Solution 1.

(a) The algorithm is **correct**.

High-level understanding

The algorithm:

- *@param* T a binary tree with distinct keys
- *@output boolean* true if T is a binary search tree (BST), false otherwise
- **recursively** check if u satisfies the BST property with respect to RIGHT-DESC(*u.left*) and LEFT-DESC(*u.right*)

Proof by induction

- **Base case:**
  - When u is a LEAF NODE, we cannot investigate the subtree with u being the root. This case is trivial.
  - When u is the parent of one or two LEAF NODES, i.e., u is in the second outmost layer, RIGHT-DESC(*u.left*) and LEFT-DESC(*u.right*) are the left child and right child respectively. Satisfying that condition means the SUBTREE with u being the root is BST.
- **Induction hypothesis:**
  - Suppose that for $u \in T$, the left and right subtrees of u are BST. Then, we aim to prove that the SUBTREE ROOTED AT u is also BST.
- **Inductive step:** We intend to prove that $u.key < u.right.key$ and $u.key > u.left.key$ to show its BST property. Because the left and right subtrees are BST, we can conclude that:
  - $u.left.key < \textsc{rightmost-desc}(u.left).key$
  - $u.right.key > \textsc{leftmost-desc}(u.right).key$
- Because when checking u, we have already checked if

  $u.key > \textsc{rightmost-desc}(u.left).key$ and $u.key < \textsc{leftmost-desc}(u.right).key$,

  $u.key < u.right.key$ and $u.key > u.left.key$ must be satisfied.

  Hence, the new SUBTREE ROOTED AT u is also BST.

Ultimately, we can show that for $u \in T$, the induction proof above covers all possible cases for u.

Now taking a step back to look at the overall structure, for the recursion, we use a stack to check the base case first, which is the case when $v.right = null$ or $v.left = null$ for RIGHT-MOST() and LEFT-MOST(). Then, we recursively invoke the parent of v until the TREE *rooted at the root of* T itself is BST.

Thus, the correctness of the algorithm is proven.

(b)

High-level understanding

For n distinct keys, the **tight time complexity** is $\Theta(n)$.

We will draw an analogy from the HEAPIFY() function that *builds a heap* from an array of n elements in $\Theta(n)$ time. We can thus prove that it runs in $\Theta(n)$ time.

Note that in the pictures below, the key difference is that the arrows are not exactly DOWN-HEAP() operations. But the idea is the same: the path through which we find the RIGHT-DESC() and LEFT-DESC() are EDGE-DISJOINT. Thus each node is traversed at most twice, once for RIGHT-DESC(), once for LEFT-DESC(). Hence the $\Theta(n)$ time.

Detailed proof

Proposition. *For a given vertex $u \in T$ where T is a complete binary tree, the path through which we find the* LEFT-DESC(*u.right*) *or* RIGHT-DESC(*u.left*) *are edge-disjoint.*

Proof. We prove by induction.

First, the base case where the whole binary tree consists of only one parent and one/two children. The RIGHT-DESC(*u.left*) and LEFT-DESC(*u.right*) operations are trivially edge-disjoint.

Then, the inductive hypothesis. We hypothesize that, for a complete binary tree, from $n-1 (n-1 \geq 2)$ level to n level, the traversal path for RIGHT-DESC(*u.left*) and LEFT-DESC(*u.right*) are edge-disjoint. Ultimately, we can see the **proposition** stands for complete binary trees of various levels.

To perform the inductive step, we need to prove that, for a complete binary tree, from n level to $n+1$ level, the traversal path for RIGHT-DESC(*u.left*) and LEFT-DESC(*u.right*) are edge-disjoint.

For convenience, let us denote the n-level binary tree as T_n and the root of the $n+1$-level binary tree as u. We can see that T_{n+1} is a combination of at most two T_n trees, for left T_{left} and right T_{right}.

From n **to** $n+1$, the only action we need to take is to add the EDGE(u, $T_{left}.root$) to the RIGHT-DESC(*u.left*) traversal path (See **Figure 1**), and add the EDGE(u, $T_{right}.root$) to the LEFT-DESC(*u.right*) traversal path (See **Figure 2**).

Let us first look at the right-desc(*u.left*) traversal path.

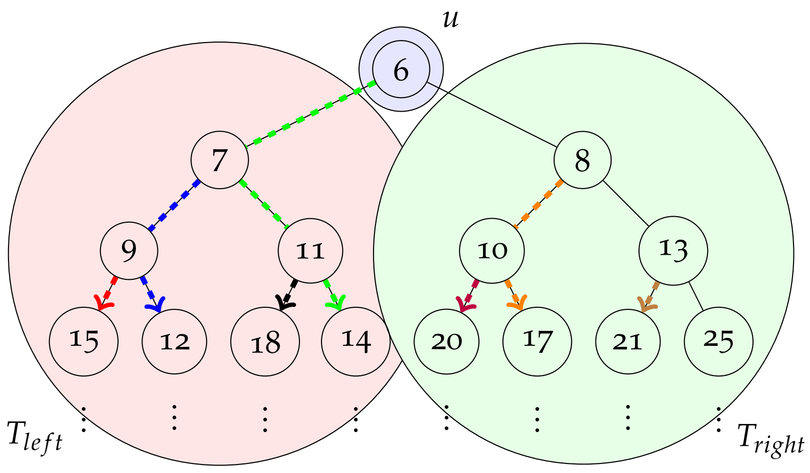

Figure 1: right-desc($u.left$)

From this example, we can see T_{right} on the right and u on the top. The only action we need to take is to attach $u = 6$ and $T_{right}.root = 7$ together.

As this only involves some fidgeting with the ROOT and u, the proof stands for all n levels and not just three levels as shown in **Figure 1**.

To generalize, for the RIGHT-DESC($u.left$) traversal path, we can see that the we only add the EDGE(u, $T_{left}.root$) to the original RIGHT-DESC($u.left$) traversal path. Because the original RIGHT-DESC($u.left$) traversal path is edge-disjoint, the new traversal path with only one edge added is also edge-disjoint. We can ignore the T_{right} subtree as it is not involved in the traversal path. And also whether the heap is a MIN-HEAP or a MAX-HEAP is not of relevance here.

Finally, if the lowest level is not fully filled, that still does not impact our proof. The only difference is that the corresponding RIGHT-DESC($u.left$) traversal path will be eliminated. The number of traversing the elements remain proportional to n.

Then, Let us first look at the left-desc($u.right$) traversal path which is in total parallel.

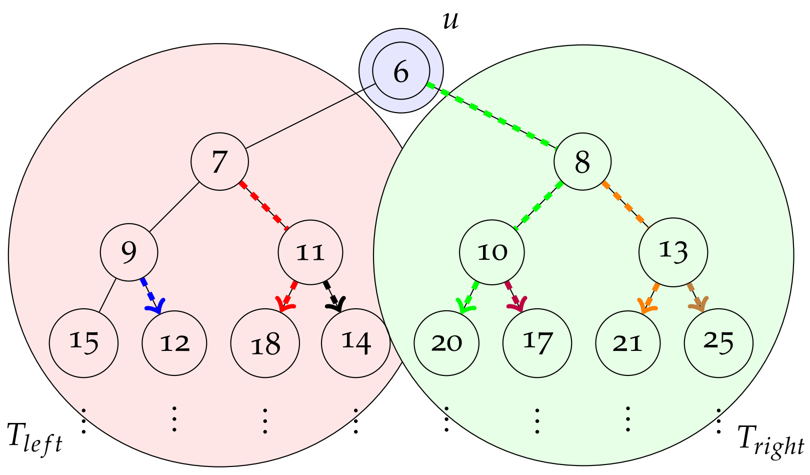

Figure 2: left-desc($u.right$)

From this example, we can see T_{left} on the left and u on the top. The only action we need to take is to attach $u = 6$ and $T_n.root = 8$ together.

As this only involves some fidgeting with the ROOT and u, the proof stands for all n levels and not just three levels as shown in **Figure 2**.

To generalize, for the LEFT-DESC($u.right$) traversal path, we can see that the we only add the EDGE(u, $T_{right}.root$) to the original LEFT-DESC($u.right$) traversal path. Because the original LEFT-DESC($u.right$) traversal path is edge-disjoint, the new traversal path with only one edge added is also edge-disjoint. We can ignore the T_{left} subtree as it is not involved in the traversal path. And also whether the heap is a MIN-HEAP or a MAX-HEAP is not of relevance here. Finally, if the lowest level is not fully filled, that still does not impact our proof.

Finally, if the lowest level is not fully filled, that still does not impact our proof. The only difference is that the corresponding LEFT-DESC($u.right$) traversal path will be eliminated. The number of traversing the elements remain proportional to n.

In conclusion for the proof, because we have the **base case** of the simplest tree that fulfills the BST property, and we have proved the inductive hypothesis, the proposition is proved. □

Conclusion

Up until now, we have proved that all the traversal path for all RIGHT-DESC($u.left$) or LEFT-DESC($u.right$) are **edge-disjoint**.

For upperbound, all vertices are traversed at **most** twice, once for RIGHT-DESC(), once for LEFT-DESC(). If on the periphery (all the way left or all the way right), they will even only be traversed once. Hence the upper-bound $O(n)$ time complexity.

For lowerbound, because all vertices are also traversed at **least** once, even for nodes on the periphery. Hence the lower-bound $\Omega(n)$ time complexity.

Therefore, the time complexity for right-desc($u.left$) or left-desc($u.right$) is

$$\Theta(n) = O(n) = \Omega(n)$$

Solution 2.

(a)

High-level understanding

Our data structure has two heaps each consisting of k elements. For simplicity, let us suppose that the array in sorted in ascending order.

In essence, we use HEAPIFY to initialize MIN-HEAP and MAX-HEAP each of k elements (refer to the "Building a Priority Queue in one go" section of *COMP2823 Lec Week 5 priority queues*). For each LEFT-FORWARD(), we get the MIN, remove it, and insert the number immediately bigger than it in that corresponding array back to the MIN-HEAP. For each RIGHT-BACKWARD(), we get the MAX, remove it, and insert the number immediately smaller than it in that corresponding array back to the MAX-HEAP.

For initialize(), we can use the HEAPIFY algorithm to create a MIN-HEAP and MAX-HEAP from all the numbers in the first and last index respectively for each array A_i where $i \in [1, k]$. That means that a MIN-HEAP for all $A_i[0]$ where $i \in [1, k]$ and a MAX-HEAP for all $A_i[len(A_i) - 1]$ where $i \in [1, k]$.

For left-forward(), we can use the REMOVE-MIN() update the heap. Then we update the pair to the new MIN() after removing the previous one. After that, we insert the number immediately bigger than CURRENT-MIN in the array that the previous MIN() was in into the MIN-HEAP.

Right-backward() follows a similar fashion.

Detailed description The discussion below is based on the assumption that the array is sorted in ascending order. *For descending order, please see section 5.*

1. initialize()

For INITIALIZE(), we extract the first and last element from each array A_i where $i \in [1, k]$ into two arrays MIN-ARRAY and MAX-ARRAY. After the extration, we augment the data structure by adding two parameters, i.e., *which-array* and *which-index* such that $A[which - array][which - index] =$ CURRENT-MIN This way, we have an object for each element containing the *value of that node*, *which-array* and *which-index* Then we use HEAPIFY to build MIN-HEAP and MAX-HEAP from these two arrays respectively. Then, we extract the MIN() and MAX() from the two heaps to return.

2. left-forward()

For LEFT-FORWARD(), we first update the HEAP to remove the current MIN() from the MIN-HEAP. Then, we extract the MIN() from the MIN-HEAP. Lastly, we insert the next largest element from the array that MAX() is in. We have already augmented the data structure of an element by adding the two parameters *which-array* and *which-index*. This means that for *current-max* $=$ MAX(), we insert

$$A[\textit{current-max.which-array}][\textit{current-max.which-index} + 1]$$

along with its two updated parameters *which-array* and *which-index* $- 1$ into the MAX-HEAP.

If the index is out of bound, we skip that process.

3. **right-backward()**

For RIGHT-BACKWARD(), we first update the HEAP to remove the current MAX() from the MAX-HEAP. Then, we extract the MAX() from the MAX-HEAP. Lastly, we insert the next smallest element from the array that MIN() is in. We have already augmented the data structure of an element by adding the two parameters *which-array* and *which-index*. This means that for *current-min* = MIN(), we insert

$$A[\textit{current-max.which-array}][\textit{current-max.which-index} - 1]$$

along with its two updated parameters *which-array* and *which-index* -1 into the MAX-HEAP.

If the index is out of bound, we skip that process.

5. **Not in ascending order**

All the processes described above still apply, except that the addition of 1 to the *which-index* will be reversed.

(b) We first argue the correctness of INITIALIZE().

Proposition. INITIALIZE() *returns a pair of the smallest and the largest number of the union.*

Proof. Because the correctness of the HEAPIFY() operation has been proved in the lectures. The burden of proof now lies in whether only selecting the first $[0]$ and the last indecies $[len(A_i) - 1]$ from each list to form the MIN/MAX-HEAP is appropriate.

Because the array is already in sorted order, we can prove by contradiction that there will be no element bigger than ones in the list extracted from first indecies. If there exists such a number, then this number should have been put in the $[0]$ position, contradicting the fact that the array is sorted in ascending order. Thus, we have proved that the MIN-HEAP is indeed the smallest k elements in the union of the arrays. The same argument can be applied to the MAX-HEAP. The correctness of REMOVE-AND-ADD-NEXT() will be discussed below to ensure all following operations run as expected.

Therefore, INITIALIZE() is correct. □

Proposition. *After* LEFT-FOWARD(), *the heap is updated* ***so that* min() *and* max() *always return the next smallest and next largest element in the union respectively****. In our analysis, the proof of correctness of the* HEAP *data structure is already covered in textbooks and lectures.*

Proof. The **proposition** is equivalent to **Statement A**.

Statement A The roots of the MIN/MAX-HEAP are always the smallest and the largest elements respectively in the union after removing visited elements S^*
* *where* $S = union - \{elements | elements\ that\ have\ been\ vistited\}$ *(union of arrays*

excluding visited elements after REMOVE-MIN() *and* REMOVE-MAX() *).*

To prove Statement A, we will prove by induction. **First, we know that as long as we prove that the smallest and the largest numbers in S are in the min/max-heap , the heap property enforces that they are the roots of the min/max-heap respectively.** In the following, we aim to prove this statement.

Base case During the INITIALIZE() process before calling the REMOVE-AND-ADD-NEXT() , we know that the smallest and the largest numbers in S are in the first [0] and last indecies $[len(A_i) - 1]$ of each array respectively. Thus the base case stands.

Inductive hypothesis When *removing* the provided visisted element and *inserting* the immediately following element, the smallest and the largest numbers in S are the roots of the MIN/MAX-HEAP respectively.

Inductive step For LEFT-FORWARD() , In a general case, after removing the visisted element from the current heap, it consists of elements of the $[i], [i+1], \ldots, [i+n]$ indecies in their respective arrays. Because the arrays are sorted, we know for sure that the next smallest element must be from the $[i]$ index of one of the arrays.

And when inserting a new element from any of the $[i+1], [i+2], \ldots, [i+n+1]$ indecies, it does not affect the existing element poised to become the next smallest element element. Thus, by the heap property, the inductive hypothesis is correct.

If the index is out of bound, we skip that process. Because that also does not affect the existing values in the heap (which are smaller), the inductive hypothesis is still correct.

A similar and parallel proof applies to RIGHT-BACKWARD() .

In conclusion, the correctness of the three functions are proven. □

(c)

For initialize() , the HEAPIFY() operation takes $O(k)$ time on $2k$ elements from the first [0] and last $[len(A_i)-]$ indecies. Returning takes $O(1)$ time.

For left-forward() , first we remove the current MIN() which takes $O(\log k)$ time. Then we return the new MIN() of the heap taking $O(1)$ time. After removal, the insertion of the two immediately following elements in their respective arrays take $O(\log k)$ time each.

For right-backward() , first we remove the current MAX() which takes $O(\log k)$ time. Then we return the new MAX() of the heap taking $O(1)$ time. After removal, the insertion of the two immediately preceding elements in their respective arrays take $O(\log k)$ time each.

In total, the running time for INITIALIZE() is $O(k)$ and for LEFT-FORWARD() and RIGHT-BACKWARD() is $O(\log k)$.

Solution 3.

(a)

High-level understanding

To draw inspiration from the problem description, we want to find all the forests (connected components) and compute the result using the formula as below.

$$\begin{aligned}\text{RISK FACTOR} &= \sum_{1\le i<j\le n} |F_i|\,|F_j| \\ &= \frac{1}{2}\left[\left(\sum_{1\le i\le n}|F_i|\right)^2 - \left(\sum_{1\le i\le n}|F_i|^2\right)\right]\end{aligned}$$

where |F| is the number of vertices in the forest F.

We first exclude u from V, getting $V-u$. Then we find all the forests inside $V-u$: F_1, F_2, …, F_n. We try to match all different forests together and compute the pairs that are *at risk*.

Detailed description

The algorithm for finding forests is shown below:

First, we iterate over V to find a set $S = \{v \in V | v \text{ is adjacent to } u\} = \{v_1, v_2, \ldots, v_m\}$ that contains all vertices immediately adjacent to u.

Then, we run a **modified Depth-first search (DFS)** on each $v \in S$ that **has not been visited**. We **mark v as visited** before continue with the next function call so that we do not visit it again. We also initialize a array *sizeOfForestArray* to store the number of vertices visited in each forest.

For each DFS($v \in S$), we do the following:
1. Mark v as visited so that we do not visit it again;
2. If there is no unvisited adjacent vertex, return 1;
3. Iterate over all adjacent vertices $v_{neighbor}$ that are not visited; for each $v_{neighbor}$, **return DFS($v_{neighbor}$)** + 1 in order to accumulate the number of vertices in a forest.

Finally, we have returned the number of vertices in the forest containing $v \in S$ in the first place. Then we will append this number to *sizeOfForestArray*.

Now that we have obtained a array of numbers *sizeOfForestArray*, we can compute the risk factor of u by the formula as below:

$$\begin{aligned}\text{RISK FACTOR} &= \sum_{1\le i<j\le n} |F_i|\,|F_j| \\ &= \frac{1}{2}\left[\left(\sum_{1\le i\le n}|F_i|\right)^2 - \left(\sum_{1\le i\le n}|F_i|^2\right)\right]\end{aligned}$$

where |F| is the number of vertices in the forest F.

This formula either (1) computes the product of all pairs of numbers in the array, or for a faster way, (2) compute the half of the sum of the differences between the square of the sum of the array and the sum of squares of every number. As proved in (c) below, **we should select option (2)** for optimal time efficiency.

(b) We first prove the correctness of the algorithm for (1) computing after finding trees, then (2) the mathematical equivalent of $= \sum_{1\le i<j\le n} |F_i| \left|F_j\right|$ and at last (c) of finding forests.

Proposition. *For a set $\{F | F \text{ is a forest in } V - u\}$,* RISK FACTOR $= \sum_{1\le i<j\le n} |F_i| \left|F_j\right|$ *where* | F | *is the number of vertices in the forest F.*

Proof. **We prove by induction.**
Base case:

Consider a single forest, by the definition of a FOREST (CONNECTED COMPONENT) of v_0, it includes v_0 and all vertices that can be reached from (connected to) v_0. Because the forest finding endeavor was attempted after removing u to obtain $V - u$, all the forests F_i that were found remains intact regardless of whether u exists or not. This means that all the vertices within the forest F_i are not *at risk* after removing u.

For any of the two forests F_i and F_j, they are not connected to each other, so all the vertices in F_i cannot be coonected to any of the vertices in F_j. In fact, only through u can they be connected.
Inductive hypothesis:

For a set of forests $S = \{F_1, F_2, \ldots, F_n\}$, when adding F_{n+1} to the set S, we hypothesize that ΔRISK FACTOR $= \sum_{1\le i\le n} |F_i||F_{n+1}|$. Here the ΔRISK FACTOR is the difference between the risk factor before adding F_{n+1} and after adding F_{n+1} for the set of forest $S \bigcup u = \{F_1, F_2, \ldots, F_n\} \bigcup u$.
Inductive step:

Now, after proving the validity of $|F_i||F_j|$ for two forests, we can prove the correctness of the algorithm by induction for any number of forests. When adding F_{n+1} to the existing $S_1 = \{F_1, F_2, \ldots, F_n\}$ to get $S_2 = S_1 \bigcup F_{n+1}$, the S_1 itself remains the same. The only difference is the new pair of vertices that are *at risk* after adding.

The most exhaustive way of doing this is to compare every vertex from F_{n+1} with every vertex from all the forests $F \in S_1$. By the definition of forest, these vertex cannot reach each other, and therefore, each pair is valid. Thus, we add ΔRISK FACTOR $= \sum_{1\le i\le n} |F_i||F_{n+1}|$ to the existing RISK FACTOR.

In the end, we have a sum of $\sum_{1\le i<j\le n} |F_i| \left|F_j\right|$. □

Proposition.

$$\sum_{1\le i<j\le n} |F_i| \left|F_j\right| = \frac{1}{2}\left[\left(\sum_{1\le i\le n} |F_i|\right)^2 - \left(\sum_{1\le i\le n} |F_i|^2\right)\right]$$

Proof.

$$
\begin{aligned}
\sum_{1\le i<j\le n} |F_i|\,|F_j| &= |F_1||F_2| + |F_1||F_3| + \cdots + |F_2||F_3| + \cdots + |F_{n-1}||F_n| && (1)\\
&= \frac{1}{2}\Big[\Big(|F_1|^2 + |F_2|^2 + \cdots + |F_n|^2\Big) && (2)\\
&\quad + (2|F_1||F_2| + 2|F_1||F_3| + \cdots + 2|F_2||F_3| + \cdots + 2|F_{n-1}||F_n|)\\
&\quad - \Big(|F_1|^2 + |F_2|^2 + \cdots + |F_n|^2\Big)\Big]\\
&= \frac{1}{2}\Big[(|F_1| + |F_2| + |F_3| + \cdots + |F_n|)^2 && (3)\\
&\quad - \Big(|F_1|^2 + |F_2|^2 + |F_3|^2 + \cdots + |F_n|^2\Big)\Big]\\
&= \frac{1}{2}\left[\left(\sum_{1\le i\le n} |F_i|\right)^2 - \left(\sum_{1\le i\le n} |F_i|^2\right)\right] && (4)
\end{aligned}
$$

□

Proposition. *The* ***modified*** *DFS algorithm takes input: V and u, and output: an array $sizeOfForestArray$ of sizes of forests, where $sizeOfForestArray[i-1]$ is the number of vertices in the i-th forest in $V - u$.*

Proof. **First,** we prove that it correctly traverses a forest with no overlap with previous forests, and ultimately covers all vertices in $V - u$. **This quesiton can be further divided into two parts: (1) no overlap (2) no vertex in $V - u$ is left out.**

For (1) no overlap, we prove by contradiction. Suppose there is an overlap, then there must be a vertex v that is reachable from two different vertices v_1 and v_2. Then when traverse from v_1 to v ($v_1 \mapsto v$), it will eventually lead to v_2. That means that v, v_1, v_2 must in the forest F_i. Thus a contradiction occurs, the algorithm will not traverse v repeatedly and there is no overlap.

For (2) no vertex in $V - u$ is left out, we prove by contradiction. Suppose there is a vertex v that is not traversed by the algorithm, then there v must not be connected to any other vertices. **If v is adjacent to u,** it must have been traversed even though the forest containing it may be a single vertex v. **If, however, v is not adjacent to u,** then because the graph G is connected, there must have been some vertex that leads to v, which will record v in a forest. In either case, there is a contradiction. Thus, no vertex in $V - u$ is left out.

Up until this point, we have proved that it correctly traverses a forest with no overlap with previous forests, and ultimately covers all vertices in $V - u$. The rest is trivial, as we simply return the **return result of the recursive call** $+1$, achieving the goal of counting the number of vertices.

Finally, we stored them in an array as expected.

□

(c)

$O(n + m)$: First, we iterate over V to find a set $S = \{v \in V \mid v \text{ is adjacent to } u\} = \{v_1, v_2, \ldots, v_m\}$ that contains all vertices immediately adjacent to u.

$O(n+m)$: Next, perform DFS on all vertices $v \in S$ to find all forests and return *sizeOfForestArray*. Because we only add the MARK AS VISITED and CHECK IF VISITED feature to the DFS which run in constant time, its time complexity is the same as proved in the lectures.

$O(numberOfForests) \leq O(n)$: Finally, we iterate over *sizeOfForestArray* to compute using RISK FACTOR $= \sum_{1 \leq i < j \leq n} |F_i| |F_j| = \frac{1}{2} \left[\left(\sum_{1 \leq i \leq n} |F_i| \right)^2 - \left(\sum_{1 \leq i \leq n} |F_i|^2 \right) \right]$. We do not know NUMBEROFFORESTS in advance, but we know that it is at most $n - 1 \leq n$.

Adding up yields

$$O(n+m) + O(n+m) + O(n) = O(3n+m) \leq O(n+m)$$

as required.