Solution 1.
To determine if a binary tree is a binary search tree (BST), we need to ensure that for any arbitrary node u, its key must be greater than the keys of all nodes in its left subtree, and smaller than the keys of all nodes in its right subtree. To verify these conditions, we can check if the key of u is greater than the maximum key in its left subtree, and smaller than the minimum key in its right subtree.

The Test-BST(T) function aims to optimise the above approach by assuming that the rightmost descendant of the left subtree contains the maximum key value, and the leftmost descendant of the right subtree contains the minimum key value.

I believe that the optimisation used in the Test_BST(T) algorithm is correct because **(invariant)** the assumption for finding the maximum/minimum) keys of left/right subtrees will always be true, given that the subtrees are already BSTs. Algorithm Test-BST(T) in fact checks for every node in T that their left and right subtrees are BSTs by going through the following steps (however, not in the exact order as the pseudo-code may check the previous assumptions later in the loop.).

Step 1. Algorithm sees trees rooted at external nodes (left and right subtree having height of 0) to be BST. This is true because an external node does not have descendents to violate the BST property.

Step 2. Algorithm sees trees rooted at internal nodes with (left / right) subtree having height of 1 to be BST if and only if (rightmost_desc(u.left).key < u.key / leftmost_desc(u.right).key > u.key). Since there is only one node in subtree with height of 1, (rightmost_desc(u.left) is u.left itself / leftmost_desc(u.right) is u.right itself). It is easy to picture that tree rooted at u will be BST if (u.left.key < u.key / u.key < u.right.key)

Step 3. Algorithm sees trees rooted at internal nodes with (left / right) subtree having height of 2 and beyond to be BST if its (left / right) subtree is BST. and when maximum(u.left) < u.key / minimum(u.right) < u.key) holds true. Since (left / right) subtree has been checked (or will be checked eventually) to be BST, (maximum(u.left) is synonymous with rightmost_desc(u.left) / minimum(u.right) is synonymous with leftmost_desc(u.right)).

Applying the principle of induction, given that Test-BST(T) returns true, we can establish that nodes of T with subtrees of height 0 or 1 are indeed BSTs (base case). Consequently, nodes with subtrees of height 2 (as per step 3) are also BSTs. Since nodes with subtrees of height 2 are BSTs, the same holds true for nodes with subtrees of height 3. This pattern can be extended to nodes with subtrees of height 4, 5, 6, and so on, ensuring that every node in T will be checked for being a BST as long as Test-BST(T) returns true.

After our proof that Test-BST(T) returns true only when all nodes in T are BSTs, we have confirmed the correctness of Test-BST(T)'s approach for finding the maximum/minimum keys in the left/right subtrees of an arbitrary node. This validates that the given pseudocode, Test-BST(T), is a robust and potentially faster alternative to the approach outlined in the first paragraph, which determines if a tree is a BST or not based on the definition of BST.

Solution 2.

a) Design

We'll be building our data structure on top of the heap-based priority queue implementation we learned in lecture 5. In our extended implementation, each node in the min/max priority queues will have a key-value pair. The **key** will hold the value of a single item from any sorted array, while the **value** will store two pieces of information: a pointer to the array the key came from, and the index of the key within that array. Additionally, we've added new functionality to the remove operations. During a **remove-min()** operation, after the usual operation, if the removed node has a next item in the array it came from, we insert() a new node with the key value of the next item and a value that holds the pointer to the same array and an incremented index. Similarly, during a **remove-max()** operation, if the previous item exists from the array the removed node has come from, we insert() a new node with that key and the corresponding pointer and (decremented) index.

During initialisation(), we create two heap-based priority queues. The first priority queue, called **x-candidate-heap**, supports min() and extended remove-min() operation, while the second priority queue, called **y-candidate-heap**, supports max() and extended remove-max() operation. To initialise them, we insert the first elements of the sorted arrays into x-candidate-heap and the last elements to the y-candidate-heap using heap sort (heapify). We then declare and initialise **x** and **y**, which together form the pinning pair, by performing extended remove-min() of x-candidate-heap and extended remove-max() of y-candidate-heap and storing removed node's key from the former operation as x and the latter as y.

During left-forward(), we inspect the top node of x-candidate-heap using the min() operation. If the key of the top node (**x`**) is less than y, it means that x` can be a suitable new candidate for updating x because the pinning pair property (x < y) will still be satisfied. Therefore, we can safely update x by assigning it the value of x`, and then remove it from the heap using the extended remove-min() operation. However, if x` is equal to or greater than y, which would violate the pinning pair property, we don't (want to) make any changes to x or to x-candidate-heap, and would rather raise an exception to indicate that an attempt to violate the pinning pair property was made.

During right-backward(), we inspect the top node of y-candidate-heap using the max() operation. We then check if the key of the top node (y`) is greater than x before updating y with y` for the exact same reason outlined in left-forward()'s description. If it's been checked that the pinning pair property holds, we update y by y`, and then remove the top node from the heap using the extended remove-max() operation. Otherwise, no action is taken and an exception is raised.

b) Correctness

Our **invariant** states that the top node of x-candidate-heap and y-candidate-heap respectively contains the next immediate candidates for x and y of the pinning pair. This is maintained throughout initialisation and subsequent operations, as the heaps always contain the correct candidate pool for x and y. At initialisation, the candidates for x and y are the first and last items of the sorted arrays respectively, since any other item in the array cannot be strictly smaller/greater than the first/last item. During left-forward() and right-backward() operations, the candidate pool is kept correct by considering the next/previous item in the originating array of the removed node from x-candidate-heap/y-candidate-heap as the next possible immediate candidate (by inserting it as node).

The definition of the min/max heap guarantees that the node with the smallest/largest key is at the top of the x-candidate-heap/y-candidate-heap. Furthermore, the left-forward() and right-backward()

operations verify that the pinning pair property (x < y) is satisfied. This property ensures that the new data structure adheres to the given requirements, as x and y are always updated based on the key of the returned node from the extended remove-min() and remove-max() operations, and these operations maintain the truth of our invariant.

c) Time / Space Complexity

The time complexity of the **initialization()** function is O(k log k) since it requires creating two heaps of size k. Creating a heap of size k involves k insertions, and each insertion can take up to O(log k) due to the upheap operation, so the worst-case time complexity is O(k log k). However, this is not as efficient as desired according to the guidelines, which made me hesitant to use heaps initially. After some research, I discovered that heaps can be built in **O(k)** time using the heapify approach. Heapify works by taking an unsorted array and organising it into a heap in place, so that the heap property is satisfied. This process involves starting from the middle of the array and working backwards to the beginning, ensuring that each subtree is a valid heap. Since the heapify process only needs to traverse each element in the array once, it has a time complexity of O(k).

source: GeeksforGeeks. (2023). Heap Sort. Retrieved April 24, 2023, from https://www.geeksforgeeks.org/heap-sort/

The time complexity of **left-forward()** and **right-backward()** functions are both **O(log k)** because they include an extended remove-min() or remove-max() operation, along with other operations that take constant time, such as storing values in a variable and branching conditions. The extended remove-min() and remove-max() operation requires O(log k) time since it involves the remove-min() and remove-max() operations as discussed in the lecture that takes O(log k) time, followed by a possible insert() operation, which can take up to O(log k) time.

Our data structure has a **space complexity** of **O(k)** because it stores two heaps, each of which has a maximum of k nodes at any given time. Since each node in the heaps contains a key, a fixed-size array pointer, and an integer index, it takes constant space. The entire data structure consists of these two heaps and two variables, x and y, which take up trivial space. Therefore, the total space used by our data structure is O(k).

Solution 3.

a. **Algorithm description**

The adjacency list data structure is assumed to represent the graph G = (V, E), as discussed in lectures.

To evaluate the risk factor of vertex u in graph G, we need to temporarily forget about u and its connections to other vertices. So, we make copies of the lists of all vertices and edges in G and call them **VList** and **EList**. Then, we remove vertex u from VList and any edge in EList that has u as one of its ends. Additionally, we remove the pointers to the removed edge at the opposite vertex of u.

After "forgetting" the vertex u, we will use VList and EList to perform depth-first search (DFS(G)). During this operation, we will keep track of the sizes of each tree that is traversed by DFS_visit(G). To achieve this, we will modify DFS_visit(G) to take an additional argument called **tree-size**, which is initially set to 1. Whenever DFS_visit(G, tree-size) is called recursively, we will pass in the incremented value of tree-size (i.e., tree-size + 1). The DFS_visit(G, tree-size) function will return the tree-size of the subtree it just traversed. This value will be added to a list called **tree-sizes** declared at the start of DFS(). At the end of DFS(G), instead of returning a list of parents, we will return the tree-sizes list, which contains the sizes of all the trees traversed during the operation.

Now, the **risk-factor** of vertex u can be determined through simple checks and calculations on the list returned by DFS(G) called tree-sizes. First, we calculate the sum of all items in tree-sizes and assign it to variable n. Then, we set the initial value of the risk-factor to 0. We iterate through each item in tree-sizes and add to the risk-factor the result of $item * (n - item)$. Finally, we return the risk-factor value divided by two as the risk factor of vertex u in graph G.

b. **Correctness**

Our **invariant** states that DFS(G) correctly returns the size of trees in G. This is because DFS(G) visits connected vertices (trees) in one go before moving on to the next group of connected vertices. The extended helper function DFS_visit(G, tree-size) recursively visits unvisited and connected vertices and counts the vertices it visits within each tree, using an additional argument called tree-size that starts at 1 and is incremented for further recursive calls. This method guarantees that each vertex in the tree is counted exactly once, and every tree's size in forest created by traversal of G by depth-first search is recorded in a list to be used for future usage.

Assuming the invariant is true and hence the extended DFS algorithm has provided us with the correct list containing sizes of trees in G, we can apply our risk-factor calculation method to determine the risk factor of node u. The risk factor is determined by considering all the possible edges that an arbitrary vertex can form with vertices in other trees, as defined in the assessment specification.

A naive approach would involve using a nested loop to iterate over all the vertices in G and counting the number of vertices that belong to other trees in the inner loop. This can be computationally expensive, taking O(n^2) time. However, we can optimise this approach by realising that the number of vertices located in other trees is equal to the total number of vertices minus the number of vertices in the current tree. Therefore, we can simply subtract the size of the current tree from the total number of vertices to get the number of vertices in other trees. By using this simplified approach, we can efficiently and accurately calculate the total number of possible edges that an arbitrary node in a tree can form with vertices in other trees.

It is important to note that the risk factor must be divided by two before being returned, as the graph G is undirected, and an edge connecting vertices A to B is the same as an edge connecting B to A. We also want to inform you that this method gracefully handles the situation when the length of the list that holds tree sizes is 1 indicating that G remains connected and the risk-factor must be 0. In this case, because the loop runs once and the total number of vertices equals the number of vertices in the first and only tree, only 0 is added to the initial risk-factor value of 0 and hence the algorithm returns 0 for the risk factor as expected.

c. **Time Complexity**

The algorithm's time complexity is O(n + m).

In the first step, the algorithm copies the original lists of vertices and edges into VList and EList, which takes O(n) and O(m) time respectively, for a total of O(n + m) time.

The next step involves removing vertex v from VList, removing pointers in vertices of VList connected with u by an edge, and removing the edges from EList that had u in one end, which takes O(m) time at most since v could have at most m edges connecting it with other vertices. This is because the adjacency list implementation stores pointers in both the vertices and edges lists, allowing us to follow the pointers and sequentially remove the necessary items in O(m) time. Even if we scan through the entire VList and EList in the most inefficient way possible, the time complexity is still asymptotically optimal at O(n + m).

The following stage of the algorithm involves an extended DFS operation, which keeps track of a numeric variable and appends its value to a list. These operations take constant time. Since DFS takes O(n+m) time at worst, this stage also takes O(n+m) time.

In the final stage of the algorithm, we work with the list of tree-sizes generated in the previous step. Since there can be at most n-1 trees in the graph when there are n vertices, the length of this list is at most n-1. Summing up all the items in the list takes O(n) time, and the subsequent constant-time calculations we perform on each item take O(n) time in the worst case. Therefore, the time complexity of this stage is O(n).

Because all 4 stages of the algorithm are asymptotically bounded by O(n + m) time, the time complexity of the algorithm is O(n + m) as required by the assessment specification.