Solution 1. The algorithm correctly tests if the input tree T is a binary search tree or not. We prove its correctness using induction on h, the height of T.

The base case is when $h = 0$. This happens when T has a single node. The input trivially has the binary search tree property and the algorithm correctly returns True.

For the inductive step, we assume the algorithm is correct on trees of height $h \geq 0$ and our job is to show that it works for trees of height $h + 1$. So let us consider an arbitrary tree T of height $h + 1$. If the algorithm returns False, the vertex u that caused the algorithm to return False and its left or right descendant is evidence that T does not have the binary search property, and so the algorithm is correct. If the algorithm returns True, let r be the root of T and T_L and T_R be the right and left subtrees of r. Notice that the T_L and T_R have height at most h and calling the algorithm on those subtrees also returns True, since the checks done by TEST-BST(T_L) and TEST-BST(T_R) are a subset of the checks done by TEST-BST(T). Therefore, by our inductive hypothesis, T_L and T_R are binary search trees. That means that RIGHTMOST-DESC($T_L.root$) has the largest key in T_L and LEFTMOST-DESC($T_R.root$) has the smallest key in T_L. Therefore, we can conclude that r is larger than all the keys in T_L and smaller than all the keys in T_R, which means that T has the binary search tree property.

Solution 2. a) Our data structure will use two heaps to keep track of the pinning interval (x, y): A min-heap H_x for x and a max-heap H_y for y. At each point in time, we will keep the invariant that H_x.TOP() $= x$ and H_y.TOP() $= y$. We also use a variable L to keep track of how many keys are there in the union inside the pinning interval (x, y), not including the endpoints x and y.

The items stored in the heaps are tuples of the form $(A_i[j], i, j)$ where the first coordinate is the priority of the entry, the second coordinate $1 \leq i \leq k$ identifies one of the arrays, and the third coordinate $0 \leq j < |A_i|$ is the index of the key in the first coordinate in A_i.

To initialize the data structure, we build H_x from the items $\{(A_i[0], i, 0) : 0 \leq i < k\}$ and H_y from the items $\{(A_i[|A_i| - 1], i, |A_i| - 1) : 0 \leq i \leq k\}$. We also set $L = n - 2$.

Before performing LEFT-FORWARD() or RIGHT-BACKWARD(), we check if $L > 0$; if we attempt to move forward or backward when $L = 0$ we return an error message, since it is would not be possible to maintain a pinning interval with $x < y$. To implement LEFT-FORWARD() we call H_x.REMOVE-MIN() to get, say, $(A_i[j], i, j)$. If $j = |A_i| - 1$, we do nothing, otherwise, we add $(A_i[j + 1], i, j + 1)$ to H_x. Similarly, to implement RIGHT-BACKWARD() we call H_y.REMOVE-MAX() to get, say, $(A_i[j], i, j)$ and add $(A_i[j - 1], i, j - 1)$ to H_y if $j > 0$. In both cases, we decrement L by 1.

b) To prove the correctness we expand the invariant a bit. In addition to H_x.TOP() $= x$, we argue that H_x contains for each array A_i the smallest key such that $A_i[j] \geq x$, or no key from that array if $x > A_i[|A_i| - 1]$. Similarly, H_y contains the largest key $A_i[j] \leq y$ for each array. We focus our attention on proving that the invariant for H_x as the argument for H_y is essentially symmetric.

The invariant clearly holds after the initialization step since we create H_x with the first element of each array. Suppose that when we execute LEFT-FORWARD(), the top element in H_x comes from A_i, then we remove that item from the heap, and replace it with the next element (if any) from A_i, which is the next larger element in A_i by virtue of the array being sorted. Therefore, if the invariant holds before the execution, it holds afterwards.

c) The construction time can be bounded as follows: A heap of size k can be constructed in $O(k)$ time, so INIT runs in $O(k)$ time.

The remove operations can be bounded as follows: Removing the minimum element $(A_i[j], i, j)$ from a heap takes $O(\log k)$ time. We can find the next item to insert in $O(1)$ time and inserting the new element into the heap takes $O(\log k)$ time.

Finally, we take a brief look at what happens for different values of k (this part is not required in your solution). On the one extreme, we could have a single list (or a constant number of lists). In this case, our heaps would contain $O(1)$ elements and all its operations take $O(1)$ time, since k is $O(1)$. The other extreme case is when each element is stored in a separate list, so we have n lists. In this case we end up with two heaps that contain all elements and the operations take $O(\log n)$ time as usual. Our analysis give us a nice way to interpolate between these extremes.

Solution 3.

a) Given a connected graph $G = (V, E)$, our algorithm computes the disrupting power of a given vertex u. First, we remove u from G and then run DFS on the resulting graph. Let $T_1, T_2, \ldots, T_k$ be the trees in the DFS forest, and $C_1, C_2, \ldots, C_k$ the corresponding vertex set of each of these trees. If $k = 1$ then u is not a cut vertex and thus its disrupting power is 0. Otherwise, if $k > 1$, we return $\frac{(\sum_i |C_i|)^2 - \sum_i |C_i|^2}{2}$.

b) The correctness hinges on a simple counting argument. We count the number of pairs of nodes $x, y \in V - u$ that are disconnected in $G[V - u]$ by grouping them into the connected components they belong to. For example, there are $|C_i| \cdot |C_j|$ pairs (x, y) of disconnected nodes in $G[V - u]$ where $x \in C_i$ and $y \in C_j$. Therefore, the total number of disconnected pairs is $\sum_{i<j} |C_i| \cdot |C_j|$

The algorithm uses an alternative and equivalent expression that can be evaluated faster: $\frac{(\sum_i |C_i|)^2 - \sum_i |C_i|^2}{2}$. The correctness of this formula is easily verified by simple algebra

$$\frac{(\sum_i |C_i|)^2 - \sum_i |C_i|^2}{2} = \frac{(\sum_{i,j} |C_i| \cdot |C_j|) - \sum_i |C_i|^2}{2} = \frac{\sum_{i \neq j} |C_i| \cdot |C_j|}{2} = \sum_{i<j} |C_i| \cdot |C_j|.$$

c) Assuming the graph is represented using adjacency lists, removing the vertex u from G takes $O(\deg(u))$ time, which is $O(n)$ in the worst case. Running DFS on $G[V - u]$ takes $O(n + m)$ time since $G[V - u]$ is a subgraph of G. From the resulting DFS forest, we can identify the individual tress and their sizes $|C_1|, |C_2|, \ldots, |C_k|$ using the techniques developed earlier in the class for working

with trees. Given those values evaluating $\frac{(\sum_i |C_i|)^2 - \sum_i |C_i|^2}{2}$ can be done trivially in $O(k)$ time, which is $O(n)$.