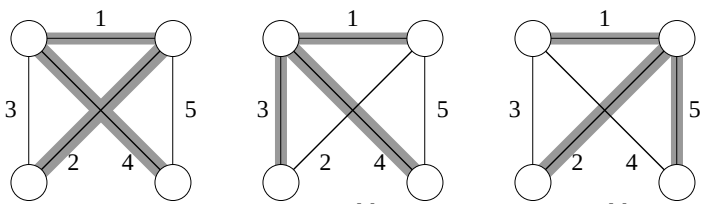
Solution to Exercise 4.5-2

We need to find the largest integer a such that $\log_4 a < \lg 7$. The answer is $a = 48$.

Solution to Problem 23-1

a. To see that the minimum spanning tree is unique, observe that since the graph is connected and all edge weights are distinct, then there is a unique light edge crossing every cut. By Exercise 23.1-6, the minimum spanning tree is unique.

To see that the second-best minimum spanning tree need not be unique, here is a weighted, undirected graph with a unique minimum spanning tree of weight 7 and two second-best minimum spanning trees of weight 8:



b. Since any spanning tree has exactly $|V| - 1$ edges, any second-best minimum spanning tree must have at least one edge that is not in the (best) minimum spanning tree. If a second-best minimum spanning tree has exactly one edge, say (x, y), that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except that (x, y) replaces some edge, say (u, v), of the minimum spanning tree. In this case, $T' = T - \{(u, v)\} \cup \{(x, y)\}$, as we wished to show.

Thus, all we need to show is that by replacing two or more edges of the minimum spanning tree, we cannot obtain a second-best minimum spanning tree. Let T be the minimum spanning tree of G, and suppose that there exists a second-best minimum spanning tree T' that differs from T by two or more

edges. There are at least two edges in $T - T'$, and let (u, v) be the edge in $T - T'$ with minimum weight. If we were to add (u, v) to T', we would get a cycle c. This cycle contains some edge (x, y) in $T' - T$ (since otherwise, T would contain a cycle).

We claim that $w(x, y) > w(u, v)$. We prove this claim by contradiction, so let us assume that $w(x, y) < w(u, v)$. (Recall the assumption that edge weights are distinct, so that we do not have to concern ourselves with $w(x, y) = w(u, v)$.) If we add (x, y) to T, we get a cycle c', which contains some edge (u', v') in $T - T'$ (since otherwise, T' would contain a cycle). Therefore, the set of edges $T'' = T - \{(u', v')\} \cup \{(x, y)\}$ forms a spanning tree, and we must also have $w(u', v') < w(x, y)$, since otherwise T'' would be a spanning tree with weight less than $w(T)$. Thus, $w(u', v') < w(x, y) < w(u, v)$, which contradicts our choice of (u, v) as the edge in $T - T'$ of minimum weight.

Since the edges (u, v) and (x, y) would be on a common cycle c if we were to add (u, v) to T', the set of edges $T' - \{(x, y)\} \cup \{(u, v)\}$ is a spanning tree, and its weight is less than $w(T')$. Moreover, it differs from T (because it differs from T' by only one edge). Thus, we have formed a spanning tree whose weight is less than $w(T')$ but is not T. Hence, T' was not a second-best minimum spanning tree.

c. We can fill in $max[u, v]$ for all $u, v \in V$ in $O(V^2)$ time by simply doing a search from each vertex u, having restricted the edges visited to those of the spanning tree T. It doesn't matter what kind of search we do: breadth-first, depth-first, or any other kind.

We'll give pseudocode for both breadth-first and depth-first approaches. Each approach differs from the pseudocode given in Chapter 22 in that we don't need to compute d or f values, and we'll use the *max* table itself to record whether a vertex has been visited in a given search. In particular, $max[u, v] = \text{NIL}$ if and only if $u = v$ or we have not yet visited vertex v in a search from vertex u. Note also that since we're visiting via edges in a spanning tree of an undirected graph, we are guaranteed that the search from each vertex u—whether breadth-first or depth-first—will visit all vertices. There will be no need to "restart" the search as is done in the DFS procedure of Section 22.3. Our pseudocode assumes that the adjacency list of each vertex consists only of edges in the spanning tree T.

Here's the breadth-first search approach:

```
BFS-FILL-MAX(G, T, w)
let max be a new table with an entry max[u, v] for each u, v ∈ G.V
for each vertex u ∈ G.V
    for each vertex v ∈ G.V
        max[u, v] = NIL
    Q = ∅
    ENQUEUE(Q, u)
    while Q ≠ ∅
        x = DEQUEUE(Q)
        for each v ∈ G.Adj[x]
            if max[u, v] == NIL and v ≠ u
                if x == u or w(x, v) > max[u, x]
                    max[u, v] = (x, v)
                else max[u, v] = max[u, x]
                ENQUEUE(Q, v)
return max
```

Here's the depth-first search approach:

```
DFS-FILL-MAX(G, T, w)
let max be a new table with an entry max[u, v] for each u, v ∈ G.V
for each vertex u ∈ G.V
    for each vertex v ∈ G.V
        max[u, v] = NIL
    DFS-FILL-MAX-VISIT(G, u, u, max)
return max

DFS-FILL-MAX-VISIT(G, u, x, max)
for each vertex v ∈ G.Adj[x]
    if max[u, v] == NIL and v ≠ u
        if x == u or w(x, v) > max[u, x]
            max[u, v] = (x, v)
        else max[u, v] = max[u, x]
        DFS-FILL-MAX-VISIT(G, u, v, max)
```

For either approach, we are filling in $|V|$ rows of the *max* table. Since the number of edges in the spanning tree is $|V| - 1$, each row takes $O(V)$ time to fill in. Thus, the total time to fill in the *max* table is $O(V^2)$.

d. In part (b), we established that we can find a second-best minimum spanning tree by replacing just one edge of the minimum spanning tree T by some edge (u, v) not in T. As we know, if we create spanning tree T' by replacing edge $(x, y) \in T$ by edge $(u, v) \notin T$, then $w(T') = w(T) - w(x, y) + w(u, v)$. For a given edge (u, v), the edge $(x, y) \in T$ that minimizes $w(T')$ is the edge of maximum weight on the unique path between u and v in T. If we have already computed the *max* table from part (c) based on T, then the identity of this edge is precisely what is stored in $max[u, v]$. All we have to do is determine an edge $(u, v) \notin T$ for which $w(max[u, v]) - w(u, v)$ is minimum.

Thus, our algorithm to find a second-best minimum spanning tree goes as follows:

1. Compute the minimum spanning tree T. Time: $O(E + V \lg V)$, using Prim's algorithm with a Fibonacci-heap implementation of the priority queue. Since $|E| < |V|^2$, this running time is $O(V^2)$.
2. Given the minimum spanning tree T, compute the *max* table, as in part (c). Time: $O(V^2)$.
3. Find an edge $(u, v) \notin T$ that minimizes $w(max[u, v]) - w(u, v)$. Time: $O(E)$, which is $O(V^2)$.
4. Having found an edge (u, v) in step 3, return $T' = T - \{max[u, v]\} \cup \{(u, v)\}$ as a second-best minimum spanning tree.

The total time is $O(V^2)$.

Solution to Exercise 16.1-4

This solution is also posted publicly

Let S be the set of n activities.

The "obvious" solution of using GREEDY-ACTIVITY-SELECTOR to find a maximum-size set S_1 of compatible activities from S for the first lecture hall, then using it again to find a maximum-size set S_2 of compatible activities from $S - S_1$ for the second hall, (and so on until all the activities are assigned), requires $\Theta(n^2)$ time in the worst case. Moreover, it can produce a result that uses more lecture halls

than necessary. Consider activities with the intervals $\{[1,4),[2,5),[6,7),[4,8)\}$. GREEDY-ACTIVITY-SELECTOR would choose the activities with intervals $[1,4)$ and $[6,7)$ for the first lecture hall, and then each of the activities with intervals $[2,5)$ and $[4,8)$ would have to go into its own hall, for a total of three halls used. An optimal solution would put the activities with intervals $[1,4)$ and $[4,8)$ into one hall and the activities with intervals $[2,5)$ and $[6,7)$ into another hall, for only two halls used.

There is a correct algorithm, however, whose asymptotic time is just the time needed to sort the activities by time—$O(n \lg n)$ time for arbitrary times, or possibly as fast as $O(n)$ if the times are small integers.

The general idea is to go through the activities in order of start time, assigning each to any hall that is available at that time. To do this, move through the set of events consisting of activities starting and activities finishing, in order of event time. Maintain two lists of lecture halls: Halls that are busy at the current event-time t (because they have been assigned an activity i that started at $s_i \le t$ but won't finish until $f_i > t$) and halls that are free at time t. (As in the activity-selection problem in Section 16.1, we are assuming that activity time intervals are half open—i.e., that if $s_i \ge f_j$, then activities i and j are compatible.) When t is the start time of some activity, assign that activity to a free hall and move the hall from the free list to the busy list. When t is the finish time of some activity, move the activity's hall from the busy list to the free list. (The activity is certainly in some hall, because the event times are processed in order and the activity must have started before its finish time t, hence must have been assigned to a hall.)

To avoid using more halls than necessary, always pick a hall that has already had an activity assigned to it, if possible, before picking a never-used hall. (This can be done by always working at the front of the free-halls list—putting freed halls onto the front of the list and taking halls from the front of the list—so that a new hall doesn't come to the front and get chosen if there are previously-used halls.)

This guarantees that the algorithm uses as few lecture halls as possible: The algorithm will terminate with a schedule requiring $m \le n$ lecture halls. Let activity i be the first activity scheduled in lecture hall m. The reason that i was put in the mth lecture hall is that the first $m-1$ lecture halls were busy at time s_i. So at this time there are m activities occurring simultaneously. Therefore any schedule must use at least m lecture halls, so the schedule returned by the algorithm is optimal.

Run time:

- Sort the $2n$ activity-starts/activity-ends events. (In the sorted order, an activity-ending event should precede an activity-starting event that is at the same time.) $O(n \lg n)$ time for arbitrary times, possibly $O(n)$ if the times are restricted (e.g., to small integers).
- Process the events in $O(n)$ time: Scan the $2n$ events, doing $O(1)$ work for each (moving a hall from one list to the other and possibly associating an activity with it).

Total: $O(n + \text{time to sort})$

[The idea of this algorithm is related to the rectangle-overlap algorithm in Exercise 14.3-7.]

Solution to Exercise 16.2-2

This solution is also posted publicly

The solution is based on the optimal-substructure observation in the text: Let i be the highest-numbered item in an optimal solution S for W pounds and items $1, \ldots, n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1, \ldots, i-1$, and the value of the solution S is v_i plus the value of the subproblem solution S'.

We can express this relationship in the following formula: Define $c[i, w]$ to be the value of the solution for items $1, \ldots, i$ and maximum weight w. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \ , \\ c[i-1,w] & \text{if } w_i > w \ , \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i \ . \end{cases}$$

The last case says that the value of a solution for i items either includes item i, in which case it is v_i plus a subproblem solution for $i-1$ items and the weight excluding w_i, or doesn't include item i, in which case it is a subproblem solution for $i-1$ items and the same weight. That is, if the thief picks item i, he takes v_i value, and he can choose from items $1, \ldots, i-1$ up to the weight limit $w - w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if he decides not to take item i, he can choose from items $1, \ldots, i-1$ up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made.

The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $w = \langle w_1, w_2, \ldots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0..n, 0..W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.

DYNAMIC-0-1-KNAPSACK(v, w, n, W)

```
let c[0..n, 0..W] be a new array
for w = 0 to W
    c[0, w] = 0
for i = 1 to n
    c[i, 0] = 0
    for w = 1 to W
        if w_i ≤ w
            if v_i + c[i − 1, w − w_i] > c[i − 1, w]
                c[i, w] = v_i + c[i − 1, w − w_i]
            else c[i, w] = c[i − 1, w]
        else c[i, w] = c[i − 1, w]
```

We can use the c table to deduce the set of items to take by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

The above algorithm takes $\Theta(nW)$ time total:

- $\Theta(nW)$ to fill in the c table: $(n+1) \cdot (W+1)$ entries, each requiring $\Theta(1)$ time to compute.
- $O(n)$ time to trace the solution (since it starts in row n of the table and moves up one row at each step).

Solution to Exercise 16.2-7

This solution is also posted publicly

Sort A and B into monotonically decreasing order.

Here's a proof that this method yields an optimal solution. Consider any indices i and j such that $i < j$, and consider the terms ${a_i}^{b_i}$ and ${a_j}^{b_j}$. We want to show that it is no worse to include these terms in the payoff than to include ${a_i}^{b_j}$ and ${a_j}^{b_i}$, i.e., that ${a_i}^{b_i}{a_j}^{b_j} \geq {a_i}^{b_j}{a_j}^{b_i}$. Since A and B are sorted into monotonically decreasing order and $i < j$, we have $a_i \geq a_j$ and $b_i \geq b_j$. Since a_i and a_j are positive and $b_i - b_j$ is nonnegative, we have ${a_i}^{b_i - b_j} \geq {a_j}^{b_i - b_j}$. Multiplying both sides by ${a_i}^{b_j}{a_j}^{b_j}$ yields ${a_i}^{b_i}{a_j}^{b_j} \geq {a_i}^{b_j}{a_j}^{b_i}$.

Since the order of multiplication doesn't matter, sorting A and B into monotonically increasing order works as well.