Illinois Institute of Technology
Department of Computer Science

Solutions to Second Examination

CS 430 Introduction to Algorithms
Spring, 2009

11:25am–12:40pm, Wednesday, April 8, 2009
104 Stuart Building

1. Dynamic Programming **[25 points]**

   Let the n integers be a_i, $1 \leq i \leq n$.

   One way to solve this problem is to generate a table T of n elements, where the $T_i = \sum_{k=1}^{i} a_k$. $T_i = T_{i-1} + a_i$, so these values can be computed in $\Theta(n)$ time. Then $T_j - T_i = \sum_{k=j}^{i} a_k$. Thus, for 12 points of partial credit, one can simply go through all pairs (i, j), $i \leq j$ and pick the maximum difference $T_j - T_i$ found. Since there are $\binom{n}{2} = \Theta(n^2)$ such pairs, the overall running time will be $\Theta(n^2)$.

   For full credit, however, we may observe that the maximum difference $T_j - T_i$ will occur between the largest value T_j and the smallest value T_i, for which $j \geq i$. Thus, we can find the maximum sum with a search of the T array from left to right keeping track of three items:

   - The smallest value s seen so far
   - The largest value l seen after s
   - The largest contiguous sum σ seen so far

   As we inspect an element e in the array, we consider two possibilities:

   - $e > l$. In this case we substitute $l \longleftarrow e$, since we have come upon a contiguous vector with larger contiguous sum. We update σ if necessary.
   - $e < s$. In this case, our contiguous vector has terminated. We compute $l - s$ and compare it to σ, recording the greater value in σ. We reset $s \longleftarrow e$ and $l \longleftarrow e$ and update σ if necessary.

   Since we use a constant amount of time per array element, the overall running time is $\Theta(n)$.

   Another linear-time solution is to define $S_0 = 0$, $S_i = S_{i-1} + a_i$ if $S_{i-1} > 0$ and $S_i = a_i$ if $S_{i-1} \leq 0$. Induction shows that S_i is the maximum sum ending at a_i. The sequence of S_i can be can be computed in $\Theta(n)$ time; the maximum of that sequence, which can then be computed in $\Theta(n)$ time, is the desired result.

2. Double-deletion Heaps **[25 points]**

(a) Let G_k be the minimum number of nodes in a tree whose root has degree k. We find that $G_0 = 1$, $G_1 = 2$, and $G_2 = 3$. For values larger than this, we have a root and k subtrees. When the tree was constructed, there was one subtree of each degree from 0 to $k-1$. However, since each of these subtrees can have lost its two largest children, the first three subtrees (of original degree 0, 1, and 2) are now just a single node. The others can be as small as the minimal subtrees with degrees ranging from 1 to $k-3$. Thus, $G_k = \sum_{i=1}^{k-3} G_i$ for $k \geq 3$. This is sufficient for the problem, but it can be simplified to $G_k = G_{k-1} + G_{k-3}$ without too much additional work.

(b) In order to have the same amortized running time, the degree of a node needs to grow logarithmically in terms of the minimum number of nodes in its subtree. Put another way, the minimum number of nodes must grow exponentially in terms of the degree of the node. If this occurs, then the analysis for running times follows as in CLR. (This is the case for Double-deletion heaps.)

3. Balloons **[25 points]**

A greedy heuristic works to pop all balloons. Sort the balloons by the position of the leading edge (easily calculated from the x_i and r_i). Then, starting at the left end, we scan through the list of balloons until we reach the trailing edge of a balloon that has not been popped yet, and then fire at this location. This shot dispatches all balloons with lower starting position. Continue this process until all the balloons are popped. Since the leading edge of each balloon is visited once and each shot pops a balloon, the scan takes $O(n)$ time. Allowing the $O(n \log n)$ time for sorting the balloons, the entire algorithm takes $O(n \log n)$.

4. Combined-Min

(a) **[12 points]** In order to sort n items, we do the following three steps:

- `make`(x) for each item x; pad with ∞ as necessary to get the number of elements to be a power of 2
- Group the sets in pairs and `combine-equal` to get sets of two items. Group the resulting sets into pairs and `combine-equal` to get sets of four items. Continue until you have one big set of all the items
- `delete-min` n times and keep track of the values it returns, which will be the n items in sorted order

(b) **[13 points]** Suppose that `combine-equal` runs in time $o(n)$ and `delete-min` runs in $o(\log n)$. We claim that the procedure we have above then sorts in time $o(n \log n)$, violating the information-theoretic bound on comparison-based sorting.

The `make` step in the algorithm requires $\Theta(n)$ time since it is applied to n items. The `combine-equal` step iteratively combines $\frac{n}{2^i}$ sets of size 2^i, for $\log n$ iterations. This step runs in time:

$$\sum_{i=1}^{\log n} \frac{n}{2^i} \quad o(2^i) = o(n \log n)$$

The final `delete-min` computation is done as n operations, each requiring time $o(\log n)$, giving a running time of $o(n \log n)$. Thus, the overall running time for our algorithm is $\Theta(n) + o(n \log n) + o(n \log n) = o(n \log n)$, contradicting our sorting lower bound.