This paper is being provided to the CSIV (Spring 2023) students as a sample. Some questions in this sample concern topics that we have not yet covered: greedy algorithms, shortest paths with negative weights, etc.; there are topics that this question paper does not cover: computer arithmetic, divide-and-conquer, randomized algorithms. See the course website for the syllabus and other details for the midterm exam (23 Feb 2023, 10:00 am to 12:30 pm).

https://www.tifr.res.in/ jaikumar/Courses/ISIBangalore/2023-CSIV/

Midterm Examination

Write your solutions clearly in the space provided after each problem. You may use additional sheets for working out your solutions; attach such sheets at the end of the question paper. **Attempt all problems.**

Name and Roll Number: ____________________

Problem	Points	Score
1	2	
2	6	
3	2	
4	6	
5	5	
6	12	
7	17	
8	10	
Total:	60	

1. Assume that we merge the following lists starting from the left, and proceeding to the right. $\boxed{2}$

$$\begin{aligned} L_1 &= [1, 9, 17, 22, 30] \\ L_2 &= [4, 10, 18] \end{aligned}$$

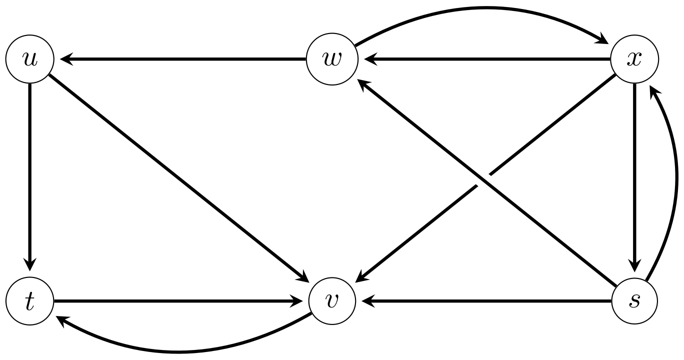
Merge performs __________________________ comparisons.

2. Perform a depth-first search on the directed graph given below, starting from vertex s, and exploring the list of neighbours of vertices in the alphabetical order: s, t, u, v, w, x. Draw the DFS tree, and fill in pre(z) and post(z) for all vertices z. $\boxed{6}$

$$\begin{tikzcd}
u \arrow[d] \arrow[dr] & w \arrow[l] \arrow[r, bend left] & x \arrow[l] \arrow[d] \arrow[dl] \\
t \arrow[r] & v \arrow[u] \arrow[l, bend left] & s \arrow[l] \arrow[ul] \arrow[u, bend right]
\end{tikzcd}$$

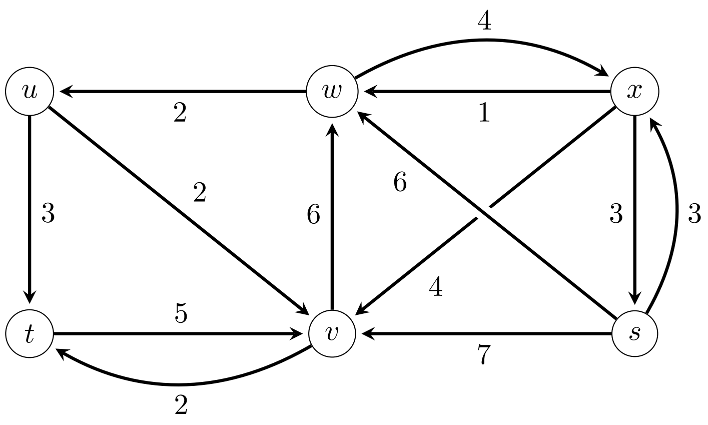
Vertex	s	t	u	v	w	x
pre	1					
post						

3. List the strongly connected components of the following graph. $\boxed{2}$



The strongly connected components are:

4. Suppose Dijkstra's algorithm is run on the following graph starting with vertex s as source. $\boxed{6}$



All six vertices are initially inserted into a min-heap. Initially, $\mathsf{dist}(s) = 0$ and $\mathsf{parent}(s) = s$; for other vertices, dist is set to ∞, and parent to None. In the first iteration DeleteMin returns s, in the second iteration it returns x, and in the third it returns w. In the following table, enter the values that dist and parent would take just after w has been removed from the heap and the values for its neighbours have been updated.

Vertex	s	t	u	v	w	x
dist	0					
parent	s					

5. How would you assign binary prefix-free codewords to the letters a, b, c ,d ,e, f, g, h if the frequencies are as follows? $\boxed{5}$

```
a:1   b:1  c:2  d:3  e:5  f:8  g:13  h:21
```

You may represent the codewords using a tree.

6. There are n activities numbered $1, 2, \ldots, n$. Each activity has a *start time* s_i and *finish time* f_i, where $s_i < f_i$. These activities have to be scheduled in a single room. Two activities i and j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. We want to schedule the maximum number of compatible activities possible. Here are three greedy strategies that repeatedly select an activity based on a rule and discard activities that conflict with the selected activity. For each of these strategies either show a counter-example (state what the strategy yields and what the optimum should be) or argue that it produces the optimum solution (that is, always schedules the maximum possible number of activities).

   (a) In the first strategy, we pick the shortest activity first. $\boxed{4}$

   (b) In the second strategy, we pick the activity with the earliest start time first. $\boxed{4}$

   (c) In the third strategy, we pick the activity with the earliest finish time first. $\boxed{4}$

   Solution: To prove that this method is optimal, we first show that there is an optimal solution that includes the first activity included by our greedy algorithm. Let this job be (s_j, f_j). Now, consider any optimal solution OPT. At most one job in OPT can be in conflict with (s_j, f_j). Remove this job and add (s_j, f_j). Thus, there is an optional solution that includes (s_j, f_j). Thus, the activity (s_j, f_j), together with the optimal solution for the collection of activities obtained by excluding (s_j, f_j) and all activities conflicting with it, will be an optimal solution. The correctness of our greedy algorithm thus follows by induction on the number of activities in the collection.

7. Let $A[i] : i = 0, 1, 2, \ldots, n-1$ be an array of n distinct integers. We wish to sort A in ascending order. We are given that each element in the array is at a position that is at most k away from its position in the sorted array, that is, we are given that $A[i]$ will move to a position in $\{i-k, i-k+1, \ldots, i, \ldots, i+k-1, i+k\}$ after the array is sorted in ascending order. Here are two proposals for sorting A.

Method I: Use insertion sort without any modification.

Method II: Use heapsort, but with the following modification. Insert the first k elements of the input array into a heap; DeleteMin; insert the next element from the input array into the heap; DeleteMin; and so on.

(a) State precisely why the number of comparisons made in Method I will always be $O(nk)$. (Note elements are moved forward and backward in the array during the course of insertion sort.) $\boxed{7}$

Solution: When $A[i]$ is inserted, if t comparisons are made then it is moved forward at least $t-1$ steps. Let $A[j]$ be the element that is moved forward the most, say, m positions. We claim that $A[j]$ does not move back at all. In fact, all later elements, even if they are moved forward m positions, will not cross $A[j]$. So, $A[j]$ once it has moved forward never moves back. Since $A[j]$ does not move more than k steps away from where it was originally, we have $m \leq k$. Thus, no element is moved forward more than k position, so no element participates in more than $k+1$ comparisons when it is inserted.

(b) State precisely why Method II always sorts the elements of the array correctly? $\boxed{6}$

Solution: *Nishad Mandlik rightly pointed out that there was an error in the description of Method II. One should keep a heap of size $k+1$ not k.* Note that the i-th element in the sorted order must originally reside in the array locations $A[0]$, ..., $A[i+k]$. Now, suppose that the first $i-1$ elements are produced in sorted order. Now, after the first $i-1$ elements have been output and the element $A[i+k]$ has been inserted, all elements out of $A[0]$, ..., $A[i+k]$ that have not already been printed are in the heap. $A[i]$ must be the minimum of these. So, the i-th element is also produced in sorted order. So, by induction, the elements are produced in sorted order.

(c) How many comparisons will Method II make in the worst case for such inputs? (Account for the DeleteMins and inserts, use big-Oh.) $\boxed{2}$

$O(n \log k)$

(d) Is there any reason to prefer Method I to Method II? $\boxed{2}$

Solution: It is stable and in place.

8. Suppose we are given a graph G, where vertices represent individuals, and edges are placed between siblings, between children and their parents, and between individuals who are married. We are also given an array $W[v] : v \in V(G)$, which contains the wealth of individual v in rupees.

(a) Design an efficient algorithm to determine everybody's richest relative, that is to generate an array $R[v] : v \in V(G)$, so that $R[v]$ is v's richest relative. You need not present the detailed code; state precisely the steps your algorithms will take. $\boxed{7}$

Solution: Step 1: Run DFS to determine for each vertex v its component number $\mathsf{comp}[v]$.
Step 2: If the number of connected components is k, create an array of length k, $\mathsf{MaxInComp}$ where $\mathsf{MaxInComp}[i]$ holds the vertex of maximum wealth in component i. Intialize this to None.
Step 3: For each vertex $v \in V$, set $\mathsf{MaxInComp}[\mathsf{comp}[v]]$ to v if v's wealth is more than the wealth of the vertex stored there.
Step 4: Set $R[v]$ to $\mathsf{MaxInComp}[\mathsf{comp}[v]]$ for all v in V.

(b) What is the running time of your algorithm? $\boxed{3}$

Solution: Step 1: $O(m + n)$
Step 2: $O(n)$
Step 3: $O(n)$
Step 4: $O(n)$.
Total: $O(m+n)$, where m is the number of edges and n is the number of vertices.