

Answer all questions. Give informal (at least) proofs for all answers. Grading will be on completeness and logical correctness, and if applicable, efficiency, as well as correctness. Out of 80 points.

Greedy algorithms Below, a greedy strategy is given for a problem. Then a lemma that helps prove the strategy optimal is stated, and a proof with some missing phrases is given. Fill in the missing phrases to complete the proof (20 points). Then give an efficient algorithm that implements the greedy strategy (with time analysis) (20 points).

Ripping Sequence A *ripping sequence* in an undirected graph is an ordering of the nodes as $v_1 \dots v_n$. The *forward degree* of a node v_i , is the number of neighbors that come later in the order, i.e., $|N(v_i) \cap \{v_{i+1} \dots v_n\}|$. The problem is to find a ripping sequence of minimal maximum forward degree.

The greedy strategy is to find a node of lowest degree, place that node first, and delete the node from the graph. Repeat, placing the new low degree node second, etc.

Theorem: The greedy algorithm produces an ordering of minimal maximum forward degree.

Proof: Let GO be the ordering produced by the greedy algorithm and let OO be any optimal solution, i.e., any ordering with the smallest possible **maximum forward degree**. Let d be the maximum forward degree of GO , and d' that of OO . We need to show $d \leq d'$. Let u be a vertex that has the maximum forward degree in GO , i.e., where the forward degree of u is d . Let G' be the graph when we delete all the nodes that occur before x in GO . Then, in G' , x has degree d and this is the **smallest degree** of any node in G' . Thus, every node in G' has at least d **neighbors** that are also in G' .

Let y be the first node from G' to occur in OO . From the above, y has **at least d neighbors in G'** . Since y is the first node in G' , all of these neighbors are later in the order in OO . Thus, the forward degree of y in OO is at least d , so the **maximum forward degree of OO , d'** , is at least d . Since OO had a minimal maximum forward degree d' and G' 's maximum forward degree d is at most this, GO also has **minimal maximum forward degree**. QED.

Implementation: We need to specify the data structures we will use to keep track of the degrees of nodes. We need to find the minimum degree node, and delete it. Then we need to decrement the degrees of its neighbors, to reflect their degrees in the graph when we delete that node. A heap allows us to find and delete a minimum element efficiently, and decrement the key of a node efficiently. So we'll use a heap of nodes in the graph keyed by degree. As with Dijkstra's algorithm, we maintain an array of pointers from the nodes to their positions in the heap.

Say the graph is given in adjacency list format. We can compute the degree of every node by running through its list, incrementing a counter. This takes total time $O(n + m)$. We then insert for each x , the pair $(x, \text{degree}(x))$ into a min-heap, while maintaining the array of pointers from x to its position in the heap. This takes time $O(n \log n)$, because we insert each of n elements and inserting takes time $O(\log n)$.

Then, while the heap is not empty, we find and delete the minimum element, $(x, \text{degree}(x))$. We put x next in the order. Then for each neighbor y of x , we follow the pointers to y 's position in the heap, decrement $\text{deg}(y)$, and re-adjust the heap structure. This takes $O(\log n)$ time for each of x 's neighbors.

We extract the minimum from the heap n times, once per node. Since we delete each node once, we will end up doing one decrement per edge. This takes time $O(m \log n)$ total. So the total time for all operations is $O(n + m + n \log n + m \log n) = O((n + m) \log n)$.

Divide-and-Conquer Below, we give a divide-and-conquer algorithm in pseudo-code. write down a recurrence for its time.(20 points) Then solve the recurrence to get an explicit formula up to order for the time of the algorithm. (20 points)

Binary Tree Iso Revisited Some graduate students in 202 pointed out that the Tree Isomorphism algorithm I gave on the sample exam can be improved, changing its $O(n^2)$ run-time to something smaller. Their improvement follows.

Consider the following recursive algorithm, which makes the following assumptions. x, y are the roots of two binary trees, T_x and T_y . $Left(z)$ is a pointer to the left child of node z in either tree, and $Right(z)$ points to the right child. If the node doesn't have a left or right child, the pointer returns "NIL". Each node z also has a field $Size(z)$ which returns the number of nodes in the sub-tree rooted at z . $Size(NIL)$ is defined to be 0. ($Size(x)$ has already been computed before this part of the algorithm is run, and so is just an $O(1)$ time look-up.)

The algorithm $SameTree(x, y)$ returns a boolean answer that says whether or not the trees rooted at x and y are isomorphic, i.e., the same if you ignore the difference between left and right pointers.

1. Program: $SameTree(x, y: \text{Nodes}): \text{Boolean};$
2. IF $Size(x) \neq Size(y)$ THEN return *False*; halt.
3. IF $x = NIL$ THEN return *True*; halt.
4. IF $SameTree(Left(x), Left(y))$ THEN Return $SameTree(Right(x), Right(y))$
5. ELSE Return $(SameTree(Right(x), Left(y)) \text{ AND } SameTree(Left(x), Right(y)))$

We make either two recursive calls or three recursive calls, depending on the result of the first recursive call. Thus, the worst-case is that we make three recursive calls. If the tree is balanced, both left and both right sub-trees will be of size $(n - 1)/2 < n/2$. The non-recursive part is constant time. So $T(n) \leq 3T(n/2) + O(1)$. Applying the Master Theorem with $A = 3, B = 2, K = 0$, since $3 > 2^0$, we are in the bottom-heavy case, and the time is $T(n) \in O(n^{\log_2 3})$.

(Note: if the tree is not balanced, and $|Left| \neq |Right|$, then some of the recursive calls will fail the size check, and return *False* immediately. There will be at most two that don't, if the sizes of the sub-trees are the same for both x and y , one the size of $Left(x)$, the other the size of $Right(x)$. So in this case we get the recurrence $T(n) = T(L) + T(R) + O(1)$, where $n = L + R + 1$. Although you did not need to do this for the quiz, it is possible to use this to show the same upper bound holds for the unbalanced case.)