COL 351 Aug 9,10,14

TUTORIAL SHEET 2

1. You are given a set of intervals on a line segment. You wish to color these segments such that no two overlapping segments get the same color. Devise a greedy algorithm for coloring the intervals which uses as few colors as possible.

   Solution Sketch: For point p, let C_p denote the number of intervals containing p. Clearly, one needs at least C_p colors. Therefore, C, defined as, $\max_p C_p$ is a lower bound on this minimum number of colors. Now, we show how to color the intervals using C colors using a greedy algorithm.

   Order the intervals in increasing order of their starting points – let this ordering be $I_1, I_2, \ldots, I_n$. Consider the intervals in this ordering. Suppose we have already colored $I_1, \ldots, I_k$. Now, we look at I_{k+1}. The intervals among $\{I_1, \ldots, I_k\}$ which overlap with I_{k+1} must contain the starting point of I_{k+1}, and so, there can be at most $C-1$ such intervals. Thus, if we have C colors, then we can color I_{k+1} with one of the unused colors.

2. Consider the problem of making change for n rupees using the fewest number of coins. Suppose that the available coins are in denominations that are powers of c, i.e., the denominations are $c^0, c^1, \ldots, c^k$ for some integer $c > 1$ and $k \geq 1$. Show that the following greedy algorithm always yields an optimal solution – pick as many coins of denomination c^k as possible, then pick as many coins of denomination c^{k-1} and so on.

   Solution Sketch: The algorithm is greedy: if M is the remaining money, then find the largest i such that $M \geq c^i$, and take a coin of value c^i, and iterate with $M - c^i$ remaining money. For the proof, we can proceed by induction. Fix an optimal solution O. Suppose, $M \geq c^i$, but $M < c^{i+1}$. If O also picks a coin of value c^i, proceed by induction. Suppose all the coins picked by O are of value c^{i-1} or less. Suppose O picks α_j coins of value c^j, $j \leq i-1$. First notice that $\alpha_j < c$ (why ?). But then $\sum_{j=1}^{i-1} \alpha_j c^j < c^i$, which is a contradiction.

3. **(KT-Chapter 4)** Suppose you are given an undirected graph G, with edge weights that you may assume are all distinct. G has n vertices and m edges. A particular edge e of G is specified. Give an algorithm with running time $O(m+n)$ to decide whether e is contained in a minimum-weight spanning tree of G.

   Solution Sketch: We need to check if the greedy algorithm will pick e or not. This will happen only if at the time it considers e, the end-points of e are in different components. Therefore, consider the graph formed by taking all edges of weight less than the weight of e. If the end-points of e, lie in different components of this graph, then e has to be in the minimum spanning tree, otherwise it cannot be in the minimum spanning tree. Now, note that we can construct such a graph and its connected components can be found in linear time (depth first search or breadth first search).

4. **(KT-Chapter 4)** Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a schedule for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$ whichever order you choose). We assume that all the values b_i and t_i are positive integers. Now, because you're just one user, the link does not want you taking up too much bandwidth – so it imposes the following constraint, using a fixed parameter r:

   (*) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt.

   Note that this constraint is only imposed for time intervals that start at 0, not for time intervals that start at any other value. We say that a schedule is valid if it satisfies the constraint (*) imposed by the link. The problem is: Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether there exists a valid schedule.

   Example. Suppose we have $n = 3$ streams, with $(b_1, t_1) = (2000, 1), (b_2, t_2) = (6000, 2), (b_3, t_3) = (2000, 1)$, and suppose the link's parameter is $r = 5000$. Then the schedule that runs the streams in the order 1, 2, 3, is valid, since the constraint (*) is satisfied:
   $t = 1$: the whole first stream has been sent, and $2000 < 5000 \cdot 1$
   $t = 2$: half the second stream has also been sent, and $2000 + 3000 < 5000 \cdot 2$.
   Similar calculations hold for $t = 3$ and $t = 4$.

   (a) Consider the following claim:
   Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.
   Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

   Solution Sketch: It is clearly false. For example, if $r = 1$, and we have two streams $(2, 1)$ and $(1, 1000)$, then the first stream does not satisfy this condition. But we can build a valid schedule by ordering the second stream before the first stream.

   (b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

   Solution Sketch: Order the streams in order of b_i/t_i, and check if this schedule has the desired property. Prove that there is a valid schedule which orders the stream in this order (if not, there will be two consecutive streams in this schedule which are out of order. Then perform an exchange and argue as we did in class.

- **(KT-Chapter 4)** Timing circuits are a crucial component of VLSI chips; here's a simple model of such a timing circuit. Consider a complete binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length l_e, which is a positive number. The distance from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf. The root generates a clock signal which is propagated along the edges to the leaves. We?ll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf. Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem: we want the leaves to be completely synchronized, and all receive the signal at the same time. To make this happen, we will have to increase the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have zero skew. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible. Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew, and the total edge length is as small as possible.

  Solution Sketch: Let the subtrees below the root r be L and R. If the height of L (where we think of height as maximum length over all root-leaf paths) is Δ more than that R, $\Delta \geq 0$, then we increase the length of the edge (r, R) by Δ. Now, we repeat the same process over L and R independently. Again, argue by induction that this greedy algorithm is optimal – if it does not increase the length of (r, R) edge by Δ, then prove that we can improve the solution.

5. **(KT-Chapter 4)** Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, \ldots, d_n$. (That is, if $V = \{v_1, \ldots, v_n\}$, then the degree of v_i should be exactly d_i.) G should not contain multiple edges between the same pair of nodes, or "loop" edges with both endpoints equal to the same node.

   Solution Sketch: If all the d_i are 0, then we know that there is such a graph: the graph has n vertices and no edges. So assume this is not the case. Sort the d_i in decreasing order: $d_1 \geq d_2 \geq \ldots \geq d_n$. Now argue that there is a graph with degree sequence $(d_1, \ldots, d_n)$ if and only if there is a graph (on $n-1$ vertices) with degree sequence $(d_2 - 1, d_3 - 1, \ldots, d_k - 1, d_{k+1}, \ldots, d_n)$, where $k = d_1$. In other words, we are saying that if a graph with sequence $(d_1, \ldots, d_n)$ exists, then we can assume that the highest degree vertex (of degree d_1) has edges to the next d_1 highest degree vertices. Let us see why. One direction of the proof is easy: if there is a graph G with degree sequence $(d_2 - 1, d_3 - 1, \ldots, d_k - 1, d_{k+1}, \ldots, d_n)$, then there is a graph with degree sequence $(d_1, \ldots, d_n)$: add a new vertex to G which has edges to the vertices with degrees $d_2 - 1, d_3 - 1, \ldots, d_k - 1$. Let us now prove the reverse (and the more non-trivial direction of the proof). Suppose there is a graph G with degree sequence $(d_1, \ldots, d_n)$. Let v_i be the vertex with degree d_i. If v_1 has edges to $v_2, \ldots, v_k$ in G, then we are done – just remove v_1 and you have the graph with the desired degree sequence. So assume there is an index i, $2 \leq i \leq k$ such that (v_1, v_i) is not an edge. Since degree of v_1 is k $(=d_1)$, there must be an index $j > k$ such that (v_1, v_j) is an edge. Since

degree of v_i is at least that of v_j, there must be a vertex v_k such that (v_i, v_k) is an edge, but (v_j, v_k) is not an edge. Now, in G, we remove the edges (v_1, v_j) and (v_i, v_k), and add the edges (v_1, v_i) and (v_j, v_k). Note that this does not change the degree of any vertex, but now, we have increased the number of edges from v_1 to the vertices in the set $\{v_2, \ldots, v_k\}$. Repeat this process till v_1 has edges to $\{v_2, \ldots, v_k\}$.