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** For the problems below, a greedy strategy is given. Then a lemma that helps prove the strategy optimal is stated, and a proof of the lemma is given with some blank spaces. For each problem, you need to complete the proof of the stated lemma (20 points), and then give an efficient algorithm that implements the greedy strategy (with time analysis) (20 points).

**Coupon ordering** Consider the following "coupon collector" problem. There are a bunch of different varieties of cereal, and each comes with a single coupon for a discount on another variety of cereal. For example, the brands could be Rice Krisps, Korn Krisps, and Bran Mush, and Rice Krisps gives you a coupon for Korn Krisps, Korn Krisps gives you a coupon for Bran Mush, and Bran Mush gives you a coupon for Korn Krisps. You can use multiple coupons when purchasing a new box, and can even get money back if the value of the coupons is greater than the price. You want to buy one box of each variety, for as little money as possible. For each variety, you are told its price, the brand the coupon in the box is for, and the variety the coupon in the box is for. For example, if BranMush.price = $1.5, BranMush.couponvalue = $.5, BranMush.coupontype = KornKrisps, this says that the price of BranMush is a dollar and a half, but if we buy it, we get a coupon for fifty cents off KornKrisps.

Our algorithm needs to, given the above information for each of $n$ varieties, find an order to buy each variety that minimizes the total discounted prices.

Greedy strategy: Define the total discount for a variety to be the sum of the values of all coupons giving a discount for that variety. Buy the variety with smallest total discount. Subtract its coupon value from the price of the variety the coupon is for. Repeat until all varieties are bought.

Exchange Lemma: Let $V$ be a variety with the smallest total discount. Substruct its coupon value from the price of the variety the coupon is for. Repeat until all varieties are bought.

Let $OPT_0$ be any optimal ordering of varieties. Let $OPT_1$ be the ordering that buys $V$ first, then buys the other brands in the same order as $OPT_0$.

**Case 1:** If $V$ has total discount 0:

Since there was no possible discount for $V$, $V$ costs us $V.price$ in both $OPT_1$ and $OPT_0$. For all other varieties, the amount we pay for that variety in $OPT_1$ is no more than the amount we pay in $OPT_0$, since we have all of the coupons we would have in $OPT_0$ and possibly the coupon from buying $V$ as well. Thus, the total costs for $OPT$ are at most those of $OPT_0$. Since $OPT_0$ is a minimal total cost ordering by assumption, so is $OPT_1$.

**Case 2:** If $V$ has total discount > 0.

Since $V$ had the smallest maximum discount, in this case, all varieties have maximum discount positive. In particular, there must be at least one coupon for each variety. Since there are exactly $n$ coupons total, that means each variety has exactly one coupon for it. We pay the same for all varieties in $OPT_0$ and $OPT_1$ except $V$ and $V'$, where $V'$ is the variety that $V$ comes with a coupon for. If $V$ comes before $V'$ in $OPT_0$, we pay the same for each variety in both. Let $d_V$ be the discount of the coupon for $V$, and $d_{V'}$ be the discount of the coupon for $V'$. Then we know $d_V \leq d_{V'}$ by the definition of $V$. If $V'$ comes before $V$ in $OPT_0$, then we pay $d_V$ more for $V$ in $OPT_1$ than in $OPT_0$ and $d_{V'}$ less for $V'$ in $OPT_1$ than in $OPT_0$. Thus, the total costs for $OPT_1$ are at most those for $OPT_0$.

In either case, $OPT_1$ is an ordering on varieties where our total payments are at most those of $OPT_0$, and hence is an optimal ordering that buys $V$ first, the same as what the greedy algorithm does.

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

**Binary Tree Isomorphism** 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 \( \text{Right}(z) \) 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 \( \text{Size}(z) \) which returns the number of nodes in the sub-tree rooted at \( z \). \( \text{Size}(\text{NIL}) \) is defined to be 0. The algorithm \( \text{SameTree}(x, y) \) returns a boolean answer that says whether or not the trees rooted at \( x \) and \( y \) are the same if you ignore the difference between left and right pointers.

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

Assume the trees rooted at \( x \) and \( y \) are both complete balanced trees with \( n \) nodes. (Every interior node \( z \) in a complete balanced tree has \( \text{Size(Left}(z)) = \text{Size(Right}(z)) \).)

The algorithm makes four recursive calls. From the balanced tree assumption that you are given, each of the four calls is to a tree of half the size. The non-recursive part of the algorithm is constant time. Thus, we get the recurrence \( T(n) = 4T(n/2) + O(1) \). This fits the form of the Master Theorem, with \( a = 4, b = 2 \) and \( k = 0 \). Since \( a = 4 > 1 = b^k \), the recurrence is bottom-heavy, and the total time is \( O(n\log_2 4) = O(n^2) \).