ALGORITHMS - CSE 202 - SPRING 2000
Divide and Conquer and Sorting, Some Solutions

Name Tree Isomorphism

Problem Give an efficient algorithm to decide , given two rooted binary trees T and T', whether T and T' are isomorphic as rooted trees.

Solution

Let $root$ be the root of T and $root'$ be the root of T', and let L, R, L', R' be the left and right subtrees of T and T' respectively. If T embeds in T', then the isomorphism f must map $root$ to $root'$, and must map every member of L either entirely to L' or entirely to R' and vice versa. Thus, we can define the isomorphism relationship Iso on sub-trees as follows: $Iso(NIL, NIL) = True$, $Iso(x, NIL) = Iso(NIL, x) = False$ if $x \neq NIL$ and otherwise

$$Iso(r_1, r_2) =$$

$$(Iso(Left(r_1), Left(r_2))ANDIso(Right(r_1), Right(r_2)))$$

OR

$$(Iso(Left(r_1), Right(r_2))ANDIso(Right(r_1), Right(r_2)).$$

The above immediately translates into a program, and the time, I claim, is $O(n^2)$. The reason is that for any nodes $x \in T, y \in T'$, the above calls itself on x, y at most once (in fact, precisely when x and y are at the same depths of their trees.) Let P be the path from $Root$ to x and P' be that from $Root'$ to y; in a tree there is exactly one' such path. Then, of the four recursive calls at any level, at most one involves nodes from both P and P'. Inductively, at any recursion depth, at most one call compares a sub-tree containing x to one containing y. Since all sub-trees called at depth d of the recursion are depth d in their respective trees, the claim follows. Therefore the total number of calls is at most $O(n^2)$.

Name $i-$th Largest Element

Problem Show that, for any constant i, the i'th largest element of an array can be found with $n + O(\log n)$ comparisons.

Hint Consider $i = 2$ first. Exercise 10-3 on p. 194 is a generalized version; if you can decrypt it, it might be helpful.

Solution

First, note that in any comparison algorithm to find the i'th largest in the array, we must compare the i'th largest element to the $i + 1$'st largest. (Otherwise, the algorithm would give the same answer if we increased the array position where the $i+1$'st element is to be between the i'th and $i - 1$'st largest.)

We'll use this to prove by induction on i that there is a comparison algorithm and constants c_i, d_i so that the algorithm makes a total of at most $n + c_i \log n$ comparisons and no element is compared to more than $d_i \log n$ other elements.

First, the base case is when $i = 1$. We claim we can find the largest element while making $n-1$ comparisons, without comparing any element to more than $\log n$ elements, so the claim is true for $c_1 = 0, d_1 = 1$. We do this by finding the maximum through a divide-and-conquer approach, recursively finding the maximum of the first and last $n/2$ elements, and returning the maximum. The number of comparisons is easily seen to be $n - 1$ by solving the recurrence, or by observing that every element but the largest is the smaller element in exactly one comparison the algorithm makes. No element is compared to more than $\log n$ others, since there are $\log n$ recursion levels, and each element is only in one sub-array at any recursion level.

Assume we have an algorithm that finds the i'th largest element in $n + c_i \log n$ comparisons, making no more than $d_i \log n$ comparisons with any particular element. To find the $i + 1$'st largest, we run the above algorithm, and then look at all elements compared to the i'th largest, and found to be smaller. There are at most $d_i \log n$ such elements. As observed above, the $i + 1$'st largest element must be in this set, and it must be the largest in this set, since all elements are less than the i'th largest. So we find the maximum using the approach in the base case, and output it. This uses at most $d_i \log n$ additional comparisons, so we can let $c_{i+1} = c_i + d_i$. No one element is compared more than $\log(d_i \log n) < \log n$ additional times, so we can let $d_{i+1} = d_i + 1$.

Thus, by induction, for every i there is a comparison algorithm that finds the i'th largest in $n + O(\log n)$ comparisons.

Name Mode

Problem

The mode of a set of numbers is the number that occurs most frequently in the set. The set (4,6,2,4,3,1) has a mode of 4.

1. Give an efficient and correct algorithm to comput the mode of a set of n numbers.
2. Suppose we know that there is an (unknown) element that occurs $n/2 + 1$ times in the set. Give a worst-case linear-time algorithm to find the mode. For partial credit, your algorithm may run in expected linear time.

Solution

1. A simple algorithm for this problem is to sort the n numbers, and then traverse the sorted array keeping track of the number with the highest frequency.

   This algorithm's worst-case running time is $\Theta(n \lg n)$ due to the MERGESORT being performed in step 1. The algorithm works because after the set is sorted identical numbers will be adjacent. Therefore, with a linear scan of the array we can find the number with the highest frequency.

2. Solution 1: If an element occurs more than $n/2$ times, then that element will necessarily be the median element. So, the $Select$ algorithm can be used to find that element in deterministic linear time. However, this algorithm has a high constant and so more direct divide and conquer approaches may be useful.

3. Solution 2:

   A $\Theta(n)$ expected-time algorithm uses the partition procedure from quicksort. A randomly chosen partition element will be used to split the array into two halves. If the partition element is less than the mode (the element which occurs $n/2 + 1$ times), then the partition element will end up in a position in the range $(1 \ldots (n/2) - 2)$. In other words, if the partition element ends up in a position to the left of the center, the mode will be in the set of numbers to the right of the partition element. If the partition element is greater than the mode the partition element will be in position $n/2 + 2$ or greater, and the mode will be in the set of numbers to the left of the partition element. Therefore, after each partition we can recursively call our algorithm on the set of numbers which includes the mode. When the size of the set is 1 then that element is our answer. The running time in the average-case is $T(n) = T(n/2) + n = \Theta(n)$.

   Inputs: array of numbers A in which one number appears $n/2 + 1$ times, starting index low, ending index $high$.

   Outputs: the number within the set with the highest frequency.

   FINDMODE(A,low,$high$)

   1. if ($low = high$)
   2. return $A[low]$

3. if ($low < high$)
4. $ploc =$ PARTITION(A,low,$high$)
5. if $ploc > \lfloor n/2 \rfloor + 1$
6. FINDMODE(A,low,$ploc - 1$)
7. else
8. FINDMODE(A,$ploc + 1$,$high$)

Solution 3:

We can use a more exact divide and conquer scheme to get a worst-case running time of $\Theta(n)$. Let f denote the most frequently occurring element in the list. We will divide the set by doing $\lfloor n/2 \rfloor$ pairwise comparisons and only keeping one representative of the pairs which are equal. We will show that as we cut down the list, we maintain the invariant that the frequency of f in the current list is strictly larger than the number of remaining elements in the list. This invariant is true for the original list. If n is even, there are more equalities involving f than all other equalities combined. Otherwise, the number of other elements ($\neq f$) will be at least as much as the frequency of f since each discarded pair contains at least one element other than f. If n is odd, we will still pair the elements and take one element from each pair that reports equality. Let m be number of pairs that report equality. If m is of the form $2k + 1$ for some integer $k \geq 0$, then we discard the last remaining element. Of the $2k + 1$ elements, f must appear at least $k+1$ times. Otherwise, the number of elements not equal to f would be more than the frequency of f in the list prior to pairing since each pair that was discarded contained at least one element that is not f. If m is of the form $2k$ for some integer $k \geq 0$, then keep the last element along with the $2k$ elements and recurse. Consider two cases to see that the invariant is maintained. If the last element is not equal to f, then of the $2k$ elements f must appear at least $k + 2$ times and thus including the last element does not violate the invariant. If the last element is f, f must appear at least k times among $m = 2k$ elements. Otherwise, f would not be the most frequently occurring element prior to pairing.

Inputs: set of numbers S in which one number appears $n/2 + 1$ times, the size of the set n

Outputs: the number within the set with the highest frequency.

FINDMODE(S, n)

1. if $n = 1$ then return $S[1]$
2. $S_{new} = \emptyset$
3. $n_{new} = 0$
4. for $i = 1$ to $\lfloor n/2 \rfloor$ by 2
5. if ($S[i] = S[i + 1]$) then
6. $n_{new} = n_{new} + 1$
7. $S_{new} = S_{new} \bigcup S[i]$
8. if (n mod 2) = 1 // n is odd
9. if (count mod 2) = 1
10. $S_{new} = S_{new} \bigcup S[n]$
11. $n_{new} = n_{new} + 1$

12. return FINDMODE(S_{new},n_{new})

Each time FINDMODE is recursively called with a smaller problem which is at most size $\lceil n/2 \rceil$. At each step in the recursion $\lfloor n/2 \rfloor$ comparisons are performed. Therefore, the total running time of the algorithm is

$$T(n) \leq T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor$$

The solution to this recurrence is $T(n) = O(n)$ and can be computed by using the substitution method with a starting guess that $T(n) \leq cn$.

Name k-way Merging

Problem Give tight upper and lower bounds in the comparison model for merging k sorted lists with a total of n elements. Lower bounds should be for *any* algorithm, not just the upper bound algorithm.

Solution

An $O(n \log k)$ algorithm for the problem is to pair the lists umerge each pair, and recursively merege the $k/2$ resulting lists. The time is given by the recurrence $T(n,k) = O(n) + T(n,k/2)$, since the time to perform the merge for each pair of lists is proportional to the sum of the two sizes, so a constant amount of work is done per element. Then unwinding the recurrence gives $\log k$ levels each with $O(n)$ comparisons, for a total time of $O(n \log k)$.

One way to prove the lower bound for any algorithm is to use our lower bound for sorting of $n \log n - O(n)$ and the upper bound for mergesort of $n \log n$ comparisons to reduce the question of sorting n unordered elements to that of merging k sorted lists of total size n as follows.

Let A be a correct algorithm for merging k lists that runs in time $T(n,k)$. We'll prove $T(n,k) \geq n \log k - O(n)$ as follows. Let $A - Sort$ be the algorithm that takes an array of n elements and divides it into k sub-arrays of at most n/k elements each and at most k leftover elements. It sorts each sub-array using mergesort , which takes at most k times $n/k \log(n/k)$. It then places each left-over element within an array, making less than $k(n/k) = n$ comparisons. It calls A on the resulting set of sorted sub-arrays, resulting in a sorted list.

The total number of comparisons for $A - Sort$ is at most $n \log(n/k) + n + T(n,k)$. Since any algorithm for sorting takes at least $n \log n - O(n)$ comparisons, $T(n,k) + n \log(n/k) + n \geq n \log n - O(n)$, so $T(n,k) \geq n \log n - n \log(n/k) - O(n) = n \log n - n \log n + n \log k - O(n) = n \log k - O(n)$.

An alternative method for proving a lower bound is to use a counting argument similar to that used to prove $\Omega(n/lgn)$ for single list sorting. In order for an algorithm to succesfully sort all k-lists of a total of n elements, it must in particular distinguish all partitionings of $1..n$ into k sorted lists of n/k elements each. There are at least $N_{n,k} = (k!)^{n/k}$ such lists. Which is $N_{n,k} > (k/2)^{(k/2)(n/k)}$, and therefore $\lg N_{n,k} > cn \lg k$ or $\Omega(n \lg k)$.