CSCI-GA.1170-001/002 Fundamental Algorithms October 7, 2014

Solutions to Problem 1 of Homework 5 (10 points)

Name: Sahil Goel *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: Number of bits(b), maximum number in this array has $= \log((\log n)^{\log n})$
$b = \log n * (\log \log n)$
We will first divide the number into (b/r) parts of r bits each and we will apply radix sort on these (b/r) parts.
Since Radix sort takes time $O(n + k)$ for array of size n and range of numbers from 0 to k and there will be (b/r) passes to radix sort as there are b/r groups of r bits each
Here $k = 2^r$
Total complexity of the program $T(n) = O(\frac{b}{r} * (n + 2^r))$

If we take $r = \log n$, Complexity of the program will be

$T(n) = O(\frac{b}{\log n} * (2n))$

Since $b = \log n * (\log \log n)$

$T(n) = O(\frac{\log n * (\log \log n)}{\log n} * (n))$

$T(n) = O(n * \log \log n)$

Therefore solution is to divide number into (b/r) i.e. $\log n * (\log \log n)/\log n = \log \log n$ parts of $\log n$ bits each and then apply radix sort on each part. □

(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: 1. First of all we create an binary search tree(having root as root) of distinct integers. This can be done while traversing the array A. Along with left and right child of the node, we will also store a count variable along with every node initially set to zero. Note: The max number of elements in tree will be $\log n$ and therefore the max height of the tree will be $\log \log n$

2. Now for every element in A, insert it into BST. Now there can be 2 cases,
(a). If the element is not present then insert it and increment the count to 1.
(b). If the element is already present then just increment the count.

Notice, for a single element the time to search and insert will be $O(\log\log n)$. For n elements total time to build binary search tree will be $O(n * \log\log n)$.

3. Now initialize an array B of size n and i=0. Traverse the tree (inorder traversal) with one minor change. For every node encountered during traversal, include this loop

```
for(int j=0;$j < node.count$;j++)
B[i++]=node.info
```

This step will take time O(n)

Hence total time complexity will be
T(n) = O(n) + $O(n * \log\log n)$
T(n) = $O(n * \log\log n)$

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms October 7, 2014

Solutions to Problem 2 of Homework 5 (6 points)

Name: Sahil Goel *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: This problem can be solved using modified version of quicksort with the following change.
Instead of returning quicksort when n=1(i.e lower = upper), return from quicksort when total number of elements become less than k. Store the indices for lower and upper index when the function returns.

After this step we will have different blocks(which may be unsorted internally) but block1 elements will be less than block 2 elements. At the end we will have indices like $(i1, j1), (i2, j2), \ldots, (i_p, j_p)$ where i is the starting position of the block and j ending position of the block.

After this step the output is $block1 < block2 < block3 < \ldots < blockp$, observe here $p > (n/k)$ since all the blocks will be of less than k size.

Complexity for this step can be calculated by using the recursion tree. The depth of the recursion tree will be $\log(n/k)$ and every step work done is O(n). Total complexity $= O(n * \log(n/k))$

Block merging part

Now we will have to merge those blocks to get blocks of size k. There will be 3 cases when we merge those blocks
1. if total length of 2 blocks is less than or equal to k, then directly merge those blocks 2. If total length of 2 blocks is greater than k
(a) Merge those arrays
(b) Find out the kth maximum element of those two blocks
(c) split the array according to kth max element (Larger goes to second block whereas smaller or equal goes to first block)

In this case complexity will be O(k) + O(k) = O(k)

Since this block merging(2 step) will run O(n/k) times , total complexity of block merging will be
O(n/k * k)= O(k)

Total complexity of the program will become, $O(n * \log(n/k))$ + O(k) which is equal to
$O(n * \log(n/k))$

Another way to view this is using the **Comparison tree**

Since at last step there will be (n/k) different blocks, which can occur in any different pattern, hence (n/k)!. And since there are k different elements in each block, we need at least k comparison to match 2 blocks and check if they are in correct order or not.

Now suppose a comparison tree has height h. The maximum number of elements at height h will be 2^h(because it is a binary tree). Also at the max depth, there will be at least (n/k)! blocks(includes every permutation for (n/k) blocks). Therefore

$2^h \geq (n/k)!$
$h \geq \log((n/k)!)$
Using stirling's approximation
$h \geq (n/k) * \log(n/k)$

Now for each block, atleast k comparisons will be needed to check if they are in correct order or not, therefore total number of comparisons will be atleast $h * k = (n/k) * \log(n/k) * k$
Minimum comparisons = $n * \log(n/k)$
Therefore comparisons = $\Omega(n * \log(n/k))$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 7, 2014

Solutions to Problem 3 of Homework 5 (6 points)

Name: Sahil Goel *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: Since we have to find the kth smallest element of two sorted lists we know that it will lie in either of the first k elements of both arrays. Hence first step is
1. newA = A[1:k] and newB = B[1:k];

Now we will use divide and conquer technique to find the kth smallest element of two sorted lists

```
ksmallest(A, B, k)
{
A = A[1:k]
B=B[1:k]
pivot = A[r];                choose r randomly between 1:k
p1=r;
p2 = binarysearch(B,pivot);              It returns the position where pivot should lie in B
if($(p1 + p2) < k$)                    kth smallest lie after p1 in A or after p2 in B
ksmallest(A[p1+1:k], B[p2+1:k], k-(p1+p2))      p1+p2 small elements neglected
else if ($(p1 + p2) > k$)              kth smallest lie before p1 in A or before p2 in B
ksmallest(A[1:p1],B[1:p2],k);
else
return max(A[p1], B[p2]);
}
```

Recurrence equation will be
$T(k) = T(x) + \log k$ $\log k$ time is used in binary search over k elements
x varies from 1 to k-1

Even if we are unlucky and pivot always divides the array in ratio 90% and 10% and kth element lies on 90% side of array then the recurrence equation will be
$T(k) = T(9k/10) + \log(k)$
$T(k) = O(\log k)$

$T(k) = 1/k * (T(1) + T(2) + \ldots + T(k-1)) + \log k$
$k * T(k) = (T(1) + T(2) + \ldots + T(k-1)) + k * \log k$
Substituting $k = k - 1$
$(k-1) * T(k-1) = (T(1) + T(2) + \ldots + T(k-2)) + (k-1) * \log k - 1$
Subtracting two equations

$k * T(k) - (k-1) * T(k-1) = T(k-1) + k * \log k - (k-1) * \log k - 1$
For very large k, $\log(k-1) = \log k$
$k * T(k) = k * T(k-1) + \log k$
$T(k) = O(\log k)$

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms October 7, 2014

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

Name: Sahil Goel *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: Since majority element exists atleast $1 + \lfloor n/2 \rfloor$ times, therefore in sorted array the majority element will lie at [n/2] position for sure. Lets analyze this

1. If first element (i.e the lowest) is the majority element, then in sorted array B, the majority element will lie from [1:x] where $x \geq 1 + \lfloor n/2 \rfloor$
2. If the last element (i.e the highest) is the majority element, then in sorted array B, the majority element will lie from [x:n] where $x \leq n/2 - 1$
3. If the majority element exists in middle somewhere, since it has more than n/2 elements, it will definitely be at n/2th position in sorted array.

So our problem is reduced to find (n/2)th maximum element in an array which we can find in O(n) time using divide and conquer technique.
Here is the procedure

```
RandSelect(A,p,r,i)
{
if p=r then return A[p]
q = Rand-Partition(A,p,r)        Rand-Partition partitions the array according to randomly chosen pivot and returns its position
k=q - p + 1        Overall rank of pivot
if i=k then return A[q]
if $i < k$ then return RandSelect(A,p,q-1,i)
else return RandSelect(A,q+1,r,i-k)
}
```

We can find the majority element by calling (A,p,r,n/2)

□

(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't use the solution from last part since we can't order two fingerprints. Lets call all the majority fingerprints as X and all other fingerprints as Y. We know that number of X > number of Y. We will need two variables :
1. CurrentFingerprint: cf
2. Count

Initially we will set cf = NULL and Count =0
Now we will traverse through all the fingerprints one by one and only match it to the cf

```
for i=1:n
    if(fingerprint[i] = cf)
        count=count+1
    else if(count = 0)
        cf = fingerprint[i]
        count=1
    else
        count = count -1;
end for;
finalfingerprint=cf;
```

In this case, we can observe that if there exists a majority element it will be stored in cf finally as there will be more increments for it than decrements because X > Y. Therefore after traversing one time i.e. n comparisons, we will find the majority fingerprint.
Now we will again traverse the array and match it to finalfingerprint. Whichever it matches to will be the culprit's fingerprint.

finalcollection stores all the majority fingerprints

```
finalcollection = NULL
For j=1:n
    if(fingerprint[j]=finalfingerprint)
        append fingerprint[j] to finalcollection
end For
```

this solution will make 2n comparisons and hence final complexity will be O(n)

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 7, 2014

Solutions to Problem 5 of Homework 5 (7 points)

Name: Sahil Goel *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:

In Preorder, the elements are accessed while traversing the tree in order Top - Left - Right
Therefore recursive procedure will have the structure like
1. Print
2. Recursive call on left child
3. Recursive call on right child

The last element in post order tree is always the root
Left subtree in inorder array lies on left side of root
Right Subtree in inorder array lies on right side of root
Using a static global variable k = 0;

Using two procedures, one is search to search for an element in an array and return its position

```
Search(arr[ ], x)
{
for i=1:length(arr)
if(arr[i]=x)
return i;
else
return -1
}
```

```
printPreOrder(in[ ], post[ ], int n)
{
% To find root position in inorder array
root = search(in, post[n]);
% post[n] represents the element at root position

print(post[n]);             Printing the root
pre[k++]= post[n];             Storing the root
if(root!=1)          Left subtree exists
printPreOrder(in[1:root], post[1:root], root )       Call for left subtree recursively

if(root!=n)          Right subtree exists
printPreOrder(in[root+1:n], post[root+1:n],n-root-1)       Call for right subtree recursively

}
```

□

(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: In BST, left subtree is always smaller than the Root and Right Subtree is always larger than the root. Also we know that root is the last element of postorder. In postorder array the structure is like [L R T], as soon as we know the root i.e. T, we can search for L and R as we know the last element of L will be less than T and first element of R will be greater than T.

In this case we will modify search procedure, which will search for the last element in post order tree which is less than the root at current state.

```
NewSearch(arr[ ], x)
{
for i=1:length(arr)
if($arr[i] > x$)
return i-1;
else
return -1
}

printPreOrder(post[ ], int n)
{
% To find index that divides left and right tree in post order tree
root = search(in, post[n]);
% post[n] represents the element at root position


print(post[n]);              Printing the root
pre[k++]= post[n];              Storing the root

if($root \geq 1$)          Left subtree exists
printPreOrder(post[1:root], root )       Call for left subtree recursively
if(root!=n)          Right subtree exists
printPreOrder(post[root+1:n],n-root-1)       Call for right subtree recursively
}
```

$\square$