CSCI-GA.1170-001/002 Fundamental Algorithms October 8, 2014

Solutions to Problem 1 of Homework 5 (10 points)

Name: Keeyon Ebrahimi *Due: Wednesday, October 8*

(a) (4 points) Suppose we want to sort an array A of n elements from the set $\{1, 2, \ldots, (\log n)^{\log n}\}$. Show how to sort this array in time $O(n \log \log n)$.

Solution: Because we know that the range is small, this is the best way to do this sort.

As we learned with the Radix sort, we can first mod each value by 10, to give us the one's digit. We now want to sort all the numbers based soley on the one's digit. Because this has a small range as well, we know that the counting sort is a perfect, for we will not have to worry about the space of 10 elements. We then sort these elements based on the ones digit. Then we mod by 100 and sort by the tens digit, sort those with the counting sort, and we continue down this path until completion. An important thing to note on this is that this is a stacking sort, meaning that values that are equivalent are sorted based on which came earliest in their original array. This is important to note because there the Counting sort is made possible because of this feature. □

(b) (6 points) Suppose we want to sort an array A of n integers such that the total number of distinct integers in A is $O(\log n)$. Show how to sort this array in time $O(n \log \log n)$.

Solution: A way to solve this is to make a modified binary search tree where while we are creating the binary search tree, when we hit a duplicate, just store its count in a different array.

1. Create a binary search tree. While creating if we check if we have a repeating element and store its count in an array
2. Find Min of Binary Search Tree
3. Append this item into a new array X times, based on its count we stored from earlier.
4. Delete Min from Binary Search Tree
5. Repeat steps 2 - 4 until Binary search tree is empty.

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 8, 2014

Solutions to Problem 2 of Homework 5 (6 points)

Name: Keeyon Ebrahimi *Due: Wednesday, October 8*

An n-element $A = \{a_1, a_2, \ldots, a_n\}$ array is said to be *k-sorted* if the first k elements are each less than each of the next k elements, which in turn are less than the next k elements, and so on. More precisely,

$$a_{(i-1)k+j} \leq a_{ik+\ell} \text{ for all } 1 \leq i < n/k, \;\; 1 \leq j, \ell \leq k \; .$$

Assume you are a given any (e.g., completely unsorted) array A, and only wish to k-sort A, meaning that you only wish to rearrange the element of A so that they become k-sorted, as defined above. Notice, there are many possible valid answers for any given A, and the algorithm is allowed to choose any one of such answers.

Assuming n is a multiple of k, show that any valid, comparison-based k-sorting algorithm for A requires $\Omega(n \log(n/k))$ comparisons. (**Hint**: You may either do this directly using a decision tree argument (this will require using Stirling's approximation), or you can have a sleeker indirect argument using the lower/upper bound for sorting.)

Solution: What we will be doing here is a modified version of Quicksort. We will

1. Pick a random pivot
2. Put all elements smaller than pivot to its left and larger to its right, just like in Quicksort.
3. Here is the difference. In Quicksort we now place the pivot in its final array position,**instead**, we will be looking at the size of the two subarrays created, and if they are of size k or less, then we just place them into our final array at their current position. If they are of a size larger than k, we repeat the process.

Explanation as to why this takes $\Omega(n \ \log(n/k))$ comparisons.

Of the $n!$ different ways the array can be set up, there are $\frac{n!}{k!}$ different arrays that satisfy our problem.

Now lets say we have $f(n)$ steps. Comparisons have two options, so we have a total of $2^{f(n)}$ cases. We now know $2^{f(n)} \geq \frac{n!}{k!}!$ which means $f(n) \geq log_2(fracn!k!)$.

Stirling's Approximation shows that $log_2(fracn!k!) = \Omega(n \ \log(n/k))$ steps, giving us the lower bound of the claim. As for the upper bound. The altered quick sort has a running time of $\Omega(n \ \log(n/k))$ because we stop the sort process at size k. □

CSCI-GA.1170-001/002 Fundamental Algorithms October 8, 2014

Solutions to Problem 3 of Homework 5 (6 points)

Name: Keeyon Ebrahimi *Due: Wednesday, October 8*

Suppose you are given two sorted lists A, B of size n and m, respectively. Give an $O(\log k)$ algorithm to find the k-th smallest element in $A \cup B$, where $k \leq \min(m, n)$.

Solution: Because we need to find the solution in $O(log\ k)$ time, we cannot just iterate through, and instead have to be cutting off a part of our test options with each iteration.

Say we set variables i and j such that $i + j = k - 1$, meaning that $i + j + 1 = k$.

If $B[j-1] < A[i] < B[j]$, then we know $A[i]$ is the solution. Because we are at the $j - 1 + i$ element, which is k. The same can be said for $A[i-1] < B[j] < A[i]$ as well

If neither of these cases pass, then we compare $A[i]$ and $B[j]$. If $A[i] < B[j]$, then $A[i] < B[j-1]$. This is because we have tested if it is between the two and it is not. Because $A[i]$ is less than both of these values, then we that $A[0...i]$ do not contain element k, because this means that $i = k-j-2$, which is less than our k element.
We can also use the exact same logic for array B and if $B[j] < A[i]$

At this point we now rid the lower half of either A or B based on the previous comparison, and subtract our the amount of eliminated values from k and try again. □

CSCI-GA.1170-001/002 Fundamental Algorithms October 8, 2014

Solutions to Problem 4 of Homework 5 (4 (+4) points)

Name: Keeyon Ebrahimi *Due: Wednesday, October 8*

(a) (4 points) Given an array A of size n and the fact that there is an element x that occurs at least $1 + \lfloor n/2 \rfloor$ times in A, design an $O(n)$ time algorithm to find x.

Solution: The essential part here is that the element occurs $\frac{n}{2} + 1$ times. The $+1$ part is essential for this algorithm to work.

We need to iterate through the array and have a counter variable that starts at zero. We also need a currently checking variable as well. As we iterate through, if the counter is at zero, we make the current item the currently checking variable. If not, then we compare the current item with the currently checking variable. If they are the same, then we increment count. If they are different, we decrement the count. Because more than half of the values are always the same, this count will eventual end up with at least one with this wanted value.

```
Algorithm(Array A)
{
  int count = 0;
  int CurrentlyChecking;
  for (int i = 0; i < A.Len; i++)
  {
    if (count == 0)
    {
       CurrentlyChecking = A[i];
    }
    if (A[i] == CurrentlyChecking)
    {
       count++;
    }
    else
    {
       count--;
    }
   }
  return CurrentlyChecking;
}
```

□

(b) (4 points (**Extra credit**)) The Criminal Investigation Unit, while investigating a certain crime, found a set of n fingerprints of which they are convinced that more than half (i.e. $1 + \lfloor n/2 \rfloor$) belong to the same criminal, but they are not sure which ones. They hire a fingerprint expert who can compare any two fingerprints manually and tell whether these two are the same or not. However, if he has to compare all $n(n-1)/2$ pairs of finger-prints, it will take a lot of time and resources. Could you help the fingerprint expert find a strategy to find the subset of more than half identical fingerprints, where the number of comparisons is only $O(n)$? (**Hint**: Notice, there is no "total order" on the set of fingerprints, so it does not make sense to say that one fingerprint is "less" or "greater" than the other. Hence, you probably cannot use the simple solution from part (a).)

Solution: We can not have any less than or greater than comparisons for this solution. The solution listed for part (a) will also work for part (b). Just iterate through the array and keep a counter on currently checking item. If the count ever goes to 0, change the currently checking item the current item in the iteration. It is explained with more detail on part A. □

CSCI-GA.1170-001/002 Fundamental Algorithms October 8, 2014

Solutions to Problem 5 of Homework 5 (7 points)

Name: Keeyon Ebrahimi *Due: Wednesday, October 8*

(a) (5 points) Design an algorithm that takes as input an INORDER-TREE-WALK and POSTORDER-TREE-WALK of a binary tree T on n nodes (both as n-elements arrays) and outputs the PREORDER-TREE-WALK of T (again, as n-element array). Notice, T is not necessarily a binary *search* tree.

Solution: To make this work, we must know a few things. We must know that we can find the root of a tree by looking at the last item of a POSTORDER-TREE-WALK of a binary tree. We must also know that all the elements to the left of the root in the INORDER-TREE-WALK are all in the left sub-tree of the root. All the elements to the right of the root in the INORDER-TREE-WALK fall into the right sub-tree of that root.

Knowing all this, we have to Recursively Append the Root, which is the last element of the POSTORDER-TREE-WALK. We then have to append the entire left tree, which is a recursive call to our original function but with inputs that are all element that are to the left of the root in the INORDER-TREE-WALK in the same order they were in during the POSTORDER-TREE-WALK. Lastly, we then have to append the entire right tree, which is a recursive call to our original function but with inputs that are all element that are to the right of the root in the INORDER-TREE-WALK in the same order they were in during the POSTORDER-TREE-WALK.

To better explain, I will show basic pseudo code with some helper functions explained just so you can see the logic behind the solution

```
Alg(Array Post, Array InOrd)
{
    // Find Root by grabbing last item of Post
    root = Post[Post.Length-1]

    // We want an array of all the elements to the left of the root within InOrd.
    // This should keep the order of original InOrd, but only have elements to the
    // left of the root.
    ArrayLeftTreeInOrd = FindAllElementsLeftTreeWithRoot(root, InOrd);

    // We now want to modify PostOrder, keeping its order but only keeping values
    // that are within ArrayLeftTree.  We pass ArrayLeftTreeElements and the
    // original PostOrder array
    PostOrderLeftTree = PostOrderOnlyWithGivenElements(PostOrder, ArrayLeftTreeInOrd);
```

```
// Recursively call with the left Tree
ArrayPreOrderLeftTree = Alg(PostOrderLeftTree, ArrayLeftTreeInOrd);

// Do the same last few steps for right tree
// Inorder Elements only right of the root
ArrayRightTreeInOrd = FindAllElementsRightTreeWithRoot(root, InOrd);

// Post Order elements in same Post Order order,
// but only elements in ArrayRightTreeInOrd
PostOrderRightTree = PostOrderOnlyWithGivenElements(PostOrder, ArrayRightTreeInOrd);

//Recursively Call with the right tree
ArrayPreOrderRightTree = Alg(PostOrderRightTree, ArrayRightTreeInOrd);

// Array with appended root, then the pre order left tree,
// then the pre order right tree
return root + ArrayPreOrderLeftTree + ArrayPreOrderRightTree;
```

□

(b) (2 points) Now assume that the tree T is a binary *search* tree. Modify your algorithm in part (a) so that it works given only the PostOrder-Tree-Walk of T.

Solution: This is a very simple addition to our previous part answer. So now we do not have the InOrder-Tree-Walk, but we do know that our binary tree is now a sorted binary tree. Well, knowing that the tree is a sorted binary tree, we know that we can calculate the InOrder-Tree-Walk by sorting the PostOrder-Tree-Walk. Now just like before, we will have InOrder and the PostOrder Tree walk. At this point we just repeat the steps from part A. The key here is that the in an binary sorted search tree, Sorting the order of the array will also give us the InOrder Tree Walk. □