CS 577: Introduction to Algorithms Fall 2019

Homework 1 Solutions

Instructor: Dieter van Melkebeek TA: Andrew Morgan

Problem 1

Argue that FindDiameterAndDepth and FindDiameter (reproduced on page 3) correctly implement their specifications and run in linear time.

Since FindDiameter just passes its input to FindDiameterAndDepth and drops part of the output, its correctness and running time follow directly from that of FindDiameterAndDepth. So it remains to address FindDiameterAndDepth.

Claim 1. FindDiameterAndDepth *correctly implements its specification.*

Proof. Since FindDiameterAndDepth is a recursive algorithm, we prove its correctness by structural induction on rooted trees. Recall the construction for rooted trees (adapted to the notation of this problem):

1. A graph consisting of a single vertex is a rooted tree whose root is that vertex.
2. Let $k \in \mathbb{Z}$ with $k \geq 1$, and for each $i \in [k]$, let T_i be a rooted tree with root r_i such that the sets of vertices are pairwise disjoint, and let r be a new vertex. The disjoint union of the T_i together with the vertex r and edges between r and each r_i is a rooted tree with root r.

The inductive hypothesis for t applications of the constructor rules ($P(t)$) asserts FindDiameterAndDepth correctly implements its specification on all rooted trees T formed by at most t applications of the constructor rule.

Base case: We need to show$P(1)$. The only way to construct a rooted tree with at most one rule is to use rule 1. The depth and the diameter of such a tree are by definition zero in this case, so FindDiameterAndDepth is correct to return zeroes.

Inductive step: We need to show $P(t) \implies P(t+1)$ for all $t \in \mathbb{Z}$, $t \geq 1$. Consider a rooted tree T obtained by $t+1$ applications of the constructor rule. If the last rule applied is rule 1, we are done as in the base case. If not, let $k \geq 1$, $T_1, \ldots, T_k$, be as in rule 2. These coincide with $T_1, \ldots, T_k$ from line 8. Continuing to lines 9–10, the inductive hypothesis implies that for each i, D_i and d_i are respectively the diameter and depth of T_i.

Observe that any longest shortest path from the root of T to any other vertex starts by taking a step to the root of some T_i and then follows a longest shortest path to another vertex within T_i. It follows that the depth of T is one plus the largest depth of a T_i. This is computed as d in line 11.

As for the diameter of T, observe that each shortest path in T falls into one of two cases.

1. The shortest path stays within some T_i. A longest shortest path of this type has length equal to the diameter of some T_i.

2. The shortest path involves the root of T. In this case, the analysis depends on whether $k = 1$ or $k > 1$.
   (a) When $k = 1$, any shortest path involving the root starts or ends at the root. The longest shortest path of this type has length equal to the depth of T, which is d.
   (b) When $k > 1$, a longest shortest path involving the root passes through the root, and travels through the two deepest subtrees directly below the root of T. Its length is therefore the largest depth of a T_i, plus two (steps through the root), plus the second largest depth of a T_i.

   This covers all the cases; the diameter of T is the largest among them. We see that the diameter of T is computed as D in lines 12–13. (Setting $d' \leftarrow 0$ when $k = 1$ ensures that $d + d'$ takes the correct value in case 2(a).)

   Thus the values returned by FindDiameterAndDepth are the diameter and depth, as required, provided the recursive calls were correct. This completes the inductive step.

Correctness of FindDiameterAndDepth now follows by induction. □

Claim 2. FindDiameterAndDepth *runs in $O(n)$ time, where n is the number of vertices in T.*

Proof. We think of the algorithm as "visiting" the vertices of T, starting at the root, and traversing the rest following the recursive calls. While visiting a vertex, the algorithm does its nonrecursive work so that each piece of work done by the algorithm is done while visiting some vertex. To bound the running time, we bound the work done at any given vertex, and sum this over all the vertices.

While visiting a leaf, the algorithm performs some $O(1)$ work. While visiting a non-leaf vertex, the algorithm performs some $O(1)$ work plus $O(1)$ additional work *per child* of that vertex. Summing this work over all vertices in the tree, we get $O(1)$ times the number of nodes plus $O(1)$ for each parent-child relationship in the tree. Since there are n nodes and $n - 1$ parent-child relationships, the total work is $n \cdot O(1) + (n - 1) \cdot O(1) = O(n)$. □

(We can just as well use the "recursion tree method" to be covered in class on 9/12. Here, the shape of the recursion tree is the same as T, and the work per node is as it is above.)

Algorithm 1

Input: T, where T is a nonempty rooted tree
Output: the diameter of T

```
1: procedure FindDiameter(T)
2:     (D, d) ← FindDiameterAndDepth(T)
3:     return D
```

Input: T, where T is a nonempty rooted tree
Output: (D, d) where D is the diameter of T and d is the depth of T

```
4: procedure FindDiameterAndDepth(T)
5:     if T's root has no children then
6:         return (0, 0)
7:     else
8:         T_1, ..., T_k ← the subtrees of T rooted at the children of the root of T
9:         for i = 1 to k do
10:            (D_i, d_i) ← FindDiameterAndDepth(T_i)
11:        d ← 1+ largest d_i
12:        if k = 1 then d' ← 0 else d' ← 1+ second largest d_i including repetitions
13:        D ← max(D_1, ..., D_k, d + d')
14:        return (D, d)
```

Problem 2

Consider the problem of powering an integer:

Input: (a, b) with $a, b \in \mathbb{Z}$ and $b \geq 1$

Output: a^b, which we define as $a^b \doteq \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}}$, where "$\cdot$" denotes multiplication.

Design an algorithm for this problem that uses at most $O(\log b)$ multiplications.

We first consider the case where b is a power of two. The key idea is to treat the expression

$$\underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}}$$

like an array to be divided into halves, rewrite it as

$$(\underbrace{a \cdot a \cdot \cdots \cdot a}_{b/2 \text{ times}}) \cdot (\underbrace{a \cdot a \cdot \cdots \cdot a}_{b/2 \text{ times}}),$$

observe that the factors are equal, and so we can compute both of them with only one recursive call. Since $b/2$ remains a power of two, this is a complete algorithm for that case.

For the general case, b is an arbitrary positive integer. Still we can use a similar decomposition into halves:

$$a^b = (\underbrace{a \cdot a \cdot \cdots \cdot a}_{\lfloor b/2 \rfloor \text{ times}}) \cdot (\underbrace{a \cdot a \cdot \cdots \cdot a}_{\lceil b/2 \rceil \text{ times}}).$$

In this case, the second part is identical to the first part up to a single factor of a. The extra factor is present if and only if b is odd. So we can recursively compute the first part, square it, and then, when b is odd, multiply in one more factor of a. This gives us Algorithm 2:

Algorithm 2

Input: $a, b \in \mathbb{Z}, b \geq 1$
Output: a^b

```
1: procedure FAST-POWER(a, b)
2:     if b = 1 then
3:         return a
4:     else
5:         c ← FAST-POWER(a, ⌊b/2⌋)
6:         if b is even then
7:             return c · c
8:         else
9:             return c · c · a
```

It remains to prove that Algorithm 2 is correct, and to analyze its running time:

Claim 3. FAST-POWER *is correct.*

Proof of claim 3. The proof is by strong induction on b. The inductive hypothesis for fixed b asserts that for all $a \in \mathbb{Z}$, FAST-POWER$(a, b) = a^b$.

In the base case, $b = 1$. When this happens, FAST-POWER follows its base case, and directly returns a in line 3. This is correct.

For the inductive step, we have $b \geq 2$, and FAST-POWER drops into the recursive case starting on line 5. Since $1 \leq \lfloor b/2 \rfloor < b$, we may invoke the inductive hypothesis, from which we conclude that $c = a^{\lfloor b/2 \rfloor}$. From here, we see that when b is even, the algorithm returns $c^2 = a^{2\lfloor b/2 \rfloor} = a^b$; when b is odd, the algorithm returns $c^2 = a^{2\lfloor b/2 \rfloor + 1} = a^b$. Thus correctness at all levels less than b implies correctness at level b.

Claim 3 now follows by strong induction. □

Claim 4. FAST-POWER *uses at most* $O(\log b)$ *multiplications.*

Proof of claim 4. The complexity of FAST-POWER can be analyzed using the recursion tree method. Each invocation of FAST-POWER has at most one recursive call, so the shape of the recursion tree is a line. When moving one level down the tree, the 'b' argument is halved (rounding down). It becomes 1 after $\lfloor \log b \rfloor$ halvings (essentially by the definition of logarithm). As a result, the depth of the recursion tree is at most $\lfloor \log b \rfloor$. Each node in the decision tree does at most two (hence $O(1)$-many) multiplications. It follows that there are at most $O(1) \cdot \lfloor \log b \rfloor = O(\log b)$ multiplications in any invocation of FAST-POWER. □

Problem 3

You are given a complete binary tree T with $n = 2^d$ leaves, where each leaf contains an integer value. Reading the leaf values from left to right yields a sequence of integers. The question is how small we can make the number of inversions in that sequence by applying any number of operations of the following type: Select an internal vertex and swap the two child subtrees.

Design an $O(n \log n)$ algorithm for this problem.

We can equivalently think of the input as the array A of leaf values, and the objective as applying certain permutations on A so as to minimize the number of inversions in A. The inversions in A can be broken into three types:

(a) inversions within the left half of A,

(b) inversions within the right half of A, and

(c) inversions that cross the boundary between the two halves.

Swap operations on nodes of the left subtree of T only affect (a). Similarly, swap operations on the right subtree of T only affect (b). A swap on the root of T only affects (c). This means that in order to minimize the total count of inversions in T, we can independently minimize the counts of (a), (b), and (c).

Minimizing the counts of (a) and (b) corresponds to simpler instances of the given problem, namely for the left subtree of T and the right subtree of T, respectively; thus we can find those quantities by recursion.

Minimizing (c) is where the real work happens. Given the recursion into (a) and (b), we can afford $O(n)$ work to compute (c) and still get an $O(n \log n)$ running time.

Recall the procedure Count-Cross from class. It takes two sorted arrays as input, and, in linear time, outputs the number of inversions in the concatenation of the first and the second array. Thus, if L and R are sorted copies of the left and right halves of A, then we can compute the minimum value of (c) as the minimum of Count-Cross(L, R) and Count-Cross(R, L). So it suffices to get our hands on L and R.

To do that, we strengthen the specification of our algorithm. In addition to computing the minimum number of inversions, we require that it return a sorted copy of A. Thus, when we recurse on the left half of A and the right half of A, these recursive calls directly return L and R as well. As discussed, this allows us to minimize the counts of (a), (b), and (c), but now we must also compute a sorted copy of A. We can meet this requirement by using the procedure Merge from class. It takes as input two sorted arrays and returns their sorted concatenation. Since we have L and R, we can use Merge to compute a sorted copy of A. Merge runs in linear time.

All together, we get a divide-and-conquer algorithm for the augmented problem, where in addition to the minimum number of inversions for T we also output the leaf values in sorted order. The algorithm is given as Algorithm 3 below. The answer to the original question is the first component of the value returned by TreeInversionCount(T), where T is the original input tree.

Correctness. Correctness follows from an inductive argument on the depth of T. The base case is when $d = 0$, i.e, T has only one node. In this case, TreeInversionCount returns 0 and an

Algorithm 3 Counting Tree Inversions

Input: T, a complete binary tree with an integer at each leaf

Output: The pair (v, S), where c is the minimum number of inversions in T's leaf node values after swapping subtrees of internal nodes, and S is a sorted array of all leaf node values in T

```
1:  procedure TreeInversionCount(T)
2:      if |T| = 1 then
3:          return (0, k) where k is the integer stored at the node in T
4:      (a, L) ←TreeInversionCount(T's left subtree)
5:      (b, R) ←TreeInversionCount(T's right subtree)
6:      c_1 ←Count-Cross(L, R)
7:      c_2 ←Count-Cross(R, L)
8:      c ← min(c_1, c_2)
9:      v ← a + b + c
10:     S ←Merge(L, R)
11:     return (v, S)
```

singleton array of that element. Since a sequence of one value never has any inversions, and an array of one element is always sorted, this is the correct result.

We now establish the inductive step. Recall that in order to minimize the total number of inversions, it suffices to compute the minimum counts of inversions of types (a), (b), and (c) independently. By the inductive hypothesis, TreeInversionCount returns correctly for T's left and right children. Thus a stores the minimum count of inversions of type (a), and b stores the minimum count of inversions of type (b). L and R store sorted copies of the left and right halves of A, respectively. As discussed, the number of inversions of type (c) equals the number of inversions in LR or in RL, depending on whether a swap is done at the root. By correctness of Count-Cross, c_1 and c_2 store these respective values, and so $c = \min(c_1, c_2)$ is the minimum count of inversions of type (c). It follows that v stores the minimum number of inversions in T's leaf node values after swapping subtrees of internal nodes. Finally, by correctness of Merge, S correctly stores a sorted copy of A. Thus TreeInversionCount correctly implements its specification.

Analysis. The running time analysis is nearly identical to the analysis for Merge Sort. In the recursive case, any given execution of TreeInversionCount makes two recursive calls, two calls to Count-Cross, a call to Merge, and $O(1)$ additional work. The recursive calls of TreeInversionCount divide the leaves into equal halves. As discussed in class, Count-Cross and Merge take time linear in the sum of the lengths of their arguments; it follows that each level of the recursion tree does a combined $O(n)$ work. As there are $O(\log n)$ levels, the running time of TreeInversionCount is $O(n \log n)$.

Problem 4

Consider the following computational problem:

Input: Array $A[1, \ldots, n]$ of positive integers.

Output: Array $C[1, \ldots, n]$ where $C[i]$ is the number of $j \in \{1, \ldots, i-1\}$ with $A[j] \geq A[i]$.

Design an $O(n \log n)$ algorithm for this problem.

We use a divide-and-conquer approach, in which we break up the given array $A[1, \ldots, n]$ into two halves, $L \doteq A[1, \ldots, \lfloor n/2 \rfloor]$ and $R \doteq A[\lfloor n/2 \rfloor + 1, \ldots, n]$, recursively find the respective solutions C_L and C_R for those subproblems, and then use C_L and C_R to efficiently compute the solution C for A. Below, we show how to compute C in $O(n)$ time, given C_L and C_R. Given that, the resulting algorithm runs in time $O(n \log n)$ as it follows the same pattern as MergeSort.

The first half of C equals C_L, i.e., $C[1, \ldots, \lfloor n/2 \rfloor] = C_L[1, \ldots, \lfloor n/2 \rfloor]$. This is because the elements that precede elements in the first half of C all occur in that first half, which equals C_L.

For the second half of C, we need to add to C_R the number of elements in L that are larger than or equal to the element from A under consideration. More precisely, for $1 \leq k \leq \lceil n/2 \rceil$, we have:

$$C[\lfloor n/2 \rfloor + k] = C_R[k] + |\{i \in [\lfloor n/2 \rfloor] \text{ such that } L[i] \leq R[k]\}| \,.$$

Note that the added term equals the number of inversions formed by the k-th element of R with elements from L in the concatenation LR, provided that we also count pairs of positions that contain equal values as inversions. In class we saw a linear-time algorithm to count the aggregate of all such inversions (and sort the concatenation LR) when both L and R are sorted in nondecreasing order. We use the same procedure here with the following modifications:

- Consider cross pairs of equal values as inversions. This can be achieved by moving the pointer into R rather than the pointer into L in the case of equal values.
- Record separately the counts for each element of R rather than aggregating them.
- Keep track of the original positions of the sorted entries in the given array A. This enables us to add the cross inversions to the correct positions.

For clarity we include pseudocode for the combining process BlockersMerge. It uses records consisting of the value of an entry of the given array and the position of the entry in the given array in order to represent the given array.

For completeness we also include pseudocode for the recursive procedure Blockers that takes an array and returns the array in sorted record format as well as the solution to the problem. See pages 9–10.

Correctness Correctness of Blockers and BlockersMerge is established by organizing the ideas from the above discussion into a typical correctness proof.

To start, we establish correctness of Blockers by induction on n. The n-th statement to be established is that Blockers matches its specification on all inputs A of length at most n. As base cases, we have $n = 0$ and $n = 1$. In the former case, we are required to return two empty arrays, as the algorithm does. In the latter case, we are required to return an array containing the pair $(1, A[1])$, and an array containing 0, as the algorithm does. For $n > 1$, Blockers enters its

recursive case. Here correctness of the algorithm follows directly from the inductive hypothesis and the specification for BlockersMerge.

Finally, it remains to establish correctness of BlockersMerge. However, due to its similarity to correctness of Count-Cross from lecture/scribe notes, we omit the details here.

Running Time Lastly, we need to analyze the running time of Blockers (hence also BlockersMerge). As with Count-Cross, BlockersMerge runs in time $O(\ell + r)$. Blockers has a recursion tree shaped identically to MergeSort, as well as the same work-per-node up to constant factors, so it runs in time $O(n \log n)$.

Algorithm 4

Input: Array $A[1, \ldots, n]$ of intergers.

Output: $(C, \widehat{A})$ where C is the solution for A, and $\widehat{A}$ contains the pairs $(i, A[i])$ for $i = 1, \ldots, n$, sorted in nondecreasing order of the second component.

```
1: procedure Blockers(A[1, ..., n])
2:     if n = 0 then
3:         return ( [], [] )
4:     else if n = 1 then
5:         return ( [ (1,A[1]) ], [0] )
6:     else
7:         (C_L, L̂) ← Blockers(A[1, ..., ⌊n/2⌋])
8:         (C_R, R̂) ← Blockers(A[⌊n/2⌋ + 1, n])
9:         return BlockersMerge(L̂, R̂, C_L, C_R)
```

Algorithm 5

Input: Arrays $\widehat{L}[1,\ldots,\ell]$ and $\widehat{R}[1,\ldots,r]$ of (position, value) records corresponding to arrays L and R, respectively; $\widehat{L}$ and $\widehat{R}$ are sorted in order of nondecreasing value. Solutions $C_L[1,\ldots,\ell]$ and $C_R[1,\ldots,r]$ for the arrays L and R, respectively.

Output: $(C,\widehat{A})$ where $\widehat{A}$ is an array of (position, value) records corresponding to the concatenation LR of L and R, sorted in nondecreasing order of value, and C is the solution for LR.

1: **procedure** BLOCKERSMERGE($\widehat{L}[1,\ell]$, $\widehat{R}[1,r]$, C_L, C_R)
2: $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$
3: $C \leftarrow C_L C_R$ (concatenation)
4: **while** $j \leq r$ **do**
5: **if** $i \leq \ell$ **and** $\widehat{L}[i].value < \widehat{R}[j].value$ **then**
6: $\widehat{A}[k] \leftarrow \widehat{L}[i]$
7: $i \leftarrow i+1$
8: **else**
9: $\widehat{A}[k] \leftarrow \widehat{R}[j]$ with 'position' component increased by ℓ
10: $C[\widehat{A}[k].position] \leftarrow C[\widehat{A}[k].position] + (\ell - i + 1)$
11: $j \leftarrow j+1$
12: $k \leftarrow k+1$
13: **if** $i \leq \ell$ **then**
14: $\widehat{A}[k,\ldots,\ell+r] \leftarrow \widehat{L}[i,\ldots,\ell]$
15: **return** $(\widehat{A}[1,\ldots,\ell+r], C[1,\ldots,\ell+r])$