15-750 Graduate Algorithms

Homework # 1 **due: January 30th**

Groundrules:

- Homeworks will generally consist of easier *exercises* and harder *problems*. You should do the exercises by yourself, but you may work with a friend or two on the problems if you want — just write down who you worked with. One exception: no fair working with someone who has already figured out (or already knows) the answer. Of course, your writeup should be in your own words.
- If you've seen a problem before (sometimes we will give problems that are "famous"), then say that in your solution (it won't affect your score, we just want to know). Also, if you use any sources other than the book or those on the web page, write that down too. It's fine to look up a complicated sum or inequality, but don't look up an entire solution.

Exercises:

1. Expectations

   We know that if X and Y are two random variables, then $\mathbf{E}[X] + \mathbf{E}[Y] = \mathbf{E}[X+Y]$.

   (a) Give an example of random variables X and Y such that $\mathbf{E}[X \cdot Y] \neq \mathbf{E}[X] \cdot \mathbf{E}[Y]$.

   (b) Two random variables X and Y are independent if for all values x and y they can take on, $\mathbf{Pr}(X = x \wedge Y = y) = \mathbf{Pr}(X = x) \cdot \mathbf{Pr}(Y = y)$. (Assume our random variables are discrete.) Prove that if X and Y are *independent*, then we do get $\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$.

2. Finding triangles.

   Suppose we want to tell if a given graph G has a triangle (a 3-clique). One way to do this is to go through the nodes one at a time and for each one, to see if there is an edge between any two of its neighbors. If all nodes have degree d, then this takes time $O(nd^2)$. If the graph is dense (e.g., $d = \Omega(n)$), then this algorithm could take time $\Omega(n^3)$. Give a different algorithm that takes time $O(n^{3-\epsilon})$ for some constant $\epsilon > 0$, even if the graph is dense. Hint: use an adjacency-matrix representation of the graph.

Problems:

3. Probability of a treap.

   Define the *size* of a node in a binary tree to be the number of nodes in the subtree rooted there (including the node itself). In a specific n-node tree T, let $s_1, s_2, \ldots, s_n$ be the sizes of the nodes. Prove that the probability that an n-node treap is isomorphic to T (has the same shape) is:

$$\prod_{1 \leq i \leq n} \frac{1}{s_i}$$

4. Simulating a tree using a set of arrays

   Almost-president and amateur hacker Al Goreithm wants to devise a data structure that stores a set of keys, and supports search and insertion. However, he doesn't want to use trees. The proposed structure instead consists of a collection of arrays, $A_0, A_1, A_2, \ldots$, where A_i has size 2^i. Each key is stored in exactly one of the arrays, and each array keeps its keys in sorted order. The invariant kept by this algorithm is that each array A_i is either full or empty. For example, if there were a total of 11 elements, then A_0, A_1, and A_3 would be full, and the rest would be empty. Also, while each individual array is in sorted order, there is no ordering relation between arrays (e.g., perhaps $A_0 = [5]$, $A_1 = [2, 6]$, and $A_3 = [1, 3, 4, 7, 8, 9, 10, 11]$).

   Searching for an element x is implemented by going through the non-empty arrays in order from smallest to largest, and for each one, using binary search to determine if x is in it or not.

   (a) Suppose the structure has n keys. What is the worst-case running time per search operation. (You can assume n is of the form $2^i - 1$).

   (b) Explain how to implement insertion in such a way that the amortized cost per insert is $O(\log n)$, where n is the total number of insertions done.

   (c) Explain why this scheme gets into trouble if we want to also support deletions (while maintaining the above invariants).

5. Supporting deletions

   We can augment the above data structure to support deletions as follows. When we want to delete an element, we leave it in place, but simply mark it as deleted. We also incremement a delete-count for the array that the element is in. Whenever the delete-count of an array A_i reaches 2^{i-1} (i.e., half of its elements are deleted) we remove the deleted elements and then either (a) move what's left into A_{i-1} if A_{i-1} was empty, or (b) merge what's left with A_{i-1} back into A_i if A_{i-1} was full. So, we maintain the invariant that each array is either empty or full (but "full" may include elements marked as deleted). We also maintain the invariant that at most half the elements in any array are marked as deleted.

   Prove that this method continues to support efficient searches and inserts, while also supporting deletions in amortized time equal to the search time (we need to find the element first) plus $O(1)$.

   Be clear and careful: for instance, you need to argue that the deletions don't ruin the good time bounds for insertion. It may help to think of each insert operation as arriving with $O(\log n)$ insert-tokens (to be placed at appropriate points in the data structure) and to think of each delete operation as arriving with $O(1)$ delete-tokens (also to be placed at appropriate points in the data structure).