts wrong or missing steps:
5 marks

For an explanation that almost has the right idea but goes very wrong very soon:
3 marks
Question 3 (10 marks)

Imagine a large social network in which friendships are always mutual (i.e. if A likes B then B likes A, and vice versa). What data structure would you choose to represent the network, so that it is possible to quickly determine if two arbitrarily chosen people have a mutual friend? Explain your choice.

Solution: For this problem an adjacency matrix is the best choice: we can compare two rows of the matrix looking for a column with a 1 in both rows in $O(n)$ time.

Comparing the adjacency lists of two vertices to find a common element can also be done in $O(n)$ time but it effectively requires constructing the two rows of the adjacency matrix and then applying the first solution. This is less efficient than working with the matrix directly.

Marking:

For “adjacency matrix” and explanation of why they chose it: 10 marks
For “adjacency matrix” and no explanation: 7 marks
For “adjacency list” and explanation of how to solve the problem in $O(n)$ time: 8 marks
For “adjacency list” and explanation that involves $O(n \log n)$ or $O(n^2)$: 5 marks
For other, such as “hash table” or “heap” : 1 mark
Question 4 (15 marks)

Here is a version of Dijkstra's Algorithm for computing the total weight of the least-weight path from vertex x to vertex y in a connected graph with non-negative weights on the edges. Note that since we are only interested in the total weight of the path, this algorithm does not keep track of the chosen edges.

```python
def Dijkstra(x,y):
    R = { all vertices except x }
    let d be a 1-dimensional array of length n  (where n = number of vertices)
    d[x] = 0  # d[] will contain all computed distances from x
    for each z in R:
        if z is a neighbour of x:
            d[z] = w(x,z)  # w(x,z) is the weight of edge (x,z)
        else:
            d[z] = infinity
    current = x
    while current != y:  # keep going until we reach y
        let v be a vertex in R with the smallest d[] value
        R = R - {v}
        for each neighbour z of v:
            if z is in R:
                if d[v] + w(v,z) < d[z]:
                    d[z] = d[v] + w(v,z)
                    current = v
        # now return the total weight of the xy-path
    return d[y]
```

What data structures would you use so that the complexity of this algorithm is in $O(n^2)$? (Remember, $n$ is the number of vertices.) You will need to choose a data structure for the graph, and also a data structure for R that lets you find the vertex in R with the smallest d[] value. Explain your choices.

Hint: recall that Dijkstra’s Algorithm is just a modified version of Prim’s MST Algorithm.

Use the next page to answer this question.
Solution:

To represent the graph G: The algorithm contains a loop of the form

\[
\text{for each neighbour } z \text{ of } v: \]

From our study of BFS, DFS, Prim’s Algorithm and Dijkstra’s Algorithm, we know that Adjacency Lists are most efficient for processing this loop.

(Students may also correctly point out that if the graph is dense, Adjacency Matrices and Adjacency Lists have the same efficiency for this algorithm.)

To represent R: A data structure for R that guarantees \(O(n^2)\) time for the algorithm is to represent R as an array with one column for each vertex, and two rows. The first row indicates whether the vertex is in R, and the second row gives the current best cost of connecting to the vertex. This is basically the same method of representing R that we saw when studying the \(O(n^2)\) implementation of Prim’s Algorithm.

Using a heap for R will give \(O(m \log n)\) complexity, which in the worst case is \(O(n^2 \log n)\)
Marking:

For G:  Adjacency List with explanation: 9 marks
        Adjacency List without explanation: 6 marks
        Adjacency Matrix with stated assumption that the graph is dense: 7 marks
        Adjacency Matrix without this assumption: 4 marks
        Other: 1 mark

For R:  Array (or something similar, such as two 1-dimensional arrays) to do the work in $O(n^2)$ time, with explanation: 6 marks
        $O(n^2)$ solution without explanation: 4 marks
        Heap with explanation: 3 marks
        Heap without explanation: 2 marks
        Other: 1 mark