Major Exam (COL 351)

Read the instructions carefully:

- You need to justify correctness and running time of each algorithm.
- If using dynamic programming, you must explain the meaning of table entries, and explain the order of computing them.
- For NP-completeness reduction, you can use the fact that 3-SAT, Independent Set, Vertex Cover, Partition, Subset Sum, Clique and Hamiltonian Cycle (in directed and in undirected graphs) are NP-complete.
- No argument should use examples – they will be ignored.
- You can assume any result proved during the lectures (but cannot assume any other result which is in the book or in the tutorials).

1. You are organizing a sports event, where each contestant has to swim 20 rounds in a pool, and then run 3 kilometers. However, the pool can be used by only one person at a time. In other words, the first contestant swims 20 rounds, gets out and then starts running. As soon as this first person is out of the pool, a second contestant begins swimming the 20 rounds; as soon as he/she is out and starts running, a third contestant begins swimming... and so on.) (**6 marks**)

   You are given a list of n contestants, and for each contestant you are given the time it will take him/her to complete swimming 20 rounds of the pool, and the time it will take him/her to run 3 kilometers. Your job is to decide on a schedule for the event, i.e., an order in which to sequence the starts of the contestants. The completion time of a schedule is the earliest time at which all contestants will be finished with swimming and running. Give an efficient algorithm that produces a schedule whose completion time is as small as possible.

   Example: Suppose there are two contestants C_1, C_2 with swimming and running times being $(10, 5)$ for C_1 and $(2, 8)$ for C_2. If we schedule them as C_1, C_2, then C_1 will finish swimming and running by time $10 + 5 = 15$. C_2 can start swimming at time 10, and so, will finish by time $10 + 2 + 8 = 20$. Note that both contestants will finish by time 20, and so the completion time is 20. If we order them as C_2, C_1, then C_2 will finish by time 10 and C_1 by time 17. Therefore, the completion time is 17. Thus, the best ordering is C_2, C_1.

   Solution: The algorithm schedules the contestants in decreasing order of their running time (**2 marks**). Let this order be $C_1, \ldots, C_n$. Now we prove correctness. Consider an optimal schedule $O_1, \ldots, O_n$. If it is same as $C_1, \ldots, C_n$, then we have nothing to prove. So assume this is not the case. Let R_i denote the running time and S_i denote the swimming time of contestant O_i. Let T denote the time by which all contestants finish. Then there will be two consecutive contestants O_i, O_{i+1} such that $R_i \leq R_{i+1}$ (**1 mark**). Let T_i be the time at which O_i starts swimming. Then O_i finishes at time $T_i + S_i + R_i$, whereas O_{i+1} finishes at time $T_i + S_i + S_{i+1} + R_{i+1}$. So,

$$T \geq T_i + S_i + S_{i+1} + R_{i+1}.$$

   Now we consider a new schedule which is same as O but swaps O_i and O_{i+1} (**1 mark**). Here, the finishing time of O_i and O_{i+1} are $T_i + S_i + S_{i+1} + R_i$ and $T_i + S_{i+1} + R_{i+1}$, and the other contestants' finishing time remains unchanged. Since $R_i \leq R_{i+1}$, we see that $T_i + S_i + S_{i+1} + R_i, T_i + S_{i+1} + R_{i+1} \leq T_i + S_i + S_{i+1} + R_{i+1} \leq T$. Therefore, the new schedule is not worse than O (**1 mark**). By continuing this swapping process, we will get the greedy schedule, and the completion time would be no worse than T (**1 mark**).

Common Mistakes: Many students have got 5 out of 6, because they did not mention the last step. You need to continue such swaps to get to greedy. Many of them assumed that $R_i < R_{i+1}$, and so, one would get a strict improvement. But R_i could be equal to R_{i+1}.

In case you got the greedy rule wrong, I have given 0 or 1 marks depending on whether the approach for proof looked ok. This is more of a subjective assessment and I will not entertain any queries on this.

2. Let S be a set of $n+1$ distinct integers. You can assume that S is given as an array. You are given an unsorted array A of size n containing exactly n out of the $n+1$ integers in S. Give an $O(n)$ time algorithm to find the integer from S which is not in A. The only operation allowed on numbers in S (or A) is comparison (you are NOT allowed to perform addition, subtraction, multiplication, etc. on these numbers). (**5 marks**)

   Solution: We use divide and conquer strategy. First find the median of A in $O(n)$ time (**2 mark**). Call this x. Let A_L be the numbers in A which are less than or equal to x and S_L be the numbers less than or equal to x in S. Define A_R and S_R similarly (**1 mark**). Now, either $|A_L| = |S_L| - 1$, or $|A_R| = |S_R| - 1$. Depending on the case, we proceed recursively (**1 mark**). Since finding median takes $O(n)$ time, we get the recurrence $T(n) = T(n/2) + O(n)$ (**1 mark**), whose solution is $T(n) = O(n)$. If you used randomized version (like quick-sort), that is ok.

3. You are given an array A containing n numbers (which could be positive, zero or negative rational numbers). A sub-array $A[i,j]$ of A, where $i \leq j$, is defined by the sequence $A[i], A[i+1], \ldots, A[j]$. For each such sub-array, define $P(i,j)$ as the product of the entries in $A[i,j]$. Give an $O(n)$ time algorithm to find the largest value of $P(i,j)$ overall sub-arrays $A[i,j]$ (note that the algorithm just outputs a number). You can assume that arithmetic operations like multiplication on numbers in A take constant time. (**6 marks**)

   Example: Suppose A is $\{-3, 10, -6, 7, 2, -1\}$. Assuming that the first element of A is denoted by $A[1]$, the sub-array $A[1,4]$ has total product $-3 \times 10 \times -6 \times 7 = 1260$, whereas sub-array $A[3,6]$ has total product $-6 \times 7 \times 2 \times -1 = 84$.

   Solution: We use dynamic programming. We maintain two tables $S[i]$ and $L[i]$, for $i = 1, \ldots, n$. The entry $L[i]$ denotes the largest value of $A_i \cdot A_{i+1} \cdots A_j$ for $j \geq i$, and similarly, $S[i]$ denotes the smallest value of $A_i \cdot A_{i+1} \cdots A_j$ for all $j \geq i$ (**2 marks**). Base case is easy: $L[n]$ and $S[n]$ are both equal to A_n (**0.5 mark**). Now, for computing $L[i]$, we proceed depending on A_i being positive or negative. Assume the first case. Then observe that $L[i] = \max(A_i, A_i \cdot L[i+1])$, and $S[i] = \min(A_i, A_i \cdot S[i+1])$ (**1 mark**). If A_i were negative, then $L[i] = \max(A_i, A_i \cdot S[i+1])$, and $S[i] = \min(A_i, A_i \cdot L[i+1])$ (**1.5 mark**). Finally, we output $\max_i L_i$ (**0.5 mark**). We compute the entries starting from $i = n$ and decreasing i (**0.5 mark**).

   Many people got the definition of $L[i]$ (or $[S[i]$ wrong) – they just said the maximum produce in $A_i \ldots A_n$. But this is wrong – it needs to include A_i. Many of them define $S[i]$ as the largest negative number. This is wrong, but then one has to be careful with initial conditions – for example, what if all entries so far are positive ? I have taken off 1 mark if one is not careful with the initial condition.

4. A town has r residents $R_1, \ldots, R_r$, q clubs $C_1, \ldots, C_q$ and p political parties $P_1, \ldots, P_p$. Each resident is a member of exactly one club and belongs to exactly one political party. You want to form a governing council for the town. The governing council must contain exactly l_i members from club C_i, for $i = 1, \ldots, q$; but it can have at most u_k members from the political party P_k, for $k = 1, \ldots, p$. Give

an efficient algorithm which either finds such a council or declares that no such council is possible. Assume that the membership information is given in a suitable data-structure. (**4 marks**)

Solution: We reduce this problem to max-flow. We have a node c_i for each club C_i, a node p_i for each political party P_i and a node r_i for each resident R_i. If R_i is a member of club C_j and party P_l, then we have an edge from c_j to r_i and another from r_i to p_l, both of capacity 1. We add a source node s and a sink node t. We have edges from s to c_i with capacity l_i and from p_i to t of capacity u_i. We now find a max-flow from s to t, and check if its value is equal to $\sum_i l_i$.

To prove correctness, note that if there is a solution, then we can send a flow as follows: we send l_i units of flow from s to c_i, and 1 unit of flow from c_i to every r_j for those residents of C_i who get chosen. If R_i is selected, we send 1 unit of flow from r_i to the corresponding political party node, and then from every p_i the flow from p_i to t is equal to the flow received by p_i. Conversely, if there is a flow saturating all the edges (s, c_i), then by integrality of flow, each r_j receives either 0 or 1 unit of flow. If r_j receives 1 unit of flow, then we select R_i. Again, it is easy to check that this satisfies all the constraints.

Common Mistakes: There has been binary marking here – you either got this right or not. Many people missed out saying that given a valid flow, one needs integrality property to recover a solution. I have taken off 0.5 marks for this. Similarly many people mentioned wrong capacities – I have taken off 1 or 2 marks. If you got the graph wrong, you got maximum of 1 mark – again, a subjective assessment depending on your solution.

5. Consider the following optimization version of PARTITION problem. You are given n integers $x_1, \ldots, x_n$ and would like to partition them into sets A and B such that $\max(\texttt{Sum}(A), \texttt{Sum}(B))$ is minimized, where $\texttt{Sum}(X)$ denotes the sum of all the numbers in X. Given a solution A, B, define its value as the quantity $\max(\texttt{Sum}(A), \texttt{Sum}(B))$. Consider the following greedy algorithm – initialize sets A and B to emptyset. Consider the numbers $x_1, \ldots, x_n$ iteratively. When looking at x_i, if $\texttt{Sum}(A) < \texttt{Sum}(B)$, add x_i it to A, else add it to B.

   (a) Prove that there is an example for which this algorithm has value 3/2 times the minimum possible value for this example. (**1 mark**)

   Solution: Consider the numbers 1,1,2.

   (b) Prove that this algorithm has the property that for any input, its value is at most 3/2 times the minimum possible value for this input. (**6 marks**)

   Solution: Consider an input $x_1, \ldots, x_n$. let A, B denote the sets constructed by our algorithm, and let O be the optimal value for this input. Assume that $\texttt{Sum}(A) \geq \texttt{Sum}(B)$, and let Σ denote the sum of all the numbers $x_1, \ldots, x_n$. Let x_i be the last number added to A, and suppose the sum of numbers in A just before adding x_i was S (**1 mark**). Then our algorithm has value $x_i + S$ (**1 mark**). Also, the sum of B would be at least S (otherwise we would not add x_i to A (**1 mark**). So $\Sigma \geq 2S + x_i$. Now observe that $O \geq x_i$ (**1 mark**) and $O \geq \Sigma/2 \geq S + x_i/2$ (**1 mark**). Combining these, we get $3O/2 \geq x_i + S$, which is the value of our solution (**1 mark**).

   (c) Suggest an algorithm which would have better ratio than 3/2 – you need not prove anything, just give a one line algorithm. (**1 mark**)

   Solution: Run the same algorithm as above, but first sort x_i in decreasing order. If say "sorted order", you get 0.5 marks. If you say increasing order, it is clearly wrong.

6. Consider the following variant of max-flow: you are given a directed graph G with positive integer edge capacities, two vertices s and t. You would like to find a flow from s to t such that flow on every edge e is either 0 or u_e, where u_e denotes the capacity of e – we call such a flow a "saturating flow".

Prove that the following problem is NP-complete: given a directed graph G with vertices s, t and positive integer edge capacities, and a parameter k, is there a saturating flow from s to t of value at least k ? (**6 marks**)

Solution: This problem is clearly in NP. A solution just needs to give a flow of value k (**0.5 mark**), and a verifier needs to check that it is a valid flow and saturates every edge with non-zero flow on it (**0.5 mark**). The most common mistake is that many people did not mention that one needs to check flow conservation constraint. A solution will only mention f_e values on each edge, so the verifier needs to check this. Most of you got 0.5 marks here.

We prove NP-completeness by reduction from the subset sum problem. Consider an instance I of the subset sum problem, where are given numbers $x_1, x_2, \ldots, x_n$ and a number S. We would like to answer if there is a subset of these numbers which add up to S. We construct an instance of the saturating flow problem as follows. We have a vertices s and s'. For every element x_i, we add a new vertex v_i, edges from s to v_i and from v_i to s', both of capacity x_i. Now we add a new vertex t and add an edge from s' to t of capacity S. Now we claim that there is a solution to the subset sum problem if and only if there is a saturating flow from s to t of value S. To show this, suppose there is a solution to the subset sum problem, and let the subset be X. Then we can have a flow as follows – we send x_i unit of flow from s to v_i to s' if $x_i \in S$. Clearly the flow on s' to t would be S. Conversely, if there is a such a flow, then let x_i be the elements such that we have x_i flow on s to v_i edge. Then these elements add to S.

The most common mistake here is that students did not add the edge s' to t. Although this would have been correct if the problem definition had asked for a saturated flow of value exactly k, but it is not correct if one wants flow of value at least k. In other words, if you did not add this edge, a saturated flow of value at least S does not guarantee a solution to subset sum (which requires the sum to be exactly S). I have taken off 1.5 marks for this mistake.

Again the grading here has been binary – I could not find any correct reduction from a problem other than subset sum (or partition). Sometimes I gave 1 mark depending on how close you were to a correct reduction. Again, please don't come to get this extra mark.