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 $O(n^2)$.

PROBLEM 1 (8-3 IN CLR): Stooge sort

Solution:

a) We will prove by induction that Stooge sort correctly sorts the input array $A[1 \ldots n]$.

The base cases are for $n = 1$ and $n = 2$. If $n = 1$ then $i = j$, and thus $i + 1 > j$; then, line 4 returns $A[i]$ (which is correct). If $n = 2$ then lines 1 and 2 guarantee that a bigger element will be returned in a position following that of a smaller element. Thus, Stooge sort correctly sorts A when $n = 1$ and when $n = 2$.

The inductive case happens when $n \geq 3$. The inductive hypothesis is that Stooge sort correctly sorts arrays of length $< n$. We must show that Stooge sort correctly sorts arrays of length n.

The input array A is essentially split into three pieces: the first third and the last third are both of length k, and the second third is of length $n - 2k$, where $k = \lfloor n/3 \rfloor$. Note 1: $j - i + 1 = n$ in the first or "top" call to Stooge sort. Note 2: Observe that the middle third is at least as long as each of the other thirds (since $k = \lfloor n/3 \rfloor \leq n/3$, but $n - 2k = n - 2\lfloor n/3 \rfloor \geq n - 2(n/3) = n/3$; this is actually necessary for the algorithm to work).

In line 6, Stooge sort is applied to the first two-thirds of A (i.e., the subarray of length $n - k$ consisting of the first and middle thirds of A). We are assuming that $n \geq 3$, so $k = \lfloor n/3 \rfloor \geq 1$ and $n - k$ is strictly less than n. This means that we can invoke the inductive hypothesis on this subarray (i.e., we assume by inductive hypothesis that the first two-thirds of A will be sorted in the first two-thirds of the new array A').

After line 6, the largest k elements of A will be in the last $n - k$ positions of A' (i.e., they will be in the middle third and/or the last third of A'). Here's why: Let x be any one of the k largest elements in A. Case 1: If x is in the last third of A, then it will be in the last third of A', and our claim is rather trivially true. Case 2: Assume x is in the first two-thirds of A. Since x is one of the k largest elements in A, at least $n - k$ other elements in A are "smaller" (actually, $\leq$) than x. At most, k of these $n - k$ elements can be in the last third of A. Therefore, at least $n - 2k$ of these elements are in the first two-thirds of A. We can conclude that there are at least $n - 2k$ elements in the first two-thirds of A that are smaller than x. But $n - 2k = n - 2\lfloor n/3 \rfloor \geq n - 2(n/3) = n/3 \geq \lfloor n/3 \rfloor = k$. Because $k \leq n - 2k$, the first k positions of A' will consist of elements that are all smaller than x. That means that x will be in the middle third of A'. In either case, x will be in the last two-thirds of A'.

In line 7, Stooge sort is applied to the last two-thirds of A' (i.e., the subarray of length $n - k$ consisting of the middle and last thirds of A'). We are assuming that $n \geq 3$, so $k = \lfloor n/3 \rfloor \geq 1$ and $n - k$ is strictly less than n. This means that we can invoke the inductive hypothesis on this subarray (i.e., we assume by inductive hypothesis that the last two-thirds of A' will be sorted in the last two-thirds of the new array A'').

Recall that after line 6, the largest k elements of A will be in the last two-thirds (i.e., the last $n - k$ positions) of A'. After line 7, the largest k elements of A (or A') will make up the last third of A'', and they will be in sorted order.

In line 8, Stooge sort is applied to the first two-thirds of A'' (i.e., the subarray of length $n - k$ consisting of the first and middle thirds of A''). We are assuming that $n \geq 3$, so $k = \lfloor n/3 \rfloor \geq 1$ and $n - k$ is strictly less than n. This means that we can invoke the inductive hypothesis on this subarray (i.e., we assume by inductive hypothesis that the first two-thirds of A'' will be sorted in the first two-thirds of the new array B).

Recall that after line 7, the largest k elements of A will be in the last third of A'' in sorted order. After line 8, the other elements (i.e., the $n - k$ smallest elements) will make up the first two-thirds of B, and they will be in sorted order; also, the last third of B will be the same as the last third of A''. Thus, B, which is the final array returned by the algorithm, will consist of all the elements of A in sorted order.

b,c) The recurrence is $T(n) = 3T(2n/3) + O(1)$, since we split arrays into three smaller arrays, each of size $\approx$ 2/3 the size of the big array (don't worry about floor and ceiling operators here); the nonrecursive portion of the code is $O(1)$ (i.e., the complexity is bounded by a constant). Its solution is, by Case 1 of the Master Theorem (with $a = 3$, $b = 3/2$, and $\epsilon = \log_{3/2} 3$, say), $T(n)\ \epsilon\ \Theta(n^{\log_{3/2} 3})$. Now, to compute $\log_{3/2} 3$, observe that it is equal to $\frac{\log_e 3}{\log_e (3/2)} (\approx 2.71)$. Thus, since $2.71 > 2.7$, we can say that $T(n)\ \epsilon\ \Omega(n^{2.7})$. The worst-case running time of Stooge sort is worse than all worst-case running times of insertion sort $(\Theta(n^2))$, merge sort $(\Theta(n \lg n))$, quicksort $(\Theta(n^2))$ and heapsort $(\Theta(n \lg n))$. Begone, Stooges!

Note: You could also look at the corresponding recursion tree, which will have depth $\approx \log_{3/2} n$ (remember that $n = 1$ and $n = 2$ are both base cases). For simplicity, let's say that the depth of all leaves in the recursion tree is $\log_{3/2} n$. Then, when we add up the number of subproblems at each recursive level, we get:

$$3^0 + 3^1 + ... + 3^{\log_{3/2} n} = \frac{1 - 3^{(\log_{3/2} n) + 1}}{1 - 3} = \frac{3^{(\log_{3/2} n) + 1} - 1}{2}$$

(Think: $\frac{\text{"first in" - "first out"}}{1 - \text{common base}}$.)

For simplicity, let's consider $3^{\log_{3/2} n}$. By Logarithm Property 6 on p.443 in Appendix A.5, $3^{\log_{3/2} n} = n^{\log_{3/2} 3}$ (look familiar?). Remember that $O(1)$ work is being done at each node in the recursion tree.

PROBLEM N. 1: Stooge sort (problem 8-3, pag. 169 of CLR, questions a,b,c).

Solution:

a) We will prove by induction that Stooge sort correctly sorts the input array $A[1 \ldots n]$. The base case is for $n = 2$. In this case the first two lines of Stooge sort guarantee that a bigger element will be returned in a position following that of a smaller element. Thus, Stooge sort correctly sorts A when $n = 2$. The inductive case happens when $n \geq 3$. Then, assume by inductive hypothesis that the recursive call to Stooge-sort in line 6 correctly sorts the elements in the first two-thirds of the input array A, and let A' be the array thus obtained. Also, assume by inductive hypothesis that the recursive call to Stooge-sort in line 7 correctly sorts the elements in the last two-thirds of the array A', and let A'' be the array thus obtained. Finally, assume by inductive hypothesis that the recursive call to Stooge-sort in line 8 correctly sorts the elements in the first two-thirds of the array A'', and let B be the (final) array thus obtained. Now we need to prove that at the end of the algorithm, the array B is sorted. To prove this, we just need to prove that for any two elements $A[i] < A[j]$, the position of $A[i]$ in B will be smaller than the position of $A[j]$, no matter which is their position in the input array A. Since this will be true for any pair $A[i], A[j]$, then the correctness of Stooge sort will follow.

We divide the proof in two cases. 1) Consider the simple case in which the position of $A[i]$ is smaller than the position of $A[j]$ in the input array A. Then, clearly, from the three facts that we have estabilished using the inductive hypothesis, we have that the position of $A[i]$ in B will be smaller than the position of $A[j]$. 2) Consider the more involved case in which the position of $A[j]$ is smaller than the position of $A[i]$ in the input array A. We need to prove that their order in B will be swapped. Here we have some other cases. First, if $A[j]$ and $A[i]$ are both in the first two-thirds of the input array A, then they will be swapped in the recursive call in line 6 (which is correct by inductive hypothesis) and will not be swapped again in the following recursive calls (again, by the inductive hypothesis). Second, if $A[j]$ and $A[i]$ are both in the last two-thirds of the input array A, then they will be swapped in the recursive call in line 7 (which is correct by inductive hypothesis) and will not be swapped again in the following recursive call (again, by the inductive hypothesis). Finally, if $A[j]$ is in the first one-third of the input array A and $A[i]$ is in the last one-third of the input array A, then we claim that their position are swapped either in the recursive call in line 7 or in the recursive call in line 8. In fact, assume their positions are not swapped in the recursive call in line 7. This means that after the execution of the recursive call in line 6 (which is correct by inductive hypothesis) the position of $A[j]$ in array A' is still in the first one-third and moreover, at least all the elements in the second third of A' are greater than $A[j]$. Then we have that since these elements are greater than $a[j]$, then they are greater than $A[i]$ (since we assumed $A[i] < A[j]$), and after the execution of the recursive call in line 7 (which is correct by inductive hypothesis), $A[j]$ will appear in the first two-thirds of A''. Now, since also $A[i]$ will remain in the first third of A'', their positions will be swapped by the execution of the recursive call in step 8.

b,c) The recurrence is clearly $T(n) = 3T(2n/3) + O(1)$. Its solution is, by case 1 of Master Theorem, $T(n) = \Theta(n^{\log_{3/2} 3}) = \Omega(n^{2.7})$. Clearly, the worst-case running time of Stooge-sort is worse than all worst-case running times of insertion sort, merge sort, quicksort and heapsort.

Name MAX SUM

Problem Use the divide-and-conquer approach to write an efficient recursive algorithm that finds the maximum sum in any contiguous sublist of a given list of n real values. Analyse your algorithm, and show the results in order notation. Can you do better?

Solution(s)

Solution 1 ($O(n\lg n)$):

Informally, one solution is the following. Divide the given list into two sublists (call them "left" and "right") of roughly 1/2 the size of the original list, and recursively "solve" each sublist by finding the maximum "contiguous sum" in both. How do we then solve our original list? Take the largest of:

(1) the maximum "contiguous sum" in the "left" sublist

(2) the maximum "contiguous sum" in the "right" sublist

(3) the maximum "contiguous sum" that "spreads" from the point at which we split the list into two sublists (this sum may or may not include values from both sublists). To compute this, we will need to compute:

(3a) the maximum "contiguous sum" whose corresponding values make up the last l positions of the "left" sublist (for some l)

(3b) the maximum "contiguous sum" whose corresponding values make up the first r positions of the "right" sublist (for some r)

We then want the maximum of (3a), (3b), and (3a)+(3b) as our answer to (3).
Observe that we could just compare (3a)+(3b) with (1) and (2), because if (3a) is the maximum "contiguous sum" in the big list, then (3a) must equal (1); the same goes for (3b) and (2).

Here is the algorithm:

Inputs: A, an array of real values; $Left$ and $Right$, the leftmost and rightmost position numbers of A in the subarray under consideration.

Outputs: the maximum "contiguous sum" in A.

procedure MAXSUM (A , $Left, Right$);
var
$Center$: integer;
The position (in A) of the last element in the left subarray.
$LeftSum, RightSum$: real;
The answers to (1) and (2)
$LeftBorderSum, RightBorderSum$: real;
# The answers to (3a) and (3b)
begin
if $Left = Right$ **then**
return $A[Left]$
The base case is $n = 1$ (i.e., when $Left = Right$).
We return the sole array entry

$Center := \lfloor (Left + Right)/2 \rfloor$
$LeftSum$:= MAXSUM($A, Left, Center$)
$RightSum$:= MAXSUM($A, Center + 1, Right$)
compute $LeftBorderSum$
compute $RightBorderSum$
Code omitted. Both these computations are $O(n)$.
return max($LeftSum, RightSum, LeftBorderSum + RightBorderSum$)
We want the maximum of (1), (2), and (3a)+(3b).
end;

The recurrence is $T(n) = 2T(n/2) + O(n)$. We split arrays into two smaller arrays, each of size $\approx$ 1/2 the size of the big array. The nonrecursive portion of the code is $O(n)$; in fact, it is $\Theta(n)$. Its solution is, by Case 2 of the Master Theorem (with $a = 2$ and $b = 2$ and thus $n^{\log_b a} = n^{\log_2 2} = n^1 = n$), $T(n)\ \epsilon\ \Theta(n \lg n)$.

Note: You could also look at the corresponding recursion tree, which will have depth $\approx \log_2 n$, or $\lg n$. At each level in the recursion tree, $O(n)$ work is being done. So, the algorithm is $O(n \lg n)$.

Solution 2 ($O(n)$):

There is a way to reduce the $O(n)$ "overhead" in the previous solution to $O(1)$. Instead of having the algorithm compute the "border sums" in $\Theta(n)$ time in each call, let's have the algorithm "paste together" information provided by results from the recursive calls. Consider the following algorithm:

procedure MAXSUM2 ($A, Left, Right$, **var** $LeftEndSum$, **var** $RightEndSum$, **var** $AllSum$)
var
$Center$: integer;
$LeftSum, RightSum$: real;
$LeftEndSum, RightEndSum$: real;
The maximum "contiguous sums" starting from the left end
(respectively, ending at the right end) of the array
$LeftBorderSum, RightBorderSum$: real;
The answers to (3a) and (3b)
$AllLeftSum, AllRightSum$: real;
The total sum of all elements in the left (respectively right) subarrays.
$AllSum$: real;
# This is the sum of all the elements in the array.
begin
if $Left = Right$ **then**
begin
$LeftEndSum := A[Left]$
$RightEndSum := A[Left]$
$AllSum := A[Left]$
return $A[Left]$
end
$Center := \lfloor (Left + Right)/2 \rfloor$
$LeftSum$:= MAXSUM2($A, Left, Center, LeftEndSum, LeftBorderSum, AllLeftSum$)

$RightSum$:= MAXSUM2($A, Center+1, Right, RightBorderSum, RightEndSum, AllRightSum$)
$AllSum := AllLeftSum + AllRightSum$
$LeftEndSum := \mathbf{max}(LeftEndSum, AllLeftSum + RightBorderSum)$
$RightEndSum := \mathbf{max}(RightEndSum, LeftBorderSum + AllRightSum)$
return max($LeftSum, RightSum, LeftBorderSum + RightBorderSum$**)**
end;

The recurrence is $T(n) = 2T(n/2) + O(1)$, since we split arrays into two smaller arrays, each roughly half the size of the larger array. The nonrecursive portion of the code is $O(1)$ (i.e., the complexity is bounded by a constant). Its solution is, by Case 1 of the Master Theorem (with $a = 2$, $b = 2$, $\log_b a = \log_2 2 = 1$ and $\epsilon = \log_2 2 = 1$, say), $T(n)\ \epsilon\ \Theta(n)$.

Note: You could also look at the corresponding recursion tree, which will have depth $\approx \log_2 n$. For simplicity, let's say that the depth of all leaves in the recursion tree is $\log_2 n$. Then, when we add up the number of subproblems at each recursive level, we get:

$$2^0 + 2^1 + ... + 2^{\log_2 n} = \frac{1-2^{(\log_2 n)+1}}{1-2} = 2^{(\log_2 n)+1} - 1 = 2n - 1$$

(Think: $\frac{\text{“first in” - “first out”}}{1-\text{common base}}$.)

Remember that $\Theta(1)$ work is being done at each node in the recursion tree.

Solution 1 (Multi-pass $\Theta(n)$ algorithm):

Let's say the array (or list) "A" is indexed from 1 to n. Set up an array L indexed from 0 to n. Let $L[i]$ be the "maximum contiguous sum" whose corresponding subarray ends with $A[i]$, the element of the A array in the ith position ($1 \leq i \leq n$). Let $L[0] = 0$, since 0 is a viable answer if all of the $A[i]$ values are nonpositive or if the A array has no elements (a weird case). Our "maximum contiguous sum" in A is thus $\max_{0 \leq i \leq n} L[i]$; this is because the true "maximum contiguous sum" is $L[0] = 0$, or it corresponds to a subarray that terminates at some position i in the array ($1 \leq i \leq n$). (There may, incidentally, be more than one subarray that corresponds to the sum.)

Here's the algorithm:

Inputs: n, array A indexed from 1 to n.
Outputs: $MaxSum$, the "maximum contiguous sum" in A.

```
procedure MAX.CONTIGUOUS.SUM (n, A);
var
   L : array [0..n];
begin
   L[0] := 0
   for i = 1 to n do
      if L[i − 1] < 0 then
         L[i] := A[i]
               # If the "maximum contiguous sum" ending at array position i − 1
               # is negative (let's call the corresponding subarray B), then we would
               # rather take A[i] as our "maximum contiguous sum" ending at position i
               # instead of attaching A[i] to B, since B just "drags A[i] down."
      else
         L[i] := L[i − 1] + A[i]
               # If the "maximum contiguous sum" ending at array position i − 1
               # is nonnegative, then we want to attach A[i] to the corresponding
               # subarray and take the sum.
      end
   end
   MaxSum:=max(L)
         # Here, max takes the maximum value in L. This operation is Θ(n).
   return MaxSum
end;
```

This iterative algorithm is $\Theta(n)$, since the **for** loop and the **max** function are both $\Theta(n)$.

Solution 2 (One-pass $\Theta(n)$ algorithm):

Set up a variable $MaxSum$ that will keep track of $\max_{0 \leq k \leq i} L[k]$ as i increases from 0 to n. Then, the second pass through the L array to find the maximum value becomes unnecessary.

In fact, we don't even need the L array! We can replace it with another variable, $CurrentSum$. We don't need to maintain the L array because (1) $MaxSum$ is already keeping track of the running maximum of the $L[i]$ quantities, and (2) $CurrentSum$ con-

tains all the information from previous array elements that is needed to update $MaxSum$. In particular, $CurrentSum$ keeps track of max(0,$L[i]$) as i increases from 0 to n.

Here's the algorithm:

Inputs: n, array A indexed from 1 to n.
Outputs: $MaxSum$, the "maximum contiguous sum" in A.

```
procedure MAX.CONTIGUOUS.SUM2 (n, A);
var
   CurrentSum, MaxSum : real;
begin
   CurrentSum := 0
   MaxSum := 0
   for i = 1 to n do
      CurrentSum := CurrentSum + A[i]
            # This is the sum we would get if we were to attach A[i] to the
            # "maximum contiguous subarray" B ending at position i - 1
            # if the sum of B's elements is nonnegative. The initialization
            # of CurrentSum and the upcoming if statement guarantee that,
            # if the B sum is negative, it will not be added to A[i], and we
            # will take A[i] as our CurrentSum. In short, CurrentSum is
            # assigned max(0, L[i - 1]) + A[i], which was how we computed
            # successive values of L[i] in the first algorithm.
      if CurrentSum < 0 then
         CurrentSum := 0
               # This guarantees that [in the next iteration], array position i + 1 will start
               # off a new subarray under consideration. Since the "maximum contiguous
               # sum" ending at position i is negative, we don't want to add it in next
               # time.
      else if CurrentSum > MaxSum then
         MaxSum := CurrentSum
               # Update MaxSum as necessary.
      end
   end
   return MaxSum
end;
```

This iterative algorithm is $\Theta(n)$, since the **for** loop is $\Theta(n)$.

Name Weighted Median

Problem

For n distinct elements $x_1, x_2, \cdots, x_n$ with positive weights $w_1, w_2, \cdots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the weighted median is the element x_k satisfying

$$\sum_{x_i < x_k} w_i \leq 1/2 \text{and} \sum_{x_i > x_k} w_i \leq 1/2$$

1. Argue that the median of $x_1, x_2, \cdots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \cdots, n$.

2. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

3. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm.

Solution

1.

2.

3. We are given an unsorted list $x_1, ...x_n$ and positive weights $w_1, ..w_n$ with $\sum w_i = 1$, and wish to find an x_i so that $\sum_{j|x_j<x_i} w_j \leq 1/2$ and $\sum_{k|x_k>x_i} w_k \leq 1/2$.

   For a weighted unsorted list $(x_1, w_1), ..(x_n, w_n)$, let $W = \sum_{1 \leq i \leq n} w_i$. We will give an algorithm which solves a more general problem: given a weighted array, and a number $0 \leq \alpha \leq W$ find an x_i so that $\sum_{j|x_j<x_i} w_j \leq \alpha$. and $\sum_{k|x_k>x_i} w_k \leq W - \alpha$.

   We know there is an algorithm $Select$ which can find the j'th largest element of a list of length n in $O(n)$ time. To find the weighted median (or in general, the α-sample), we first find the non-weighted median x_l using Select. Then we divide the list into the subset S of elements smaller than x_l and the subset B of elements bigger than or equal to x_l. Since x_l is the unweighted median, both S and B have at most $n/2$ elements. (If our list contains only a single element, the recursion bottoms out, and we return that element, which is a correct solution since both sums in the two inequalities are over empty sets, hence are 0, and $0 \leq \alpha \leq W$.)

   We then calculate $W_S = \sum_{x_j \in S} w_j$. If $W_S > \alpha$, we recursively call our algorithm on S, α. We output the result of this recursive call, x_k, since x_k satisfies: $\alpha \geq \sum_{j|x_j \in S, x_j < x_k} w_j = sum_{j|x_j<x_k} w_j$, since only elements of S can be smaller than an element of S, and $\sum_{j|x_j>x_k} w_j = \sum_{j|x_j \notin S} w_j + \sum_{j|x_j \in S, x_j > x_k} w_j \leq (W - W_S) + (W_S - \alpha) = W - \alpha$ since all elements not in S are larger than x_k. Otherwise, recursively call the algorithm on $B, \alpha - W_S$. The proof of correctness in this case is similar to the last. Thus, the above algorithm always finds a valid solution.

   Since the time used by the sub-routine select is $O(n)$ as is that to divide the list into B and S and to compute W_S, there is a constant c so that the time taken

by the algorithm on an n element input, $T(n)$, is at most cn + the time taken by a recursive call on an input of 1/2 the size. I.e., $T(n) \leq cn + T(n/2)$. Thus $T(n) \leq cn + cn/2 + cn/4 + ... \leq 2cn$, so the algorithm takes $O(n)$ time.