23 Minimum Spanning Trees

23.1 Growing a minimum spanning tree

23.1-1

> Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

(Removed)

23.1-2

> Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

Let G be the graph with 4 vertices: u, v, w, z. Let the edges of the graph be $(u, v), (u, w), (w, z)$ with weights 3, 1, and 2 respectively.

Suppose A is the set $\{(u, w)\}$. Let $S = A$. Then S clearly respects A. Since G is a tree, its minimum spanning tree is itself, so A is trivially a subset of a minimum spanning tree.

Moreover, every edge is safe. In particular, (u, v) is safe but not a light edge for the cut. Therefore Professor Sabatier's conjecture is false.

23.1-3

> Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Let T_0 and T_1 be the two trees that are obtained by removing edge (u, v) from a MST. Suppose that V_0 and V_1 are the vertices of T_0 and T_1 respectively.

Consider the cut which separates V_0 from V_1. Suppose to a contradiction that there is some edge that has weight less than that of (u, v) in this cut. Then, we could construct a minimum spanning tree of the whole graph by adding that edge to $T_1 \cup T_0$. This would result in a minimum spanning tree that has weight less than the original minimum spanning tree that contained (u, v).

23.1-4

> Give a simple example of a connected graph such that the set of edges $\{(u, v)😒 there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

(Removed)

23.1-5

> Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

Let A be any cut that causes some vertices in the cycle on once side of the cut, and some vertices in the cycle on the other. For any of these cuts, we know that the edge e is not a light edge for this cut. Since all the other cuts won't have the edge e crossing it, we won't have that the edge is light for any of those cuts either. This means that we have that e is not safe.

23.1-6

> Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

(Removed)

23.1-7

> Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

First, we show that the subset of edges of minimum total weight that connects all the vertices is a tree. To see this, suppose not, that it had a cycle. This would mean that removing any of the edges in this cycle would mean that the remaining edges would still connect all the vertices, but would have a total weight that's less by the weight of the edge that was removed. This would contradict the minimality of the total weight of the subset of vertices. Since the subset of edges forms a tree, and has minimal total weight, it must also be a minimum spanning tree.

To see that this conclusion is not true if we allow negative edge weights, we provide a construction. Consider the graph K_3 with all edge weights equal to -1. The only minimum weight set of edges that connects the graph has total weight -3, and consists of all the edges. This is clearly not a MST because it is not a tree, which can be easily seen because it has one more edge than a tree on three vertices should have. Any MST of this weighted graph must have weight that is at least -2.

23.1-8

> Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights

> of T'.

Suppose that L' is another sorted list of edge weights of a minimum spanning tree. If $L' \ne L$, there must be a first edge (u, v) in T or T' which is of smaller weight than the corresponding edge (x, y) in the other set. Without loss of generality, assume (u, v) is in T.

Let C be the graph obtained by adding (u, v) to L'. Then we must have introduced a cycle. If there exists an edge on that cycle which is of larger weight than (u, v), we can remove it to obtain a tree C' of weight strictly smaller than the weight of T', contradicting the fact that T' is a minimum spanning tree.

Thus, every edge on the cycle must be of lesser or equal weight than (u, v). Suppose that every edge is of strictly smaller weight. Remove (u, v) from T to disconnect it into two components. There must exist some edge besides (u, v) on the cycle which would connect these, and since it has smaller weight we can use that edge instead to create a spanning tree with less weight than T, a contradiction. Thus, some edge on the cycle has the same weight as (u, v). Replace that edge by (u, v). The corresponding lists L and L' remain unchanged since we have swapped out an edge of equal weight, but the number of edges which T and T' have in common has increased by 1.

If we continue in this way, eventually they must have every edge in common, contradicting the fact that their edge weights differ somewhere. Therefore all minimum spanning trees have the same sorted list of edge weights.

23.1-9

> Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

Suppose that there was some cheaper spanning tree than T'. That is, we have that there is some T'' so that $w(T'') < w(T')$. Then, let S be the edges in T but not in T'. We can then construct a minimum spanning tree of G by considering $S \cup T''$. This is a spanning tree since $S \cup T'$ is, and T'' makes all the vertices in V' connected just like T' does.

However, we have that

$$w(S \cup T'') = w(S) + w(T'') < w(S) + w(T') = w(S \cup T') = w(T).$$

This means that we just found a spanning tree that has a lower total weight than a minimum spanning tree. This is a contradiction, and so our assumption that there was a spanning tree of V' cheaper than T' must be false.

23.1-10

> Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges in T. Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \in T$ and a positive number k, and define the weight function w' by

> $$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y), \\ w(x, y) - k & \text{if } (u, v) = (x, y). \end{cases}$$
>
> Show that T is a minimum spanning tree for G with edge weights given by w'.

(Removed)

23.1-11 ⋆

> Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges not in T. Give an algorithm for finding the minimum spanning tree in the modified graph.

If we were to add in this newly decreased edge to the given tree, we would be creating a cycle. Then, if we were to remove any one of the edges along this cycle, we would still have a spanning tree. This means that we look at all the weights along this cycle formed by adding in the decreased edge, and remove the edge in the cycle of maximum weight. This does exactly what we want since we could only possibly want to add in the single decreased edge, and then, from there we change the graph back to a tree in the way that makes its total weight minimized.

23.2 The algorithms of Kruskal and Prim

23.2-1

> Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.

Suppose that we wanted to pick T as our minimum spanning tree. Then, to obtain this tree with Kruskal's algorithm, we will order the edges first by their weight, but then will resolve ties in edge weights by picking an edge first if it is contained in the minimum spanning tree, and treating all the edges that aren't in T as being slightly larger, even though they have the same actual weight.

With this ordering, we will still be finding a tree of the same weight as all the minimum spanning trees $w(T)$. However, since we prioritize the edges in T, we have that we will pick them over any other edges that may be in other minimum spanning trees.

23.2-2

> Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

At each step of the algorithm we will add an edge from a vertex in the tree created so far to a vertex not in the tree, such that this edge has minimum weight. Thus, it will be useful to know, for each vertex not in the tree, the edge from that vertex to some vertex in the tree of minimal weight. We will store this information in an array A, where $A[u] = (v, w)$ if w is the weight of (u, v) and is minimal among the weights of edges from u to some vertex v in the tree built so far. We'll use $A[u].1$ to access v and $A[u].2$ to access w.

```
PRIM-ADJ(G, w, r)
    initialize A with every entry = (NIL, ∞)
    T = {r}
    for i = 1 to V
        if Adj[r, i] != 0
            A[i] = (r, w(r, i))
    for each u in V - T
        k = min(A[i].2)
        T = T ∪ {k}
        k.π = A[k].1
        for i = 1 to V
            if Adf[k, i] != 0 and Adj[k, i] < A[i].2
                A[i] = (k, Adj[k, i])
```

23.2-3

> For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

Prim's algorithm implemented with a Binary heap has runtime $O((V + E) \lg V)$, which in the sparse case, is just $O(V \lg V)$. The implementation with Fibonacci heaps is

$$O(E + V \lg V) = O(V + V \lg V) = O(V \lg V).$$

- In the sparse case, the two algorithms have the same asymptotic runtimes.
- In the dense case.
  - The binary heap implementation has a runtime of

    $$O((V + E) \lg V) = O((V + V^2) \lg V) = O(V^2 \lg V).$$

  - The Fibonacci heap implementation has a runtime of

    $$O(E + V \lg V) = O(V^2 + V \lg V) = O(V^2).$$

  So, in the dense case, we have that the Fibonacci heap implementation is asymptotically faster.
- The Fibonacci heap implementation will be asymptotically faster so long as $E = \omega(V)$. Suppose that we have some function that grows more quickly than linear, say f, and $E = f(V)$.
- The binary heap implementation will have runtime of

  $$O((V + E) \lg V) = O((V + f(V)) \lg V) = O(f(V) \lg V).$$

However, we have that the runtime of the Fibonacci heap implementation will have runtime of

$$O(E + V \lg V) = O(f(V) + V \lg V).$$

This runtime is either $O(f(V))$ or $O(V \lg V)$ depending on if $f(V)$ grows more or less quickly than $V \lg V$ respectively.

In either case, we have that the runtime is faster than $O(f(V) \lg V)$.

23.2-4

> Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

(Removed)

23.2-5

> Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

For the first case, we can use a van Emde Boas tree to improve the time bound to $O(E \lg \lg V)$. Comparing to the Fibonacci heap implementation, this improves the asymptotic running time only for sparse graphs, and it cannot improve the running time polynomially. An advantage of this implementation is that it may have a lower overhead.

For the second case, we can use a collection of doubly linked lists, each corresponding to an edge weight. This improves the bound to $O(E)$.

23.2-6 ⋆

> Suppose that the edge weights in a graph are uniformly distributed over the halfopen interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

For input drawn from a uniform distribution I would use bucket sort with Kruskal's algorithm, for expected linear time sorting of edges by weight. This would achieve expected runtime $O(E\alpha(V))$.

23.2-7 ⋆

> Suppose that a graph G has a minimum spanning tree already computed. How quickly can we update the minimum spanning tree if we add a new vertex and incident edges to G?

(Removed)

23.2-8

> Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1

> and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.
>
> Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

The algorithm fails. Suppose $E = \{(u, v), (u, w), (v, w)\}$, the weight of (u, v) and (u, w) is 1, and the weight of (v, w) is 1000, partition the set into two sets $V_1 = \{u\}$ and $V_2 = \{v, w\}$.

Problem 23-1 Second-best minimum spanning tree

> Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \ge |V|$ and all edge weights are distinct.
>
> We define a second-best minimum spanning tree as follows. Let $\square$ be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T such that $W(T) = \min_{T'' \in \square - \{T'\}} \{w(T'')\}$.
>
> **a.** Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.
>
> **b.** Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.
>
> **c.** Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $\max[u, v]$ for all $u, v \in V$.
>
> **d.** Give an efficient algorithm to compute the second-best minimum spanning tree of G.

a. To see that the second best minimum spanning tree need not be unique, we consider the following example graph on four vertices. Suppose the vertices are $\{a, b, c, d\}$, and the edge weights are as follows:

	a	b	c	d
a	−	1	4	3
b	1	−	5	2
c	4	5	−	6
d	3	2	6	−

Then, the minimum spanning tree has weight 7, but there are two spanning trees of the second best weight, 8.

b. We are trying to show that there is a single edge swap that can demote our minimum spanning tree to a second best minimum spanning tree. In obtaining the second best minimum spanning tree, there must be some cut of a single vertex away from the rest for which the edge that is added is not light, otherwise, we would find the minimum spanning tree, not the second best minimum spanning tree. Call the edge that is selected for that cut for the second best minimum spanning tree (x, y). Now, consider the same cut, except look at the edge that was selected when obtaining T, call it (u, v). Then, we have that if consider $T - \{(u, v)\} \cup \{(x, y)\}$, it will be a second best minimum spanning tree. This is because if the second best minimum spanning tree also selected a non-light edge for another cut, it would end up more expensive than all the minimum spanning trees. This means that we need for every cut other than the one that the selected edge was light. This means that the choices all align with what the minimum spanning tree was.

c. We give here a dynamic programming solution. Suppose that we want to find it for (u, v). First, we will identify the vertex x that occurs immediately after u on the simple path from u to v. We will then make $\max[u, v]$ equal to the max of $w((u, x))$ and $\max[w, v]$. Lastly, we just consider the case that u and v are adjacent, in which case the maximum weight edge is just the single edge between the two. If we can find x in constant time, then we will have the whole dynamic program running in time $O(V^2)$, since that's the size of the table that's being built up. To find x in constant time, we preprocess the tree. We first pick an arbitrary root. Then, we do the preprocessing for Tarjan's off-line least common ancestors algorithm (See problem 21-3). This takes time just a little more than linear, $O(|V|\alpha(|V|))$. Once we've computed all the least common ancestors, we can just look up that result at some point later in constant time. Then, to find the w that we should pick, we first see if $u = \text{LCA}(u, v)$ if it does not, then we just pick the parent of u in the tree. If it does, then we flip the question on its head and try to compute $\max[v, u]$, we are guaranteed to not have this situation of $v = \text{LCA}(v, u)$ because we know that u is an ancestor of v.

d. We provide here an algorithm that takes time $O(V^2)$ and leave open if there exists a linear time solution, that is a $O(E + V)$ time solution. First, we find a minimum spanning tree in time $O(E + V \lg(V))$, which is in $O(V^2)$. Then, using the algorithm from part c, we find the double array max. Then, we take a running minimum over all pairs of vertices u, v, of the value of $w(u, v) - \max[u, v]$. If there is no edge between u and v, we think of the weight being infinite. Then, for the pair that resulted in the minimum value of this difference, we add in that edge and remove from the minimum spanning tree, an edge that is in the path from u to v that has weight $\max[u, v]$.

Problem 23-2 Minimum spanning tree in sparse graphs

> For a very sparse connected graph $G = (V, E)$, we can further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with Fibonacci heaps by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, we choose, for each vertex u, the minimum-weight edge (u, v) incident on u, and we put (u, v) into the minimum spanning tree under construction. We then contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, we first identify sets of vertices that are united into the same new vertex. Then we create the graph that would have resulted from contracting these edges one at a time, but we do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph may be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

Initially, we set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, we initialize the attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. We use the $orig$ attribute to reference the edge from the initial graph that is associated with an edge in the contracted graph. The c attribute holds the weight of an edge, and as edges are contracted, we update it according to the above scheme for choosing edge weights. The procedure MST-REDUCE takes inputs G and T, and it returns a contracted graph G' with updated attributes $orig'$ and c'. The procedure also accumulates edges of G into the minimum spanning tree T.

```
MST-REDUCE(G, T)
    for each v ∈ G.V
        v.mark = false
        MAKE-SET(v)
    for each u ∈ G.V
        if u.mark == false
            choose v ∈ G.Adj[u] such that (u, v).c is minimized
                UNION(u, v)
                T = T ∪ {(u, v).orig}
                u.mark = v.mark = true
    G'.V = {FIND-SET(v): v ∈ G.V}
    G'.E = Ø
    for each (x, y) ∈ G.E
        u = FIND-SET(x)
        v = FIND-SET(y)
        if (u, v) ∉ G'.E
             G'.E = G'.E ∪ {(u, v)}
             (u, v).orig' = (x, y).orig
             (u, v).c' = (x, y).c
        else if (x, y).c < (u, v).c'
             (u, v).orig' = (x, y).orig
             (u, v).c' = (x, y).c
    construct adjacency lists G'.Adj for G'
    return G' and T
```

a. Let T be the set of edges returned by MST-REDUCE, and let A be the minimum spanning tree of the graph G' formed by the call $\text{MST-PRIM}(G', c', r)$, where c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G.

b. Argue that $|G'.V| \le |V|/2$.

c. Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (Hint: Use simple data structures.)

d. Suppose that we run k phases of MST-REDUCE, using the output G' produced by one phase as the input G to the next phase and accumulating edges in T. Argue that the overall running time of the k phases is $O(kE)$.

> **e.** Suppose that after running k phases of MST-REDUCE, as in part (d), we run Prim's algorithm by calling $\text{MST-PRIM}(G', c', r)$, where G', with weight attribute c', is returned by the last phase and r is any vertex in $G'.V$. Show how to pick k so that the overall running time is $O(E \lg\lg V)$. Argue that your choice of k minimizes the overall asymptotic running time.
>
> **f.** For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

a. We'll show that the edges added at each step are safe. Consider an unmarked vertex u. Set $S = \{u\}$ and let A be the set of edges in the tree so far. Then the cut respects A, and the next edge we add is a light edge, so it is safe for A. Thus, every edge in T before we run Prim's algorithm is safe for T. Any edge that Prim's would normally add at this point would have to connect two of the trees already created, and it would be chosen as minimal. Moreover, we choose exactly one between any two trees. Thus, the fact that we only have the smallest edges available to us is not a problem. The resulting tree must be minimal.

b. We argue by induction on the number of vertices in G. We'll assume that $|V| > 1$, since otherwise MST-REDUCE will encounter an error on line 6 because there is no way to choose v. Let $|V| = 2$. Since G is connected, there must be an edge between u and v, and it is trivially of minimum weight. They are joined, and $|G'.V| = 1 = |V|/2$.

Suppose the claim holds for $|V| = n$. Let G be a connected graph on $n + 1$ vertices. Then $G'.V \le n/2$ prior to the final vertex v being examined in the for-loop of line 4. If v is marked then we're done, and if v isn't marked then we'll connect it to some other vertex, which must be marked since v is the last to be processed.

Either way, v can't contribute an additional vertex to $G'.V$. so

$$|G'.V| \le n/2 \le (n + 1)/2.$$

c. Rather than using the disjoint set structures of chapter 21, we can simply use an array to keep track of which component a vertex is in. Let A be an array of length $|V|$ such that $A[u] = v$ if $v = \text{FIND-SET}(u)$. Then $\text{FIND-SET}(u)$ can now be replaced with $A[u]$ and $\text{UNION}(u, v)$ can be replaced by $A[v] = A[u]$. Since these operations run in constant time, the runtime is $O(E)$.

d. The number of edges in the output is monotonically decreasing, so each call is $O(E)$. Thus, k calls take $O(kE)$ time.

e. The runtime of Prim's algorithm is $O(E + V \lg V)$. Each time we run MST-REDUCE, we cut the number of vertices at least in half. Thus, after k calls, the number of vertices is at most $|V|/2^k$. We need to minimize

$$E + V/2^k \lg(V/2^k) + kE = E + \frac{V \lg V}{2^k} - \frac{Vk}{2^k} + kE$$

with respect to k. If we choose $k = \lg\lg V$ then we achieve the overall running time of $O(E \lg\lg V)$ as desired.

To see that this value of k minimizes, note that the $\frac{Vk}{2^k}$ term is always less than the kE term since $E \ge V$. As k decreases, the contribution of kE decreases, and the contribution of $\frac{V \lg V}{2^k}$ increases. Thus, we need to find the value of k which makes them approximately equal in the worst case, when $E = V$. To do this, we set $\frac{\lg V}{2^k} = k$. Solving this exactly would involve the Lambert W function, but the nicest elementary function which gets close is $k = \lg \lg V$.

f. We simply set up the inequality

$$E \lg \lg V < E + V \lg V$$

to find that we need

$$E < \frac{V \lg V}{\lg \lg V - 1} = O(\frac{V \lg V}{\lg \lg V}).$$

Problem 23-3 Bottleneck spanning tree

> A ***bottleneck spanning tree*** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.
>
> **a.** Argue that a minimum spanning tree is a bottleneck spanning tree.
>
> Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show how to find a bottleneck spanning tree in linear time.
>
> **b.** Give a linear-time algorithm that given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.
>
> **c.** Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. ($Hint$: You may want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 23-2.)

a. To see that every minimum spanning tree is also a bottleneck spanning tree. Suppose that T is a minimum spanning tree. Suppose there is some edge in it (u, v) that has a weight that's greater than the weight of the bottleneck spanning tree. Then, let V_1 be the subset of vertices of V that are reachable from u in T, without going though v. Define V_2 symmetrically. Then, consider the cut that separates V_1 from V_2. The only edge that we could add across this cut is the one of minimum weight, so we know that there are no edge across this cut of weight less than $w(u, v)$.

However, we have that there is a bottleneck spanning tree with less than that weight. This is a contradiction because a bottleneck spanning tree, since it is a spanning tree, must have an edge across this cut.

b. To do this, we first process the entire graph, and remove any edges that have weight greater than b. If the remaining graph is connected, we can just arbitrarily select any tree in it, and it will be a bottleneck spanning tree of weight at most b. Testing connectivity of a graph can be done in linear time by running a breadth first search and then making sure that no vertices remain white at the end.

c. Write down all of the edge weights of vertices. Use the algorithm from section 9.3 to find the median of this list of numbers in time $O(E)$. Then, run the procedure from part b with this median value as input. Then there are two cases:

First, we could have that there is a bottleneck spanning tree with weight at most this median. Then just throw away the edges with weight more than the median, and repeat the procedure on this new graph with half the edges.

Second, we could have that there is no bottleneck spanning tree with at most that weight. Then, we should run a procedure similar to problem 23-2 to contract all of the edges that have weight at most the weight of the median. This takes time $O(E)$ and then we are left solving the problem on a graph that now has half the edges.

Observe that both cases are $O(E)$ and each recursion reduces the problem size into half. The solution to this recurrence is therefore linear.

Problem 23-4 Alternative minimum-spanning-tree algorithms

> In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.
>
> **a.**
>
> ```
> MAYBE-MST-A(G, w)
>     sort the edges into nonincreasing order of edge weights w
>     T = E
>     for each edge e, taken in nonincreasing order by weight
>         if T - {e} is a connected graph
>             T = T - {e}
>     return T
> ```
>
> **b.**
>
> ```
> MAYBE-MST-B(G, w)
>     T = Ø
>     for each edge e, taken in arbitrary order
>         if T ∪ {e} has no cycles
>             T = T ∪ {e}
>     return T
> ```

> **c.**
>
> ```
> MAYBE-MST-C(G, w)
>     T = Ø
>     for each edge e, taken in arbitrary order
>         T = T ∪ {e}
>         if T has a cycle c
>             let e' be a maximum-weight edge on c
>             T = T - {e}
>     return T
> ```

a. This does return an MST. To see this, we'll show that we never remove an edge which must be part of a minimum spanning tree. If we remove e, then e cannot be a bridge, which means that e lies on a simple cycle of the graph. Since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to that of e. By exercise 23.1-5, there is a minimum spanning tree on G with edge e removed.

To implement this, we begin by sorting the edges in $O(E \lg E)$ time. For each edge we need to check whether or not $T - e$ is connected, so we'll need to run a DFS. Each one takes $O(V + E)$, so doing this for all edges takes $O(E(V + E))$. This dominates the running time, so the total time is $O(E^2)$.

b. This doesn't return an MST. To see this, let G be the graph on 3 vertices a, b, and c. Let the eges be (a, b), (b, c), and (c, a) with weights $3, 2$, and 1 respectively. If the algorithm examines the edges in their order listed, it will take the two heaviest edges instead of the two lightest.

An efficient implementation will use disjoint sets to keep track of connected components, as in MST-REDUCE in problem 23-2. Trying to union within the same component will create a cycle. Since we make $|V|$ calls to MAKESET and at most $3|E|$ calls to FIND-SET and UNION, the runtime is $O(E\alpha(V))$.

c. This does return an MST. To see this, we simply quote the result from exercise 23.1-5. The only edges we remove are the edges of maximum weight on some cycle, and there always exists a minimum spanning tree which doesn't include these edges. Moreover, if we remove an edge from every cycle then the resulting graph cannot have any cycles, so it must be a tree.

To implement this, we use the approach taken in part (b), except now we also need to find the maximum weight edge on a cycle. For each edge which introduces a cycle we can perform a DFS to find the cycle and max weight edge. Since the tree at that time has at most one cycle, it has at most $|V|$ edges, so we can run DFS in $O(V)$. The runtime is thus $O(EV)$.