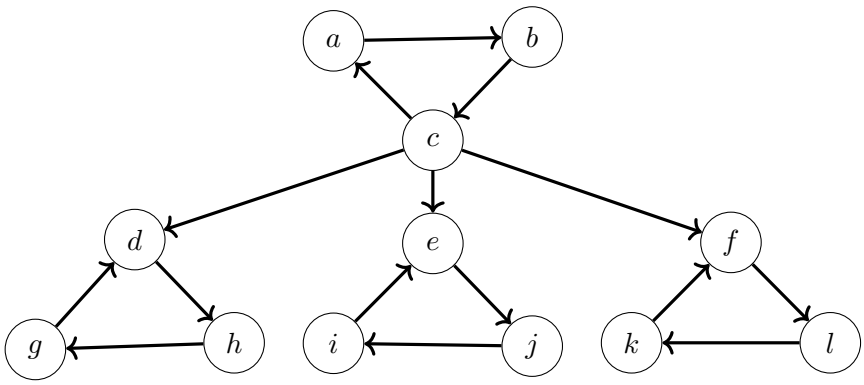
Midterm Examination
(2.5 hours)

Write your roll number in the space provided on the top of each page. *Write your solutions clearly in the space provided after each problem. If the space provided is insufficient, please write your solution on additional sheets, and* ***clearly state*** *in the main paper that where your solution appears. You may also use additional sheets for working out your solutions; attach all additional sheets at the end of the question paper.* **Attempt all problems.**

Name and Roll Number: _____

Problem	Points	Score
1	10	
2	10	
3	15	
4	15	
5	15	
Total:	65	

1. Consider the following directed graph G_0.



(a) State why G_0 is not *strongly connected.* [2]

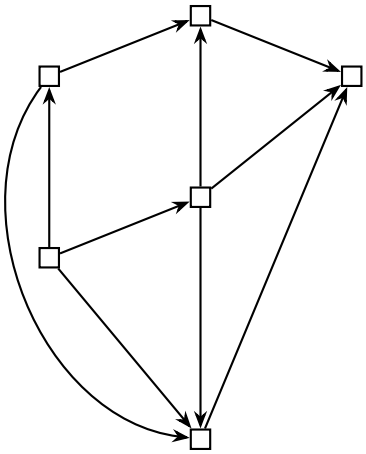
(b) Find its strongly connected components, and draw the DAG of strongly connected components. [3]

(c) Find the smallest set of edges whose addition to G_0 will make it strongly connected. [3]

(d) State why no smaller set can do the job? [2]

2. (Topological labelling) Consider the directed graph G_1 shown below; note that it has no directed cycles. We wish to assign distinct labels in the range $\{1, \ldots, 6\}$ to the vertices of the G_1 so that for every directed edge (u, v), the label of u is greater than the label of v. We call such a labelling a *topological labelling*.

   (a) Find a topological labelling for the graph G_1 given below. Write the label clearly next to each vertex. [5]



   (b) Such a topological labelling can be found using DFS. Complete the following DFS program by supplying the appropriate code in the space provided above the line marked by (???) (but do not modify any other line of the code), so that it prints the topological labelling for the input graph G. Assume that the graph is available in the memory of the program in the format we discussed in class; in particular, v.name is the name of the vertex, and v.adj is the list of vertices adjacent to v. Each vertex has an additional attribute label; in the end v.label should be the label assigned to vertex v. [5]

```
def explore(v):
    v.visited = True
    for w in v.nbrs:
        if not w.visited: explore(w)




    # (???)

def dfs(list_of_vertices):
    global clock
    clock = 0
    for v in list_of_vertices:
        v.visited = False
        v.label = 0
    for v in list_of_vertices:
        if not v.visted:
            explore(v)
    for v in list_of_vertices:
        print(v.name, v.label)
```

3. Suppose M is an m-bit number. Consider the sequence $x_0, x_1, \ldots$, defined by $x_0 = 1$ and $x_{k+1} = Ax_k + B \bmod M$, where $A, B \in \{0, 1, \ldots, M-1\}$.

   (a) Suppose that $\gcd(A-1, M) = 1$. By appealing to some version of Euclid's algorithm discussed in class, show how a number $C \in \{1, 2, \ldots, M-1\}$ can be found efficiently (in time polynomial in m) such that $C(A-1) = 1 \pmod{M}$. [5]

   (b) Next, suppose an n-bit number N is given. Describe an algorithm to determine x_N in time polynomial in m and n. You need not write any code; just write the steps in text, and state which algorithms we studied in class you will use for each step. (You may start by showing that there is a D such that $x_{k+1} - D = A(x_k - D)$; state how such a D can be computed using the number C computed above. Or, you may use some other more direct method to determine x_N.) [10]

4. For a sequence $X = (x_1, x_2, \ldots, x_n)$ of n distinct numbers and another sequence $Y = (y_1, y_2, \ldots, y_k)$ of k distinct numbers, let

$$n_i = |\{j : x_j \leq y_i\}|,$$

for $i = 1, 2, \ldots, k$. Assume $k \leq n$.

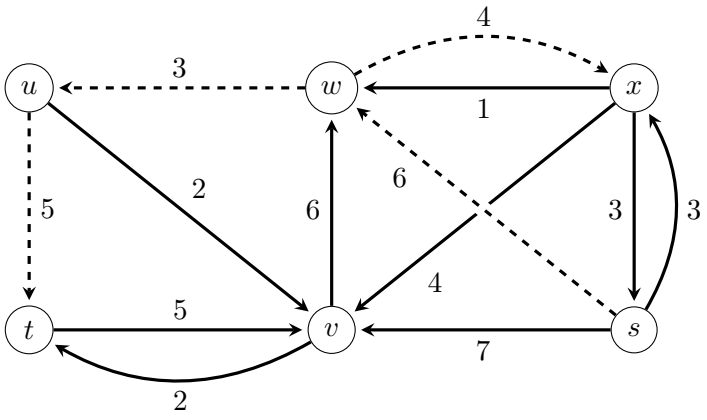
(a) Describe an efficient algorithm that given X and Y as input, outputs $n_1, n_2, \ldots, n_k$. Assume that two numbers can be compared in unit time, and numbers of magnitude at most n can be added in constant time. You do not have to write the code; just mention the steps clearly, and state how long your algorithm takes in terms of k and n. For full credit, you must present an algorithm that runs in time $O(n \log k)$. 10

(b) Argue that for every $k \leq n$, any comparison-based algorithm that solves the above problem must make $\Omega(n \log k)$ comparisons; alternatively, argue that any decision tree where comparisons are performed at the the internal nodes and answers are declared at the leaves must have height $\Omega(n \log k)$. [Hint: As we did while discussing the element distinctness problem in class, consider the partially ordered set (on $X \cup Y$) of the outcomes that led to each leaf; what must this look like?] 5

5. Consider a directed graph $G = (V, E)$ with a source vertex s, from which every other vertex is reachable. Each edge e in the graph has a positive capacity $\mathsf{cap}(e)$. Consider a path p from s to a vertex $v \in V$. The *bottleneck capacity* of p is the minimum weight of an edge on p: 15

$$\mathsf{bcap}(p) = \min_{e \in p} \mathsf{cap}(e);$$

note that min taken over an empty set is ∞, denoted in the code below by INF. As a concrete example, consider the following graph.



Here the path $s \longrightarrow w \longrightarrow x$ has bottleneck capacity 4, while the path $s \longrightarrow w \rightarrow u \rightarrow t$ has bottleneck capacity 3. (The edges in these paths are made dashed in the above picture to easily identify them, but the dashed lines have no other significance.) Complete the code on the next page so that in the end, for each vertex v, an (s, v)-path of maximum bottleneck capacity, and the capacity of that path are printed. The graph is available in the memory in the form of adjacency lists (as discussed in class): for a vertex u, the list u.out_nbrs contains pairs (v, cap), where cap is the capacity of the edge (u, v). You can use the functions min or max in your code.

```
def max_bottleneck_path(s):
    for v in vertex_list: v.bcap = 0     # initialize all bcap values to 0
    (s.bcap, s.prev) = (INF, None)
    H = makeheap(vertex_list)    # maxheap wrt the bcap values of the vertices
    while H:                     # exit while loop if heap H is empty
        u = deletemax(H)
        for (v, cap) in u.out_nbrs:

            if v.bcap < ______(________ , ________)  :

                v.bcap = ______________________

                v.prev = ______________________

                bubble_up(H,v)

max_bottleneck_path(source)   # initially source is set to the vertex s
for v in vertex_list:
    path = []

    # The while loop below uses the information generated above
    # to trace the path backwards from the vetex to the source.
    w = v
    while w:                          # exit loop if w == None
        path = path.append(w.name)

        w = _______________

    print(v.name, ':', 'path=', [v for v in reversed(path)], 'capacity=', v.bcap )
```