COL 702 Advanced Data Structures and Algorithms

Major Exam, Sem I 2017-18, Max 80, Time 2 hr

Name ____________________ **Entry No.** ____________________ **Group**

Note Write in the space provided below the question including back of the page.
Every algorithm must be accompanied by a proof of correctness, time and space complexity. You can however quote any result covered in the lectures without proof.

1. Give a 1 line justification or construct counterexamples wherever applicable corresponding to each statement. **(2×5 marks)**

   (a) We can use hashing even when the keys are not totally ordered.

   Hashing works on unordered elements (including posets) for *searching* - however it doesn't support successor or predecessor search.

   (b) Given points $(1, 1), (2, 4) \ldots (n, n^2)$, we can construct their convex hull in $O(n)$ time.

   We can sort the x-coordinates in $O(n)$ time using radix sort and the apply Graham scan in another $O(n)$ time.
   There was some confusion if the given set of points were sorted, so I have given 1 marks if sorting was assumed. Moreover the lower bound of $\Omega(n \log n)$ is not applicable since integers in the range [1,n] can be sorted in $O(n)$ time.

   (c) Consider two problems $\Pi_1, \Pi_2 \in \mathcal{P}$. Then $\Pi_1 \leq \Pi_2$ in polynomial time.

   The reduction function f mapping $x \in \Pi_1$ is defined as follows -
   f first runs a polynomial time algorithm on a given instance $x \in \Pi_1$ and if it is YES then it maps to some fixed precomputed YES instance y of Π_2 else it maps to a fixed NO y' instance of Π_2.

   (d) If a graph has more than $|V| - 1$ edges and there is a unique heaviest edge, then the edge cannot be a part of the MST.

   If the edge is a *bridge* in non-biconnected graph, then it must clearly be included in the MST.

   (e) The problem to decide if a graph is 2 colourable is NP Complete.

   A 2 colorable graph can be recognized easily in linear time (it is a bipartite graph), so if it is NPC then $P = NP$.

2. An $n \times n$ grid has integer (possibly negative) labels on each square. A player starts from any given end square and travels to the other side by moving to one of the 3 adjacent squares in one step. The *reward* collected by the player is the sum of the integers in all the squares traversed by the player. Design an efficient (polynomial time) algorithm that maximizes the reward collected by the player. **(20 marks)**

	A	B	C	D	
1	26	13	−10	5	
START 2	21	−20	11	50	END
3	−5	45	18	−12	
4	32	11	−25	8	

Figure 1: An example - From the starting square 2A, the three possible steps are 1B, 2B, 3B. The reward corresponding to route indicated by the shaded squares is 21+45+18+50 = 134.

Let $A(i, j)$ denote the most profitable path to square (i, j) (row i, column j). Then we can write the following recurrence

$$A(i, j) = \min\{A(i-1, j-1), A(i, j-1), A(i+1, j-1)\} + C(i, j) \text{ for } j > 1$$

where $C(i, j)$is the reward corresponding to square (i, j). For $j = 1$ (the starting column) $A(i, 1) = C(i, 1)$. The recurrence is justified since path ending at (i, j) can be reached from only these three squares and the route till then must also be best possible.

This is a dynamic programming algorithm that starts from the first column and proceeds from the leftmost to the rightmost column and each $A(i, j)$ can be filled with 3 table look-ups. This implies $O(n^2)$ time and $\max_i A(i, n)$ is the final answer. The space required is $O(n)$ since only the previous column needs to be stored.

Common problems
(i) Unclear about the final objective function, including what are source destination.
(ii) Confusion about the direction of the forward path.
(iii) Writing a recurrence without defining the function or parameters and not explaining the basis of the recurrence and why it is correct.
(iv) Greedy algorithm doesn't work.

3. The partition problem is defined as follows -
Given a set S of n integers, can we partition S into sets A and B such that $\sum_{x\in A} x = \sum_{y\in B} y$? For example, if the set of numbers is $\{2, 4, 5, 6, 7\}$, then the answer is yes, since 2+4+6 = 5 +7.
Show that, if there is a polynomial time algorithm for the 0-1 Knapsack problem, then the partition problem can be solved in polynomial time. **(5 marks)**

Consider a knapsack problem where the size of knapsack is $\sum_{x\in S} = T$ If T is not even then return NO. Else define a knapsack problem with the following parameters - capacity $= T/2$. Let the profits $p_i = w_i = x_i$ for each element $x_i \in S$. If the optimal solution has value $T/2$, then the partition has a solution and the converse is also true, i.e., if the optimal solution is not $T/2$, then there is no partition. Therefore, if the knapsack problem has a polynomial time solution, then so does the partition problem.

Common problems

(i) Knapsack problem has been modified and the parameters are not defined. A 0-1 knapsack must be specified with knapsack capacity, weights and profits for each element. A knapsack problem returns the maximum profit. NO OTHER VERSION IS ACCEPTABLE FOR REDUCTION.
(ii) The solution from partition and knapsack have not been related formally - the "iff" condition has not been formally defined.
(iii) Knapsack has been reduced to partition which is completely wrong.

(ii) In the above partitioning problem, suppose the n integers are in the range $[1, n]$, then design an efficient algorithm for the partition problem. **(10 marks)**

Let $P(i, j)$ denote that there is a subset of $\{x_1, x_2 \ldots x_i\}$ that sums up to j. Here $1 \leq i \leq n$ and $1 \leq j \leq n^2$. we can write the following recurrence for $P(i, j)$.

$$P(i, j) = P(i - 1, j) \text{ OR } P(i - 1, j - x_i)$$

and $P(1, x_1)$ is True. The two dimensional table is filled in increasing order of i and for a fixed i, in increasing order of j. At most two lookups are necessary to fill up a table entry, so the algorithm takes $O(n^3)$ steps. The final answer is YES iff $P(n, T/2)$ is true, false otherwise.

Common problems
(i) Incorrect or incomplete recurrence without correctness proof.
(ii) Absence of a formal recurrence and some imprecise description of the DP.

4. Consider the following algorithm for constructing a convex hull of a set S of n points in a plane.

   Mergehull($CH(S)$)

   (i) Split S into two roughly equal parts S_1 and S_2.
   (ii) Construct their convex hulls $CH(S_1)$ and $CH(S_2)$ recursively.
   (iii) Merge $CH(S_1)$ and $CH(S_2)$ to obtain $CH(S)$.

   Provide the detailed implementation of each step, particularly Step (iii) and analyze the algorithm to bound its running time in the best possible manner. **(15 marks)**

   Step (i) can be implemented by arbitrarily dividing the input into sets with about $n/2$ points each (median finding is not required).
   Step (ii) is just a recursive call and Step (iii) merges the recursively constructed convex hulls with maximum $n/2$ points each. We assume that the procedure returns the points on the hull in an ordered sequence. To merge two (possibly overlapping hulls), H_1 and H_2 we first create a sorted sequence in x direction. This can done by considering the upper hull and lower hull H_1^u, H_1^l of H_1 and upper and lower hulls H_2^u, H_2^l of H_2. These are sorted in x directions, so we can merge the 4 sequences in $O(n)$ steps. For the combined sequence H, we can separate them into upper and lower hulls by identifying the extreme x coordinates p_l and p_r and then separating the points below and above the line (p_l, p_r). This takes $O(n)$ steps.
   Following this we can construct the upper and lower hulls by running Graham Scan on the two sorted sequence in $O(n)$ time. Therefore the recurrence for the running time satisfies

   $$T(n) = 2T(n/2) + O(n) \quad T(3) = O(1)$$

   which implies that $T(n) = O(n \log n)$.

   Common Problems
   (i) The recursive format has not been adhered to. For example calling Jarvis' march or Grahan scan recursively doesn't make any sense.
   (ii) Merging separable convex hulls by finding tangents is not described with adequate details. While the approach is correct, it requires very careful reasoning about why all pairs of points, i.e., n^2 need not be tried to find the tangents. The ordering of slopes of the convex hull is exploited. Without this rigorous justification, marks have been penalised.

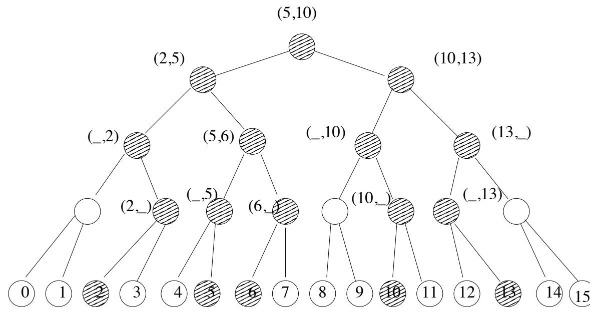

Figure 2: The shaded leaf nodes correspond to the subset S in a universe of size 16. The pairs (a, b) associated with a node corresponds to the maximum and minimum marked nodes in the left and right subtrees respectively and undefined if there is no marked nodes in the subtree. For example, the path starting at 8 encounters the first shaded node on the path from 10 implying that the successor of 8 is 10 and the predecessor is 6 which can be precomputed

5. We would like to design a data structure that supports *predecessor* queries faster than doing a conventional binary search. The elements are from an universe $\mathcal{U} = \{0, 1, 2 \ldots N - 1\}$ where N is a power of 2. Given any subset S of n elements ($N \leq n^{\log^4 n}$), we want to construct a data structure that returns $\max_{y \in S} y \leq X$ for any query element $X \in \mathcal{U}$. We consider a hypothetical complete binary tree on $\mathcal{U}$ where all the elements are in the leaf nodes. So there are $\log N$ levels. The elements of S are marked in the corresponding leaf nodes and corresponding leaf to root paths are also marked. Each internal node of the binary tree T stores the smallest and the largest element of S in its subtree. If there are none, then these are indicated. This information can be computed at the time of marking out the paths to the node. The smallest and the largest element passing through a node can be maintained easily.

   Given a query X, consider the path from the leaf node for X to the root. The predecessor of X can be uniquely identified from the first *marked* node on this path since we can identify the interval of S that contains X. Building on this scheme
   (i) Complete the details of the search procedure, and analyze the query time and space complexity. **(10 marks)**

   Given a query X, consider the path from the leaf node for X to the root. The predecessor of X can be uniquely identified from the first *marked* node on this path since we can identify the interval of S that contains X. We can sort the set S in $O(n \log n)$ time and compute the predecessor/successor information.

   Note that all the ancestors of a marked node are also marked. So from any given leaf node, if we must identify the closest marked ancestor. With a little thought, this can be done using an appropriate modification of binary search on the path of ancestors of X. Since it is a complete binary tree, we can map them to an array and query the appropriate nodes. This takes $O(\log(\log N))$ steps and so for $N = \Omega(n^{polylogn})$ - this is $O(\log \log n)$. This is much superior to $O(\log n)$ by normal binary search.

   Clearly, we cannot afford to store the binary tree with N nodes. So, we observe that it suffices if we store only the n paths which takes space $O(n \log N)$. Note that these can be hashed appropriately to enable binary search on the paths since we can calculate the addresses of nodes of a complete binary tree. Using universal hashing, the expected search time is $O(\log \log n)$. For part of the tree addresses that are not hashed, it will return failure.

(ii) Suggest ways for reducing the space complexity to $O(n)$. **(10 marks)**

For a further reduction, we can do a two phased search. In the first phase, we build a search tree on a uniform sample of $n/\log N)$ keys which are exactly $\log N$ apart, so that the space is $O(\frac{n \log N}{\log N}) = O(n)$. In the second phase we do a normal binary search on an interval containing at most $\log N$ elements that takes $O(\log \log N)$ steps.

Common Problems
Most answers mentioned $O(\log N) = O(\log^5 n)$ for the first part which is worse than binary search time of $O(\log n)$. This defeats the entire purpose of the exercise that too with space $O(N)$.