CSL 630 Data Structures and Algorithms

Major Exam, Sem I 2014-15, Max 80, Time 2 hr

Name _____ **Entry No.** _____ **Group**

Note (i) Write your answers neatly and precisely. You won't get a second chance to explain what you have written. Partial credits will be hard to come by.

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. We want to maintain a data structure on a (dynamic) set of elements S such that we can find the median in $O(1)$ time. The data structure should support insertion, deletion and extract-median operations in $O(\log n)$ time for n elements. Moreover, the data structure on n elements can be constructed in $O(n)$ time initially. **(15 marks)**
   Hint: Recall that a heap maintains the minimum/maximum element with similar properties.

   For a set S with n elements, let us denote the the i-th ranked element by S_i. Then the median element is S_m (for n even $m = n/2$ and for n odd $m = (n+1)/2$. We will maintain two heaps - a max-heap for elements $\leq S_m$ and the other a minheap for elements $\geq S_{m+1}$. Observe that the maximum element of H_1 is the median and we will maintain this property.

   - *Build-heap* Select the median and partition into two subsets based on the median. Then construct the two heaps - call them H_1 and H_2. All the steps can be done in linear time.
   - *Report Median* Return the max element of H_1. Time: $O(1)$.
   - *Insert x* By comparing x with S_m, we will will know which of the heaps it should belong. Suppose, $x > S_m$ - then we will insert in H_2. Insertion of x would change the number of elements and therefore, the median itself could change. We will have to restore the invariant on H_1 and H_2. For $x > S_m$, the minimum element of H_2 is the new median, so we will delete (extract-min) from H_2 and insert into H_1 using the normal algorithms for min-heap and max-heap.
     Overall the time taken is $O(\log n)$.
   - *Delete* Similar to insert and we may have to insert delete from H_1 and H_2 to maintain the invariant. Time: $O(\log n)$.
   - *Extract Median* Special case of the delete operation. Note that the minimum element of H_2 becomes the new median and it must be transfered from H_2 to H_1.

   Typical answers Many answers have tried to combine trees and heaps in some ambiguous manner where the build-heap takes $O(n \log n)$ time. They have tried to maintain median in the root of an AVL tree without the right reasoning. Actually having the median as the root element was not a requirement and may not be easily achieved. One would have to maintain an explicit pointer to the median wherever it is or argue that the median occurs within $O(1)$ distance from the root.

   There were also attempts to define a *median* heap in the same way as minheaps but there were no clear invariants that could be maintained.

   All these schemes essentially are balanced BSTs to maintain a sorted set. I have awarded about 50% of the credit for such schemes.

2. Given a sequence of real numbers (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 sequences.
For example in the sequence -3, -1, 4, , 6 , -3, 5, -4, 2, the subsequence that attains the maximum sum is 4,6,-3,5.
Design a linear time algorithm for this problem. **(15 marks)**

We will develop an induction definition of the problem. Let $L(i)$ denote the longest monotonically increasing subsequence in $x_1, x_2 \ldots x_i$ and $L'(i)$ be the sum of the elements of $L(i)$. Initially $L(1) = x_1$. L_{i+1} is either L_i or x_{i+1} is the last element of $L(i+1)$. To handle the second case, we must also keep track of the longest subsquence ending at x_i - call it $E(i)$.Note that $L(i)$ may be the same as $E(i)$ - initially $E(1) = L(1)$. A subsequence can be denoted by the starting and ending indices. So

$$E(i+1) = E(i) \cdot x_{i+1} \quad if \quad E'(i) > 0, \quad else \quad x_{i+1}$$

This can be seen as follows. Let the maximum subsequence ending at $i+1$ be $x_j, x_{j+1}, \ldots x_{i+1}$. If this sequence has length at least 2, then, $E(i) = x_j, x_{j+1} \ldots x_i$, otherwise there is a contradiction in the defn of $E(i) or E(i+1)$. If $E'(i)$ is -ve, then $E(i+1) = x_{i+1}$.

$$L(i+1) = \max\{L(i), E(i+1)\}$$

The algorithm returns $L(n)$ and each of the n updates can be done in $O(1)$ time. So the algorithm runs in $O(n)$ steps.

Alternate view The maximum subsequence = $\max_i E(i)$. We can make two passes through the array - once for computing $E(i)$ and in the next pass, report the maximum among all $E(i)$.

Typical answers Most answers wrote code of the above algorithm without adequate exlanation or a formal correctness proofs. They have been awarded between 10 -12 marks. Proof of correctness is not explaining the algorithm but **Why** the correct answer will be output - typically done using induction which is the recurrence in this case.

3. While constructing a skip list, Professor Thoughtful decided to promote an element to the next level with probability p ($p < 1$) and calculate the best value of p for which the product $E_q \times E_s$ is minimized where E_q, E_s are the expected query and expected space respectively. He concluded that $p = 1/2$ is not the best. Justify, giving the necessary calculations. **(15 marks)**
You may need to use the fact that $\log_2 3 > 3/2$.

   Follwing the calculations done in the lecture , $E_q = O(\log_k n)$ and $E_s = O(\frac{1}{1-p} \cdot n)$ where $k = 1/p$. Both of these bounds follow from the expectation of geometric random variables having success probabilities p and $1 - p$ respectively (for the space bound the probablity that an element is not copied is $1 - p$). So, the expression

$$E_q \times E_s = O(n \times \frac{\log_2 n}{\log_2(1/p)} \times \frac{1}{1-p})$$

   For the best value of parameter p, we need to maximize the denominator $\log_2(1/p) \cdot \frac{1}{1-p}$. The right method is to use calculus to maximize the expression by taking derivative w.r.t. p. However, you can verify that $p = 1/3$ gives a larger value than $p = 1/2$.

   Partial marks have been given who have attempted to derive the general expression and maximize it, even if they didn't establsh that $p = 1/3$ gives a better performance than $p = 1/2$.

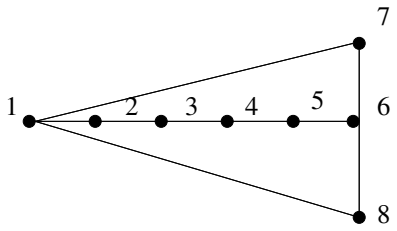
4. The *girth* of an undirected, unweighted graph is the length of the smallest cycle in a graph. Design an efficient algorithm for finding the girth of a graph.
Hint: For any edge e, how do you find the smallest cycle containing that edge ? **(15 marks)**

Let $G = (V, E)$ be the given graph. We observe that
Claim : For any edge $e = (u, v)$ the smallest cycle containing e, which we denote by $C^*(e)$ comprises of the shortest path from u to v (not including (u, v)) and the edge (u, v). It is ∞, if e does not lie on any cycle.

Proof By contradiction - Suppose the smallest cycle is not the shortest path between u and v in $G - \{e\}$. Then we can obtain a smaller cycle by using the shortest path between u and v.

Since the girth of the graph $g(G) = \min_{e \in E} C^*(e)$, we can compute it by shortest paths in the graph $G-(u, v)$ for all edges $e = (u, v)$. This no more than $m \cdot T_{SSSP}$ where T_{SSSP} is the time for computing single source shortest paths which is polynomial time - for undirected, unweighted graph this can be done using BFS in $O(m)$ steps. Using more careful computation in a BFS tree it is possible to do this faster.

Typical answers Many answers have tried to use DFS with some variations. Note that DFS cannot be used to find shortest paths (or shortest cycles) since it goes depth first. There are many possible cycles in a graph (exponentially many) and DFS will discover these cycles in some unpredictable order - if you try to bring in the distance from root, then it is BFS and that should be recognized. For example, consider the graph below. When you do DFS from vertex 1, the shortest cycle 1, 8, 6, 7, 1 may not show up if you go 1, 2, 3, 4, 5, 6, 7, (6) 8. (since it contains two back-edges. So it depends on the DFS numbering that you cannot pre-compute.



So, no credit has been given for DFS based solutions.

5. Given a set S of n integers (possibly with repetitions), and a number m, we want to determine if there is a subset $P \subset S$ such that $\sum_{x \in P} x = m$. For example, for $S = \{10, 11, 30, 6, 3, 25\}$, the answer is YES for $m = 46$ but it is NO for $m = 22$.

   (a) Is the above problem an NP complete problem ? Justify.
   You can assume that the problems discussed in class like 3SAT, partition and vertex cover are known NPC problems. **(10 marks)**

   The problem is in NP since we can guess the subset of numbers that adds up to m and then add to verify if it is indeed so.
   For for proving that it is NPC, we can reduce the partition problem to it by choosing $m = \frac{1}{2} \cdot \sum_{xinS} x$. Clearly this can be done in polynomial time.
   Note: When the sum is not even, we can map it to $m = \infty$ which will have a NO answer.

   (b) If m is a $2 \log n$ bit integer, design an efficient polynomial time algorithm. **(10 marks)**

   We can solve this using a dynamic programming approach similar to the knapsack problem. Let $S(i, j)$ be true iff there is a subset of elements in $x_1, x_2 \ldots x_i$ that sums to j where $-m \leq j \leq m$. The running time is $O(n \cdot m$ where $m \leq n^2$ since it is a $2 \log n$ bit number.