1. *7 points: 1 point for (a), 4 for (b), 2 for (c).*

Problem 21-1 on page 582 (page 518 in 2/e).

Solution:

(a) [4 , 3 , 2 , 6 , 8 , 1]

(b) We show, by induction on the number of iterations i of the outer loop, that the entries *extracted*$[j]$ for which *extracted*$[j] \leq i$ have been computed correctly.

Think of a problem instance as a sequence in which elements from $\{i, i+1, \ldots, m\}$ are inserted and there are k instances of EXTRACT-MIN. Along with our induction proof we show that the *sequence* of K_j's for each instance is correct, i.e., the p-th element in the sequence (list) corresponds to the set of insertions between the p-th and $p+1$-st extraction. Finally, the set of insertions before the j-th (original) extraction is correctly labeled as K_j.

If a single EXTRACT-MIN operation is removed from a sequence then the sequence of insertions after it is effectively concatenated to the sequence of insertions before it.

Base. If $i = 0$ there is nothing to prove.

Induction step. For arbitrary i, we can reduce the problem instance to a smaller instance by observing that when line 4 is reached, K_j is the subset of insertions that take place right before the j-th EXTRACT-MIN. Since the current value of i is the minimum overall, it will be extracted by the first EXTRACT-MIN after the elements of K_j have been inserted.

Executing the EXTRACT-MIN leaves elements $i+1, \ldots, n$ and a sequence of EXTRACT-MIN's that does not include the j-th one. Let $E_j, E_{j+2}, \ldots, E_k$ be the sequence of EXTRACT-MIN's, starting with the j-th, for which *extracted* has already been given a value. Then the next EXTRACT-MIN, E_{k+1} is the first one after the j-th that still remains in the reduced sequence. In the list of the K sets, $K_{k+1} = K_\ell$ is the set of insertions before E_ℓ before the current iteration. At the end of the current iteration, I_ℓ is concatenated to I_j, which, in the reduced instance, means K_ℓ is unioned with K_j.

So in the i-th iteration, the correct value of *extracted*$[j]$ is filled in and the problem instance is reduced appropriately to a smaller one. □

Note: It's important to distinguish between an *empty* K_j and one that does not appear on the list. The empty K_j's still *exist*. Also, it's important to distinguish between *destroying* a K_j (removing it from the list) and deleting its elements. The algorithm never does the latter.

In the example ...

$i = 1$: $1 \in K_6$ so *extracted*$[6] = 1$; then $K_7 = \{1, 5, 7\}$ (we don't need to worry about the 1 – it will not be considered again).

$i = 2$: $2 \in K_3$ so *extracted*$[3] = 2$; then $K_4 = \{2, 6, 9\}$.

$i = 3$: $3 \in K_2$ so *extracted*$[2] = 3$; then $K_4 = \{2, 3, 6, 9\}$.

$i = 4$: $4 \in K_1$ so *extracted*$[1] = 4$; then $K_4 = \{2, 3, 4, 6, 8, 9\}$.

$i = 5$: $5 \in K_7$; do nothing.

$i = 6$: $2 \in K_4$ so *extracted*$[4] = 6$; then $K_5 = \{2, 3, 4, 6, 8, 9\}$ (the original K_5 was empty.

$i = 7$: $7 \in K_7$ do nothing.

$i = 8$: $8 \in K_5$ so *extracted*$[5] = 2$; then $K_7 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

(c) The K_j sets can easily be created initially by reading the sequence and doing adding each element of I_j to the set K_j via a UNION. The only parts of the algorithm that are non-trivial are lines 5 and 6. We maintain the K_j's in a doubly-linked list L. An array *position* is set up so that, if i is the root of a tree in the disjoint set forest, *position*$[i]$ is a pointer to the position of K_j in the list, where K_j is the set to which i belongs. The values of *position*$[i]$ for the roots are easy to maintain during

a LINK operations. This allows us to destroy a particular K_j (remove it from the list) in constant time and the K_ℓ in line 5 will simply be the next element on the list.

The K_j's in the list are (pointers to) the roots of the corresponding disjoint set forest trees, or null if the set is empty.

Then line 2 is simply a $j \leftarrow$ FIND-SET(i), which retrieves not only the root of the tree containing i but also the value of j. In line 5, we let K_ℓ be the item after K_j in L. Then line 6 is UNION$i, root(K_\ell)$ followed by a removal of K_j from L: constant time since L is doubly linked a pointer to the position of K_j is maintained with *pointer*[FIND(i)]. If K_ℓ is empty, only the removal from the list takes place.

The worst case running time is dominated by the $O(n\alpha(n))$ taken by the UNION's and FIND-SET's: note that there are at most n of each. Otherwise, the time per iteration of the for-loop is constant.

In part (b), 2 points for any attempt at induction, remaining points for details that are convincing or almost so; in part (c), 1 point for disjoint set union and time bound, 1 point for list manipulation.

2. *4 points: 2 points each part.*

(a) Exercise 22.3-8 on page 611 (22.3-7 on page 548 in 2/e). Conjecture: If there's a path from u to v and $u.d < v.d$, then v is a descendant of u.

Counterexample: Consider a graph G with only three vertices, t, u, and v and edges tu, ut, and tv. Suppose DFS-VISIT is called for t and then recursively for u and v in that order. Clearly $u.d < v.d$ and there is a path utv from u to v but v is neither an ancestor nor a descendant of u.

(b) Exercise 22.3-11 on page 612 (22.3-10 on page 549 in 2/e). How can a vertex u appear as the only vertex in a DFS tree even though it has both incoming and outgoing edges?

Example: The incoming edges and outgoing edges must all be cross edges. A simple example illustrates this. Let $G = (\{u, v_{in}, v_{out}\}, \{v_{in}u, uv_{out}\})$ and suppose DFS-VISIT is called for v_{out}, u, and v_{in} in that order. Then u will be in a tree by itself.

3. *16 points: 2 points each part*

Problem 22-2 on page 621 (page 558 in 2/e).

(a) If the root r of G_π has two children x and y, there will not be any path from x or any of its descendants to y – the only edges other than tree edges are back edges; thus an edge from a descendant of x must lead to r or one of the descendants of x. Removing r would leave no path between x and y.

If r has only one child x, it cannot be an articulation point. Removing r will still leave x and all of its descendants connected to each other.

(b) Suppose s is a child of v and neither s nor any of its descendants have a back edge to a proper ancestor of v. Then there will be no paths from s to any proper ancestor of v and removal of v will leave no path from s to the root.

If a child of s of v *does* have a descendant with a back edge to a proper ancestor of v, then there will be a path that does not include v from any descendant of s (including s) to any proper ancestor of v and also to nodes reachable from those proper ancestors via other tree branches. Thus removal of v does not disconnect s and its descendants from any of the rest of the tree, except the descendants of other children of v. However, if all descendants children of v have paths to proper ancestors of v, then they all have paths to each other and removal of v will not disconnect the graph.

(c) You can modify depth-first search so that, when v is visited it sets $v.low$ to be the minimum among $v.d$, and for all $vw \in E$, $w.d$, if vw is a back edge, or $w.low$, if vw is a tree edge. When vw is a tree edge, $w.low$ is calculated during DFS-VISIT(w), allowing $v.low$ to be updated if appropriate after DFS-VISIT(w). The time bound is the same as for DFS.

(d) Based on (a) and (b) we need to check (i) if the root of the DFS tree has more than one child – then the root is an articulation point; and (ii) if any vertex v has $v.low = v.d$ – then d is an articulation point because none of its descendants have back edges to proper ancestors

of v. The modified version of DFS-Visit in Fig. 1 outputs the articulation points of G. The procedure Articulation-Points(v) is called with v = any arbitrary vertex. If there is more than one connected component, an arbitrary vertex in each component does the trick, as with the outer loop of DFS.

(e) If vw lies on a simple cycle, there is a path between v and w that does not use vw. Thus any path that uses vw can use the other path instead and vw is therefore not a bridge.

Conversely, if vw is not a bridge, then, when it is removed, there must be a path between x and y for any two vertices x and y. In particular, there must be a path between v and w that does not use vw and vw combined with that path forms a simple cycle.

(f) The idea is to use the *low* values to determine whether an edge between a vertex v and its parent $v.\pi$ in the tree is a bridge: this is the case whenever $v.\pi \neq$ ***null*** and $v.low = v.d$. A simple modification of the algorithm in Fig. 1 can be used to check this. All we need to do is set $w.\pi = v$ before the recursive call that takes place if w is not marked. This is no accident – the endpoints of a bridge are both articulation points. Note: It is not necessary to count the children of the root – whether or not the root is an articulation point is irrelevant to the computation of bridges.

(g) It is easy to see that two biconnected components cannot have more than one vertex in common: if v and w were each in both C_1 and C_2, there would be a path between v and w using edges of C_1 and another path using edges of C_2. Combining these would yield a cycle that includes edges from both C_1 and C_2, a contradiction of the fact that C_1 and C_2 are maximal.

Now we can argue that each non-bridge edge belongs to exactly one biconnected component, i.e., the bcc's partition these edges.

Let vw be a non-bridge edge. Then vw must be part of a simple cycle and therefore belong to at least one component. If vw belongs to more than one component, then at least two components have both v and w in common, a contradiction to our first observation.

(h) The challenge here is that, in order to know whether the edges among v and its descendants form a biconnected component, you have to know whether v is an articulation point or the edge between v and its parent is a bridge. Neither of these can be determined until $v.low$ has been computed at the end of the visit to v. The solution, initially proposed by Tarjan, is to stack the edges as you go, popping them off the stack when you know you've reached the end of a component.

4. *4 points: 2 points each part.*

(a) Exercise 23.1-5 on page 629 (page 566 in 2/e).

Solution: e is the maximum-weight edge on some cycle of $G = (V, E)$. Let T be the minimum spanning tree of G. We prove it by contradiction. That is, if any minimum spanning tree in G must contain e, T includes e. Thus, removing e divides the graph G into the two components of the tree that are disconnected. Since e is the maximum-weight edge on some cycle of $G = (V, E)$, there is an edge e' with $weight(e') \leq weight(e)$ such that e' crosses the two components. Let $T' = (T - \{e\}) \bigcup \{e'\}$. Then T' is a spanning tree of G and the weight of $T' \leq$ the weight of T. Since T is a minimum spanning tree of G, $weight(T') = weight(T)$ (i.e., $weight(e') = weight(e)$). This means that T' is a minimum spanning tree of G as well and does not contain e. This is a contradiction. Hence, there is a minimum spanning tree of G that doesn't include e.

(b) Exercise 23.2-8 on page 637 (page 574 in 2/e).

Solution: Consider the graph

```
(ab, 1)
(ac, 2)
(bd, 3)
(cd, 4)
```

and suppose the algorithm lets $V_1 = \{a, b\}$ and $V_2 = \{c, d\}$. Splitting each of these sets in half results in single vertices and empty MST's, the base case for the recursion. The algorithm will first

add the only edge of V_1, i.e., ab, to the MST; then the edge cd from V_2 is added. The minimum cost edge crossing the cut between V_1 and V_2 is ac. So the MST created by the algorithm is $\{ab, cd, ac\}$ for a cost of 7. But the tree $\{ab, ac, bd\}$ has a cost of 6.

5. *14 points: 2 points for (a), 4 each for (b), (c), (d).*

Problem 23-1 on page 638 (page 575 in 2/e). Second best MST.

(a) To see that there is a unique MST if all edge weights are distinct, let T be one of them and T' the other. Take an edge e of $T - T'$ and add it to T to create a cycle C. Consider a cutset D for which e is a light edge, and, because $|C \cap D|$ is even, contains another edge $f \in T$. We know that $w(f) < w(e)$ so $w(T - f + e) < w(T)$, a contradiction.

The second best MST is not necessarily unique. Consider a graph with edges (and weights):

```
a b 1
a c 3
a d 4
b c 2
b d 5
```

The minimum spanning tree is `ab ad bc` with cost 7. There are two second best spanning trees: `ab ac ad` and `ab bc bd`; both have cost 8.

(b) Since any spanning tree has exactly $n - 1$ edges, any second-best spanning tree must have at least one edge that is not in the minimum spanning tree. If a second-best spanning tree has exactly one edge, say xy, that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except that xy replaces some edge, say uv, of the minimum spanning tree. In this case, $T' = T - uv + xy$, 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 spanning tree.

Let T be the minimum spanning tree of G, and suppose that there exists a second-best spanning tree T' that differs from T by two or more edges. There are at least two edges in $T - T'$, and let uv be the edge in $T - T'$ with minimum weight. If we were to add uv to T', we would get a cycle C. This cycle contains some edge $xy \in T' - T$ (since otherwise, T would contain a cycle). We claim that $w(xy) > w(uv)$.

We prove this claim by contradiction, so let us assume that $w(xy) < w(uv)$. (Recall the assumption that edge weights are distinct, so that we do not have to concern ourselves with $w(xy) = w(uv)$.) If we add xy 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 $T'' = T - u'v' + xy$ forms a spanning tree, and we must also have $w(uv) < w(xy)$, since otherwise T'' would be a spanning tree with weight less than $w(T)$. Thus, $w(uv) < w(xy) < w(uv)$, which contradicts our choice of uv as the edge in $T - T'$ of minimum weight. Since the edges uv and xy would be on a common cycle C if we were to add uv to T', the set of edges $T' - xy + uv$ 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 spanning tree.

(c) We can fill in $\max[u, v]$ for all $u, v \in V$ in $O(n^2)$ 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. Suppose we use DFS.

The algorithm initializes $\max[u, v]$ to ***null*** for each pair of vertices u, v. Then it does a DFS-Visit from each vertex, with the additional feature that DFS-Visit takes two arguments, the starting point (root of the tree) and the current vertex being visited. So,

```
DFS-Visit(u, x) is
    for all v adjacent to u do
        if max[u, v] = null and v ≠ u then
         (1) if x = u or w(x, v) > max[u, x] then
               (2) max[u, v] ← xv
         (3) else max[u, v] ← max[u, x] endif
        endif
        DFS-Visit(u, v)
    end do
end DFS-Visit
```

Lines (1)-(3) ensure that, when a path is extended by adding edge xv, then either then weight of xv is the new maximum – because it's larger than anything on the path, or the maximum on the path stays what it was for v's parent x.

(d) From part (b) we know that a second best tree is the result of a single swap from the minimum spanning tree T. All we need to do, therefore, is find the best such swap, i.e., find an edge $uv \notin T$ that minimizes $w(\max[u, v]) - w(uv)$. The steps for the resulting algorithm are:

- Find the minimum spanning tree T — easily done in $O(n^2)$ using Prim's algorithm with an unordered list as a heap.
- Compute $\max[u, v]$ for every pair of vertices u, v — takes $\Theta(n^2)$ using part (c).
- Find $\min_{uv \notin T}(w(\max[u, v]) - w(uv))$ — takes $O(m)$; just traverse all the edges.
- Report the edge uv that achieved the minimum in step 3 and output $T' = T - max[u, v] + uv$ as a second best spanning tree — constant time, assuming we keep track of the edge that achieved the minimum value in step 3.

Total time is $\Theta(n^2)$, dominated by the computation of the $\max[u, v]$ values. The problem can actually be solved within the same time bound as it takes to find the original MST using a technique described by Gabow.[1]

[1] Harold N. Gabow, *Two Algorithms for Generating Weighted Spanning Trees in Order*, Technical Report CU-CS078-75, University of Colorado, August 1975. Probably published as a journal paper since then.

```
▷ initially called with v = any arbitrary vertex
▷ initially all vertices are unmarked and time = 0
ARTICULATION-POINTS(v) is
    mark v
    time ← time + 1; v.d ← time; v.low ← time
    ▷ cut becomes true if v is a cut vertex.
    cut ← false
    ▷ children counts the number of children of v in case it is the root.
    children ← 0
    for vw ∈ E do
        if w is not marked then
            ARTICULATION-POINTS(v); children ← children + 1
            if w.low < v.low then v.low ← w.low endif
        else ▷ back edge
            if w.d < v.low then v.low ← w.d endif

        endif
    end do
    if v.d = 1 and children > 1 then v is an articulation point
    else if v.low = v.d then v is an articulation point endif
end ARTICULATION-POINTS
```

Figure 1: Algorithm to compute articulation points of a connected undirected graph G.

```
▷ when the algorithm concludes, each non-bridge edge vw has a label C[vw], its biconnected component.
▷ initially all vertices are unmarked and time = 0
▷ component is the current component number, initially 0
▷ S is a stack of edges, initially empty
BCC(v) is
    mark v
    time ← time + 1; v.d ← time; v.low ← time
    for vw ∈ E do
        if w is not marked then
            w.π ← v
            push vw onto S
            BCC(w)
            if w.low ≥ v.d then
                ▷ vw is a bridge, w and its descendants form a component
                component ← component + 1
                while top of S ≠ vw do
                    pop xy from S and set C[xy] = component
                end do
                pop vw
            else if w.low < v.low
                ▷ there's a cycle between w back to an ancestor of v
                v.low ← w.low

        else ▷ back edge
            if w.d < v.low  and w.π ≠ v then
                ▷ there's a cycle that includes vw and the path from w to v in the tree
                push vw onto S
                v.low ← w.d
            endif
        endif
    end do
    if v.low = v.d then
        ▷ v is an articulation point, need to assign a different component to each subtree
        for vw ∈ E do
            component ← component + 1
            while top of S ≠ vw do
                pop xy from S and set C[xy] = component
            pop vw from S and set C[vw] = component
            end do
        end do
    endif
end BCC
```

Figure 2: Algorithm to compute biconnected components of a connected undirected graph G.