Mo Felonda

CS 577 HW3

(where d is the depth of the tree and is ≥ 0)

Input: A complete binary tree T with $n = 2^d$ leaves, where each leaf contains an integer value.
Output: i, a count of the minimum possible inversions; and S, a sorted array

```
1.  procedure MINIMUM-INVERSIONS(T)
2.      if d = 0 then
3.          return 0
4.      else
5.          (i_L, L) ← MINIMUM-INVERSIONS(the subtree rooted at T's left child) (L is an array of leaves)
6.          (i_R, R) ← MINIMUM-INVERSIONS(the subtree rooted at T's right child) (R is an array of leaves)
7.          i_CROSS ← COUNT-CROSS(L,R)
8.          i_CROSS2 ← COUNT-CROSS(R,L)
9.          i ← i_L + i_R
10.         S ← MERGE(L,R)
11.         if i_CROSS ≤ i_CROSS2
12.             i ← i + i_CROSS
13.         else
14.             i ← i + i_CROSS2
15.         return (i,S)
```

Proof: Induction Hypothesis: For $n \leq K$ (where n is the number of leaves and k is all positive integers), MINIMUM-INVERSIONS returns the minimum possible number of inversions on all valid inputs.

Base Case: $d = 0$, or the tree has a depth of 0, so the number of leaves is $n = 2^0 = 1$. This is the minimum input specified by the algorithm, and if there is only one leaf in the tree, there cannot be any inversions. So by line 3, the algorithm correctly returns 0.

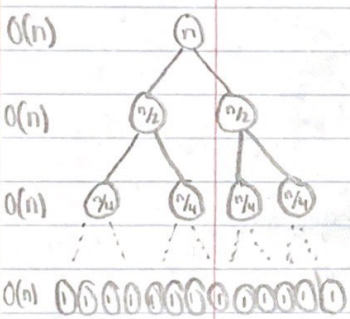
Recursive Case: When $d > 0$, or the tree has a depth greater than 0, the tree is recursively split into two arrays, with L containing the subtree rooted at T's left child, and R containing the subtree rooted at T's right child. These recursive calls also increment an inversion counter allocated to each array (i_L & i_R). We then use the subroutine COUNT-CROSS[1] to count inversions across L and R, and then alternatively across R and L, since the number of inversions can differ (i_{CROSS} & i_{CROSS2}).

[1] COUNT-CROSS subroutine is used with permission from the divide and conquer scribe notes.

HW3 Continued

Recursive Case (cont.): Then, the inversions counted when recursively sorting the arrays are added to the total counter. Following this, the arrays are concatenated by the MERGE[2] subroutine and are thus fully sorted. Most importantly, the number of inversions between each call of COUNT-CROSS is compared so that only the lesser of the two is added to the total counter. Since we only add the lesser of the two, and COUNT-CROSS has been proven in the Scribe Notes to return the correct number of inversions, we can be sure that our procedure correctly returns the minimum possible number of inversions on all valid inputs. Finally, the count of inversions is returned along with the sorted copy of the array (i, s).

Time-Complexity: We know from the divide and conquer scribe notes that the COUNT-CROSS subroutine is assumed to be linear. We also know that the MERGE subroutine is assumed to be linear as well. As such, we must only worry about the recursive call, since the rest of the procedure is simple math and comparisons done in constant time. During each call of recursion (or level, when looking at the tree) the number of comparisons is $\leq$ the sum of the sizes of all nodes in the level. All sizes are the same at each level, as the size is an equal fraction of the size of the parent node. This means that the total size is upper bounded on each level by n. Thus, the total number of comparisons at each level is at most n. Since each node is split equally among its children, the depth of the tree is $O(\log n)$. Since the tree has $O(\log n)$ levels, each with $O(n)$ comparisons, we know that the efficiency of the recursive call is $O(n \log n)$, which is thus the efficiency of the whole procedure.



Termination: Termination is implied since the depth of the tree is at most $O(\log n)$. The base case or correct number of inversions will always be returned.

[2] MERGE subroutine is used with permission from the Program Correctness scribe notes.