For groups of 77, the algorithm still works in linear time. The number of elements greater than xx(and similarly, the number less than xx) is at least

4(⌈12⌈n7⌉⌉−2)≥2n7−8,4(⌈12⌈n7⌉⌉−2)≥2n7−8,

and the recurrence becomes

T(n)≤T(⌈n/7⌉)+T(5n/7+8)+O(n),T(n)≤T(⌈n/7⌉)+T(5n/7+8)+O(n),

which can be shown to be O(n)O(n) by substitution, as for the groups of 55 case in the text.

For groups of 33, however, the algorithm no longer works in linear time. The number of elements greater than xx, and the number of elements less than xx, is at least

2(⌈12⌈n3⌉⌉−2)≥n3−4,2(⌈12⌈n3⌉⌉−2)≥n3−4,

and the recurrence becomes

T(n)≤T(⌈n/3⌉)+T(2n/3+4)+O(n),T(n)≤T(⌈n/3⌉)+T(2n/3+4)+O(n),

which does not have a linear solution.

We can prove that the worst-case time for groups of 33 is Ω(nlgn)Ω(nlg⁡n). We do so by deriving a recurrence for a particular case that takes Ω(nlgn)Ω(nlg⁡n) time.

In counting up the number of elements greater than xx (and similarly, the number less than xx), consider the particular case in which there are exactly ⌈12⌈n3⌉⌉⌈12⌈n3⌉⌉ groups with medians ≥x≥x and in which the "leftover" group does contribute 2 elements greater than xx. Then the number of elements greater than xx is exactly 2(⌈12⌈n3⌉⌉−1)+12(⌈12⌈n3⌉⌉−1)+1 (the −1−1 discounts xx's group, as usual, and the +1+1 is contributed by xx's group) =2⌈n/6⌉−1=2⌈n/6⌉−1, and the recursive step for elements ≤x≤xhas n−(2⌈n/6⌉−1)≥n−(2(n/6+1)−1)=2n/3−1n−(2⌈n/6⌉−1)≥n−(2(n/6+1)−1)=2n/3−1 elements. Observe also that the O(n)O(n) term in the recurrence is really Θ(n)Θ(n), since the partitioning in step 4 takes Θ(n)Θ(n) (not just O(n)O(n)) time. Thus, we get the recurrence

T(n)≥T(⌈n/3⌉)+T(2n/3−1)+Θ(n)≥T(n/3)+T(2n/3−1)+Θ(n),T(n)≥T(⌈n/3⌉)+T(2n/3−1)+Θ(n)≥T(n/3)+T(2n/3−1)+Θ(n),

from which you can show that T(n)≥cnlgnT(n)≥cnlg⁡n by substitution. You can also see that T(n)T(n) is nonlinear by noticing that each level of the recursion tree sums to nn.

We assume that are given a procedure MEDIANMEDIAN that takes as parameters an array AA and subarray indices pp and rr, and returns the value of the median element of A[p..r]A[p..r] in O(n)O(n) time in the worst case.

Given MEDIANMEDIAN, here is a linear-time algorithm SELECT′SELECT′ for finding the iith smallest element in A[p..r]A[p..r]. This algorithm uses the deterministic PARTITIONPARTITION algorithm that was modified to take an element to partition around as an input parameter.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | SELECT'(A, p, r, i)  if p == r  return A[p]  x = MEDIAN(A, p, r)  q = PARTITION(x)  k = q - p + 1  if i == k  return A[q]  else if i < k  return SELECT'(A, p, q - 1, i)  else return SELECT'(A, q + 1, r, i - k) |

Because xx is the median of A[p..r]A[p..r], each of the subarrays A[p..q−1]A[p..q−1] and A[q+1..r]A[q+1..r] has at most half the number of elements of A[p..r]A[p..r]. The recurrence for the worst-case running time of SELECT′SELECT′ is T(n)≤T(n/2)+O(n)=O(n)T(n)≤T(n/2)+O(n)=O(n).

1. The median can be obtained recursively as follows. Pick the median of the sorted array A. This is just O(1) time as median is the n/2th element in the sorted array. Now compare the median of A, call is a ∗ with median of B, b ∗ . We have two cases. • a ∗ < b∗ : In this case, the elements in B[ n 2 · · · n] are also greater than a ∗ . So the median cannot lie in either A[1 · · · n 2 ] or B[ n 2 · · · n]. So we can just throw these away and recursively solve a subproblem with A[ n 2 · · · n] and B[1 · · · n 2 ]. • a ∗ > b∗ : In this case, we can still throw away B[1 · · · n 2 ] and also A[ n 2 · · · n] and solve a smaller subproblem recursively. In either case, our subproblem size reduces by a factor of half and we spend only constant time to compare the medians of A and B. So the recurrence relation would be T(n) = T(n/2) + O(1) which has a solution T(n) = O(log n).
2. Solution: There are numerous solutions to this problem. We will present a greedy strategy. Consider any path P in G. The courses in P must be scheduled in different semesters, since each course (but the last) in P is a pre-requisite for the next course in P. Therefore, we require a number of semesters at least equal to the length of P. Applying the argument to the longest path Q in G, the smallest number of semesters must be at least as large as the length of Q. Let k be the length of Q. If we could develop an algorithm that schedules all courses in k semesters, it would be optimal. A greedy strategy suggests itself. The algorithm operates in rounds. In round i, i > 1, the algorithm finds all courses that have no pre-requisites, schedules them in semester i, and deletes them from G. The algorithm terminates when G is empty.

Let us prove the optimality of the algorithm. Note that there must be at least one course without a pre-requisite in a DAG, as proven in (3.19) on page 102 of your textbook. Thus the algorithm removes at least one course in each round. The remaining graph is a DAG, since deleting a node from a DAG cannot introduce cycles. Therefore, the algorithm terminates in at most n rounds.

We will now prove that the algorithm actually terminates in k rounds, thereby establishing its optimality. Note that it is not enough to prove that the courses in the longest path Q are scheduled in k semesters. We have to prove, in addition, that for every path in G, the courses in that path are scheduled within k semesters. To do so, we define the depth dv of a course v ∈ G to be the number of courses in the longest path in G that terminates at v; we include v in this count. Clearly, the largest depth of a course is k. It suffices to prove that the greedy algorithm schedules every course v in semester dv.

We can prove this fact by induction. The base case is dv = 1. These are precisely the courses in G that have no pre-requisites. Indeed, the algorithm schedules these courses in semester 1. Now, for the inductive hypothesis, assume that the algorithm schedules all courses w with dw ≤ l in semester dw. We will now prove the statement for semester l + 1, i.e., for courses v with dv = l + 1. Consider the graph Gl that remains at the end of round l. Let v be a course with depth l+1 in G. What is the depth of v in Gl? If this depth is 1, we are done with the proof, since the algorithm will schedule v in this round (which is l + 1). Suppose the depth of v in Gl is larger than 1. Then v must have a pre-requisite u in Gl . What is du, the depth of u in G? By the inductive hypothesis, the algorithm has already scheduled all courses w with dw ≤ l in semester dw. Therefore, du must be larger than l, implying that the depth of v is dv > du + 1 > l + 1, which contradicts the fact that dv = l + 1. Note that we used the fact that the depth of a course must be at least 1 more than the depth of any pre-requisite for the course. Consequently, v has no pre-requisite in Gl , meaning that v will be scheduled in semester dv = l + 1, as desired. This completes the proof. As for the running time, we can modify the algorithm for topologically sorting a DAG on pages 103 and 104 of your textbook to achieve a running time of O(m + n), where m is the number of pre-requisite pairs in G. Briefly, for every course v, we maintain a counter pv set to the number of pre-requisites of v in G. In the first round, we find all courses with pv = 0 using a linear scan through G and schedule them. In round i, when we schedule a course u, we (i) decrement by 1 the pv values for all courses v for which u is a pre-requisite and (ii) add such a node v to the list of nodes to be scheduled in the next round if pv reaches 0. Since G is provided to us in adjacency list format, we can perform this operation in time proportional to the number of courses for which u is a pre-requisite. We process any edge (u, v) only when we schedule u. Therefore the total work done by the algorithm is O(m + n). Remarks: Some students proved the optimality of the algorithm by using the argument that the greedy algorithm always stays ahead of any other algorithm. Suppose the greedy algorithm uses l semesters to schedule all courses and the optimal algorithm uses m < l semesters. Let Gi , 1 ≤ i ≤ l be the set of the courses the greedy algorithm schedules in semester i and let Oi , 1 ≤ i ≤ m be the set of courses that the optimal algorithm schedules in semester i. Then for all 1 ≤ j ≤ m, [ j i=1 Gi ⊇ [ j i=1 Oi . These students proved the statement by induction, and then showed that m = l. This proof is valid and elegant, but it gives you less insight into the problem since it does not relate l to any structural quantity in G, namely the length of the longest path in G. It is not enough just to say that since “the greedy algorithm schedules each courses as soon as it can be scheduled, it must output the smallest number of semesters.” How do we know that an algorithm that does adopts a different strategy will not use fewer semesters? I deduced points for solutions without proof of correctness.

Another technical inaccuracy in many proofs was claiming that the smallest number of semesters in which the courses can be scheduled equals the length k of the longest path Q in G. This statement is not immediately obvious. The only thing that is clearly true is that the smallest number of semesters must be at least as large as k. It is always possible that more semesters may be needed. The algorithm is a constructive proof that all courses can indeed be scheduled in k semesters. I deduced some points for students who made this mistake.

Many students based their proof on notions such as “depth” of a node without defining the term. Since I could not guess what you mean, I deducted points, depending on how difficult it was to understand your solution.

Other students decided to use depth-first search, often by incrementing and decrementing counters keeping track of the length of the longest path. However, can you prove that you can indeed use DFS to compute the longest path in a DAG? I deducted 10 points for

DFS-based answers.

Let Ancestor[n][n] be an two dimentional array. If there is a query “is u an ancestor of v” our program will answer “yes” if A[u][v] is true otherwise it will report “no”.

Algorithm:

DFS-visit ( u , list){

Process\_ancestor();

time = time + 1

u.color = GRAY;

foreach v belongs to adj[u] {

append(list,v);

DFS-visit (v,list);

remove\_last\_element(); // which will remove v from the end of the list.

}

}

Process\_parent ( list ) {

if ( list.size > 1 ) {

for ( i=1; i<=list.size; i++) {

for ( j=i+1; j<=list.size; j++)

ancestor[list[i]][list[j]] = true; // list[i] is a ancestor of list[j]

}

}

}

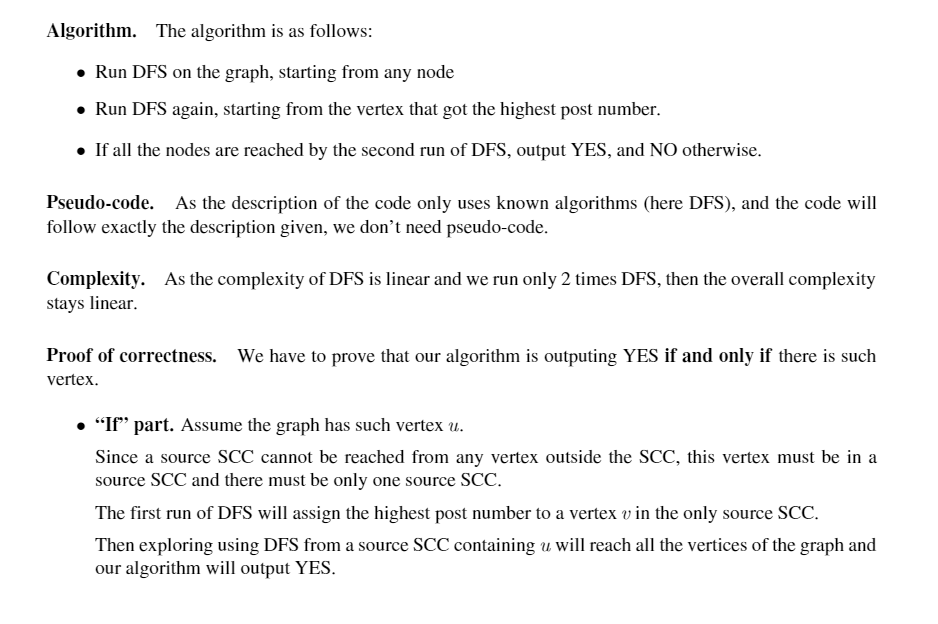
Process\_query( u,v) {   // This will answer our question.

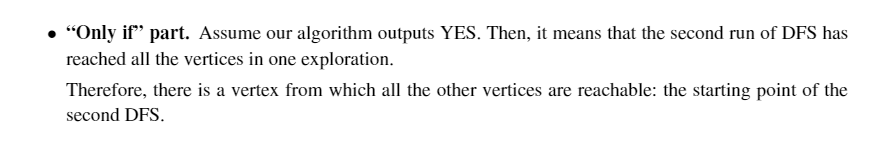
if ( ancestor[u][v] == true ) return yes;

return false;

}

1. D





1. D

