Algorithms I

Tutorial 2 Solution Hints

August 19, 2016

Problem 1

We create a balanced BST from the given integers. At each node, we also maintain the size of the sub-tree rooted at this node in the BST. Now, given any given x, we start from the root and search for x. We maintain a variable *count* initialized to 0. Whenever, we follow the right child of a node, we add the size of sub-tree of the left child of the node to *count*. We also add 1 more for the current node itself. After the search is finished, *count* has the answer.

Problem 2

We maintain two balanced BSTs. Both the BSTs will contain the n pairs given initially. One of them uses a_i as the key while the other one uses b_i. For search with x, we can search in both the BSTs for key x. For delete we do the same except once we find a pair (x, y) or (y, x). We delete the pair from both the BSTs.

Problem 3

First we sort the given pairs with respect to a_i. Now for any i, for all j such that $j < i, a[j] < a[i]$. So, for each i, we now need to find number of j such that $j < i$ and $b[j] < b[i]$.

We iterate through the array from index 1 to n. We also maintain a balanced BST. Whenever we are at index i, the BST contains all b_j such that $j < i$. So, at index i, we just need to know how many elements in the BST are less than b_j. This can be done using the solution for Problem 1.

Problem 4

We iterate over the array in the order $1, 2, 3 \ldots n$. We maintain a balanced BST. At any index i, $i < k$, we insert $a[i]$ in the BST and assign $b[i]$ to be the maximum element in the BST. for $i > k$, we first delete $a[i - k]$ from the BST. After that we insert $a[i]$ in the BST and assign $b[i]$ to be the maximum element in the BST.

Problem 5

We create a balanced BST. At each node, we also maintain a *count*, which is the number of elements in the array with key equal to this node. We insert all a_i in the BST. If a_i was not present in the BST before, we create a new node and initialize its *count* to be 1. If it's already present, we just increment *count* by 1. After all the elements have been inserted, we can do an inorder traversal and get the sorted sequence.

Problem 6

We create a balanced BST where each node also stores a count of the key. We insert all the elements of the array in the BST (Similar to the previous problem). We also have a min heap of size k that stores $(count, key)$ indexed by count. After insertion, we do a traversal of the BST (any of preorder, inorder or postorder is fine). At any node during the traversal, if the size of heap is less than k, we just insert it's count and key in the heap. If the size is more than or equal to k, we delete the min element from the heap and then insert the count and the key of current node.

Problem 7

We maintain a min heap of size k where each node stores (x, i, j), where $x = A_i[j]$. We use x as the key for the heap. Initially we insert $(A_i[0], i, 0)$ for $1 \leq i \leq k$. Now, we repeat this n times:

- Extract and delete min element from the heap, let's say $(A_i[j], i, j)$
- Check if $j + 1 < |A_i|$. If yes, we insert $(A_i[j + 1], i, j + 1)$.

Problem 8

Try it yourself

Problem 9

We iterate over a from index 1 to n. We also maintain a hash table. At index i, we check if $s - a[i]$ is already present in the hash table. If yes, we report that we have found such pair of indices. Otherwise, we insert $a[i]$ in the hash table and continue searching.

Problem 10

We first create an array p such that $p[i]$ gives the prefix sum till index i i.e. $p[i] = \sum_{j=1}^{i} a[i]$. This can be done using the following recurrence, $p[0] = a[0]$, $p[i] = p[i-1]+a[i]$. Now, we iterate over array p in the order $1, 2 \ldots n$. We also maintain a hash table which stores $(x, count)$ where x is the key and $count$ is the number of i such that $p[i] = x$. We initialize a variable $answer$ to the number of i such that $p[i] = 0$ and iterate over array a from 1 to n. At index i, we search for $p[i]$ in the hash table. If don't find $p[i]$, we just insert $(p[i], 1)$ in the hash table. If we find $p[i]$, let's it's corresponding value in the hash table be c, we add c to the $answer$ and update the value of $p[i]$ in the hash table to $c + 1$. Finally, $answer$ contains the required number of sub-arrays.