1. You are given a set of \( n \) tuples \((x_i, p_i)\) \(1 \leq i \leq n\) where \(x_i\)s are key-values and \(p_i\)s are priorities. We want to define a binary search tree on \(x_i\)s and simultaneously a min-heap on the \(p_i\)s. For example, if the given elements are \((4, 6), (1.5, 0.8), (5, 5), (8, 2.5), (6, 5, 8.5)\) then the following binary tree satisfies the above two properties. The main observation for this problem is that because of the min-heap property, the node having the lowest value of \(p_i\), say \(p_i^*\) will be the root node. Subsequently, all the nodes that have \(x_i < x_i^*\) will be in the left subtree and all nodes with \(x_i > x_i^*\) will be in the right subtree. Then the left and right subtrees can be built recursively. So there is a unique such structure unless the \(p_i\) values are not distinct.

Write True/False against the following statements \((2 \times 4)\)

(a) It is not always possible to build such a Binary tree that satisfies both properties. 
\textbf{false}  
(b) It is always possible to build, but there may be multiple such trees. 
\textbf{false}  
(c) It is always possible and there is a unique tree that satisfies both properties. 
\textbf{true}  
(d) It is possible only when the the points are maximal on plane. 
\textbf{false}  

2. Given a sequence of real numbers \(x_1, x_2, \ldots, x_n\) (not necessarily positive), find a subsequence \(x_i, x_{i+1}, \ldots, x_j\) of consecutive elements such that the sum of the numbers in the subsequence is maximum over all possible contiguous sub-sequences. These subsequences may be of any length. For example in the sequence \(2, -3, 4, 6, -3, 5, -4, 2\), the subsequence that attains the maximum sum is \(4, 6, -3, 5\) having value \(4+6-3+5 = 12\).

Answer the following questions about the problem \((1 \times 4)\)

(a) If all numbers are non-negative, then the maximum valued subsequence is \textbf{sum of all elements}.  
(b) If all numbers are negative then the maximum weighted subsequence is \(\max_i \{x_i\}\).  
(c) The maximum valued subsequence cannot have consecutive negative numbers \textbf{false}.  
(d) The maximum valued subsequence cannot become negative at any intermediate element, i.e., if \(x_i, x_{i+1}, \ldots, x_j\) is maximum then for all \(i \leq k \leq j, x_i + \ldots x_j\) cannot be negative. \textbf{false}
Let $L(i)$ denote value of the maximum subsequence in $x_1x_2\ldots x_i$. Let $P(i)$ denote the value of the maximum subsequence that contains $x_i$, then complete the following recurrence relations. Initially $L(1) = P(1) = x_1$ and we successively compute $L(i), P(i)$ for $1 \leq i \leq n$.

\begin{align}
L(i + 1) &= \max\{L(i), P(i) + x_{i+1}, x_{i+1}\}, \quad i < n \\
P(i + 1) &= \max\{x_{i+1}, P(i) + x_{i+1}\} \\
\end{align}

The recurrence maintains the highest valued subsequence in $x_1\ldots x_i$ as $L(i)$ and compares it with the current subsequence involving the latest element that is scanned. It scans the elements from left to right and inductively maintains these values. The only subsequence that can be better than $L(i)$ must contain the latest element (from induction). Unless the current subsequence is negative, the current element $x_i$, is appended to the sequence. Otherwise $x_i$ on its own defines the highest valued current sequence. (e) The running time of the algorithm based on the aforementioned recurrence relation is $O(n)$.

3. Consider the following strategy for generating a random permutation of $n$ distinct objects.

For object $i = 1$ to $n$ do

Repeat

Generate a random integer $r$ from $[1..n]$.

If the $r$-th slot is empty assign object $i$ to it.

until object $i$ is not assigned

done

(i) What is the expected number of times the inner loop is executed for object $i$? Your answer must define the random variable precisely. (The inner loop is repeated till an empty slot is picked for the object under consideration). (3) Let $X_i$ be a random variable that denotes the number of times the inner loop runs for the $i$-th object. Since $i - 1$ slots are already occupied, the probability of finding an empty slot when $r$ is chosen uniformly at random (and independently) is $p_i = \frac{n-(i-1)}{n}$. So $X_i$ follows a geometric distribution, i.e., $Pr[X_i = k] = 1 - p_i^{k-1} \cdot p_i$ and $E[X_i] = \frac{1}{p_i} = \frac{n}{n-i+1}$.

(ii) What is the expected total time complexity of the algorithm? Your answer must define the random variable precisely. (3) Let $X$ be a r.v. that denotes the running time of the algorithm. Clearly

$$X = \sum_i X_i = n \cdot \sum_{i=1}^{n} \frac{1}{n-i+1} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = nH_n = O(n \log n)$$

where $H_n$ represents the harmonic sum.

(iii) What is the total time complexity if $r \in [1,2n]$ instead of $[1,n]$? (2) In this case, $p_i \geq 1/2$, so $E[X_i] \leq 2$ implying $E[X] \leq 2n$.

4. A vertex cover of a graph $G = (V,E)$ is a subset $W \subseteq V$ such that for all $(x,y) \in E$ at least one of the endpoints $x, y \in W$.

(i) For a given tree $T$ design an efficient algorithm to find the minimum cardinality vertex cover of $T$. The tree is not necessarily balanced, nor is it binary. (10)

(ii) Will your algorithm work if every vertex has a non-negative real number weight and we want the least weighted vertex cover? Justify or provide necessary modifications. (5)

The vertex cover of a tree can be constructed as follows - If the root is included then all the edges incident on the root are covered, therefore we need to inductively find the minimum cardinality vertex cover of each of the subtrees.

If the root is not included, then we have to choose all its children (to cover the edges incident on
the root) and then choose the minimum cardinality vertex cover of all the subtrees rooted at the grandchildren of the root.

For a singleton vertex, we do not need to include any vertex and that is the terminating condition. The above can be written out as a DP recurrence. To transform into an algorithm, we need to store the information with the subtrees. One natural data structure is to store the parent information for every vertex. The algorithm will start from the leaf nodes and move to the parent node after the optimal vertex cover has been computed for all the siblings (other children of the parent node. One initial DFS/BFS can be used to compute the level of every node and we compute from the highest level to the lowest (root).

For each node \( v \in T \), the computation of the recurrence takes \( C(v) \) steps where \( C(v) \) denotes the number of children of \( v \). So the total time is \( \sum_{v \in T} C(v) = |E| = n - 1 = O(n) \).

An alternate solution is based on discarding degree 1 nodes (leaf nodes) from the solution. In this case, you select the parent of the leaf nodes, discard the incident edges and repeat it for the remaining tree. This gives a vertex cover but the proof of minimality has to be argued carefully. It is greedy-like but the proof will be based on induction of contradiction. That is, if there is an optimal solution that uses leaf nodes, then one should be able to transform it to an equally good solution that does not use leaf nodes by swapping them with their parents. Without a proper correctness proof, these solutions have been penalised. Most of these solutions have been presented in a code like manner using undefined variables. Also this solution doesn’t work for weighted cover.

(ii) When the vertices have weights, \( W(v) \), then the recurrence for the optimum VC for a subtree rooted at \( v \) can be modified as

\[
A(v) = \max \{ W(v) + \sum_{u \in C(v)} A(u), \sum_{u \in C(v)} W(u) + \sum_{u \in G(v)} A(u) \}
\]

where \( G(v) \) denotes the grand-children of \( v \) and the two terms correspond to choosing the root and not-choosing respectively. Now the rest of the computation is identical to the unweighted cover and takes linear time.

5. We are given \( n \) jobs \( J_1, J_2 \ldots J_n \) that have precedence constraints \( J_i \prec J_k \) to denote that \( J_i \) must be completed before \( J_k \). Each job takes unit time and can be scheduled only after the predecessor (if any) has been completed. We have multiple processors available that can execute the jobs in parallel. We would like to complete all the jobs in the minimum number of time steps using as many processors as needed. However all the jobs that are being executed simultaneously cannot have any dependence among each other. For example if \( J_2 \prec J_1 \) and \( J_2 \prec J_3 \), then we can schedule \( J_2 \) in the first round followed by \( J_1, J_3 \) in the second round using 2 processors. Clearly we cannot do it in less than 2 rounds.

(ii) Given a set of precedence constraints, design an efficient algorithm that produces a schedule to minimize the number of time steps to complete all jobs. A schedule should specify what jobs will be running in each round. For the previous example, the schedule would be \( T = 1: J_2, \ T = 2: J_1, J_3 \). Justify why your algorithm achieves the minimum number of rounds and why is it efficient in terms of running time. (10)

(ii) Discuss how will you handle a situation where the jobs may have different execution times. (5)

The first check is to verify that the dependency graph defined the directed edges \( (i, k) \) corresponding to the relation \( J_i \prec J_k \) is a DAG. Otherwise there is no feasible schedule. The next observation is that the minimum number of parallel rounds is the length of the longest path in the DAG, say \( \ell \).

We will now design a schedule such that all jobs can be completed within \( \ell \) rounds. In round 1, we schedule all those jobs whose vertices have indegree 0. We delete these vertices and in the resulting Digraph we again schedule the vertices having indegree 0. This is repeated until the graph is empty.

3
Correctness: Let $\text{depth}(u)$ denote the length of the longest path (from any source vertex) to $u$. One can introduce an artificial unique source vertex to simplify arguments. Using induction on $\text{depth}(u)$ one can easily argue that the above algorithm schedules $J_u$ in round $\text{depth}(u)$. Moreover, all the jobs that are scheduled in the same round have no dependencies between them (otherwise depth will differ).

*Note that the depth is not the BFS depth, which is the shortest path from the source but it is the LONGEST path.*

Analysis : To implement the algorithm, one can keep track of the incoming degrees of vertices - all vertices having incoming degree 0 are scheduled in a given round. Initially, a simple traversal of the DAG will yield the incoming degrees of every vertex (or it can be read out from the adjacency lists). You can keep separate lists corresponding to each degree and move vertices to appropriate lists when their degrees are modified. There can be at most $n$ lists in the beginning. Each of the list operation (insert and delete) can be charged to the edge that is removed. So, overall the running time is $O(|E|)$.

Many solutions have only focussed on the procedure of identifying the degree 0 vertices that leads to an obviously feasible schedule. However the minimality has to be argued correctly using an inductive argument based on a PRECISE induction hypothesis. Without this, the minimality argument falls through. For example ”For all iterations $i$, the jobs scheduled in the $i$-th iteration cannot be scheduled earlier” and this can be proved by induction. Most solutions could only identify the base case correctly but didn’t complete the proof. Alternately induction can be done on the length of the longest path.

(ii) When jobs have different running times, the above approach still works with suitable modifications. Instead of $\text{depth}(u)$, we have compute the longest weighted path to $u$ where weights equal to the execution times of a job $J_u$ are placed on the outgoing edges of vertex $u$. We can use dynamic prog algorithm to compute the lengths of the longest paths. (In general the longest path problem is hard, but for DAGs it is the same as shortest paths). The schedule is now calculated as the earliest time when a job can be scheduled, so the notion of rounds is no longer relevant. However, many jobs could be running simultaneously.
6. You are given a ground set \( S = \{x_1, x_2, \ldots, x_n\} \) and a family of subsets \( T_1, T_2, \ldots, T_m \) where \( T_i \subset S \). We want to find out if there are \( k \) subsets \( T_{i_1}, T_{i_2}, \ldots, T_{i_k} \) where \( i_1, i_2, \ldots, i_k \in \{1, 2, \ldots, m\} \) such that \( T_{i_1} \cup T_{i_2} \cup \ldots \cup T_{i_k} = S \), i.e., their union contains all the elements of \( S \).

For example, if \( S = \{1, 2, 3, 4\} \), \( T_1 = \{1, 3\} \), \( T_2 = \{2, 3\} \), \( T_3 = \{4\} \), \( T_4 = \{1, 2\} \) \( k = 3 \), then answer is Yes (setcover : \( \{(1, 3), (2, 3), (4)\} \) but for \( k = 2 \), the answer is No.

(i) Show that the above problem, known as the set cover problem is NP Complete. (10)

(a) The set cover problem is in NP since, one can verify efficiently if the union of elements of given subsets equals the ground set.

We will reduce the vertex cover problem (NPC proved in class) to set-cover that will establish the NP-hardness of set-cover. For this, consider a graph \( G = (V, E) \) and an integer \( k \), that is an arbitrary instance of a VC problem. We define the ground set as \( E \) (the set of edges of the graph). The family of subsets is defined as \( S_u = \{(u, w) \mid u, w \in E\} \), i.e. all those edges that are incident on the vertex \( u \). Therefore, the total number of subsets in this system is \( |V| \). It can be easily argued that if \( W \subset V \) is a vertex cover of \( G \), then \( \cup_{x \in W} S_x = E \), i.e., this family of subsets is a set-cover of the ground set \( E \). Conversely, if for some \( W \subset V \), \( \cup_{x \in W} S_x = E \), then \( W \) is a vertex cover of \( G \).

Most attempts could only do the NP part for which they were given 2 marks. Many faltered by trying to do the reduction in the opposite direction. That is, the tried to create a 3SAT formula (or vertex cover instance) from a given instance of set-cover. You were supposed to create a setcover instance from an arbitrary 3CNF or VC instance.

(ii) Consider subsets \( S_{(i,j)} \) containing elements \( \{i, i+1, \ldots, j\} \) where \( i < j \) and \( i, j \in \{1, 2, \ldots, n\} \). Given \( S_{(i_1,j_1)}, S_{(i_2,j_2)}, \ldots, S_{(i_m,j_m)} \) design an efficient algorithm to identify the minimum number of subsets whose union contain all the elements in \( \{1, 2, \ldots, n\} \). (5)

We use a greedy approach by picking the interval corresponding to the largest number of uncovered elements starting from the first uncovered element. These can be thought of as intervals on the real line such that the union of the intervals cover \([1, n]\). To argue that this results in the smallest cover, we use a proof by contradiction. Consider an optimal cover \( OPT \) such that the intervals are sorted by their starting points. Our algorithm produces another set \( A \) such that \( A \neq OPT \). Look at the first interval \( I \in OPT - A \). Since algorithm \( A \) chose the longest interval of uncovered elements, say \( I' \) we can substitute \( I \) by \( I' \) without increasing the size of the cover. This argument can be repeated showing that \( A \) has cardinality equal to \( OPT \) and therefore optimal.

A proof can also be based on the observation that

The first \( i \) intervals of \( A \) covers at least as many elements of the first \( i \) intervals of \( OPT \) for all \( i \).

This can be easily proved by induction.