2nd Lt David Crow

ENG/20M

Analysis of Algorithms Homework 3

**Chapter 5, Problem 2**



**Solution**

*Overall, this problem took me 30 minutes. I’d rate it a 2 on the difficulty scale; I only needed to change one line of the book’s counting-inversions algorithm to find a suitable solution, and this change doesn’t affect the runtime or correctness of the given algorithm. I worked with Savannah Hyde.*

For this problem, we can actually just modify the book’s algorithm for counting inversions.

1 Merge-and-Count(, )

2 Maintain a pointer into each list, initialized to point

to the front elements

3 Maintain a variable for the number of versions, initialized

to

4 While we still have elements to examine in either list

5 Let and be the elements pointed to by the

pointers

6 Append the smaller of these two to the output list and

advance that pointer

7 Reset the pointers

8 While we still have elements to examine in either list

9 Let and be the elements pointed to

by the pointers

10 If then

11 Increment by the number of elements

remaining in

12 Increment the pointer for

13 Else

14 Increment the pointer for

15 Endif

18 Endwhile

19 Once one list is empty, append the remainder of the other list to

the output

20 Return and the merged list

As we can see above, we only need a few changes: we first merge the lists into our output list, and, only after this is done, we count the number of inversions. Here, instead of incrementing simply when is less than , we increment for each element such that is less than *half the value of* . With these simple changes, the algorithm now counts the number of significant inversions (as opposed to just counting inversions).

This algorithm, as stated in the book, assumes that and are sorted lists­. For our modified algorithm, under this assumption, we know the algorithm is correct because the book’s algorithm is correct (as proved by the authors). Still, I’ll restate the proof here.

Given two sorted lists, appending an element to the output indicates that is smaller than every ; thus, there are no inversions in this step. When is instead appended to the output, it must necessarily be smaller than all remaining elements in (these are standard inversions); has a *significant* inversion with each element such that is twice the value of . We’ve now covered both cases (where is added and where is added), so we now know the above algorithm correctly counts the number of significant inversions between and . Thus, the algorithm is correct.

Now that we know merge-and-count is correct, we can then use the book’s second algorithm to sort a given list and count its inversions. We don’t need to make any changes, so I’ll just paste it below:



Because each iteration of the while loop on line 4 of Merge-and-Count runs in constant time, and because in each iteration we either add one element from or one element from (and never look at that element again), the while loop will require no more than iterations.

Because each iteration of the while loop on line 8 of Merge-and-Count runs in constant time, and because in each iteration we either increment or , this too runs in . Thus, the worst-case running time of Merge-and-Count is .

Because Sort-and-Count recursively sorts both halves of the elements in a given list (and then calls Merge-and-Count), and because Merge-and-Count runs in , we know that Sort-and-Count runs in .

In the best case, we still have a runtime of . Sort-and-Count can’t know whether the list it receives is pre-sorted, so it must always recursively break apart the list. Merge-and-Count might receive a list in which all elements are less than the first element in , but it’d still have to iterate over elements at the top level. Thus, our runtime doesn’t change.

Because the runtime is bounded above and below by , this indicates that all cases (best, worst, and average) should run in . Of course, input models can certainly affect the overall runtime, but any changes aren’t going to be drastic.

**Chapter 5, Problem 6**

****

**Solution**

*Overall, this problem took me 30 minutes. I’d rate it a 3 on the difficulty scale; the algorithm was easily found and neither the proof nor the analyses proved particularly challenging. I worked with Savannah Hyde.*

Let’s assume we are given the root node . If we start with , we can probe its two children to determine whether or not is a local minimum. If it is, we can simply return . If it is not, we can recursively call our algorithm on the smaller of ’s children.

Here’s the formal algorithm:

1 FindLocalMinimum(Node )

2 If does not have a left child,

3 If does not have a right child,

4 If and or if is a leaf

5 Return as a local minimum

6 Else if

7 Return FindLocalMinimum(left)

8 Else

9 Return FindLocalMinimum(right)

10 Endif

This algorithm, of course, is pretty straightforward. Let’s prove its correctness. We’ll do so with a proof by contradiction.

Let’s assume the algorithm cannot find a local minimum. This can occur in one of two ways:

1. A local minimum does not exist in the tree
2. The algorithm indicates a node is a local minimum, but node is actually not a local minimum (or vice versa). Necessarily, either
   1. The parent of has a lower value than that of , or
   2. One (or both) of the child nodes of has a lower value than that of .

Reaching a contradiction in Case 1 is trivial. Because we know that all nodes have unique, real values, there is necessarily an absolute minimum. This absolute minimum, of course, is also a local minimum in some area of the tree. Thus, Case 1 leads to a contradiction.

Case 2 effectively says the algorithm will mis-identify a node as a local minimum. This means that one of ’s neighbors – in other words, either the parent node (case 2a) or at least one of the two child nodes (case 2b) of – must have a lower value than that of .

Case 2a implies that the parent of recursively called the algorithm on . This means that the value of is less than the values of and ’s sibling. This is a contradiction, though, because we’ve assumed that .

Case 2b implies that the algorithm should have recursively called itself on one of ’s child nodes (whichever has a lower value). However, the algorithm returned as a local minimum, and we thus have a contradiction.

Because all cases lead to a contradiction, we’ve shown that our algorithm will always find a local minimum. This proves its correctness.

Let’s now conduct an asymptotic analysis of the algorithm.

In the best case, the given root node is itself a local minimum. In this case, we must probe the root and each of its children. Thus, we will always probe three nodes when the root is a local minimum. Because we perform exclusively constant-time work (assignments and comparisons) before returning, our best case runs in , or constant, time.

In the worst case, for all parent nodes , and for all child nodes , . In other words, nodal values strictly decrease as we move from parent to child. This indicates that we won’t reach a local minimum until we reach a leaf node. In this case, at every recursive call, the algorithm will visit either of the two child nodes; this effectively throws away half of all remaining nodes in the search space. Because we throw away half of the remaining search space at each recursive call, we know we visit nodes and, at each visited node, we perform constant-time work (assignments and comparisons); thus, our worst case runs in time.

Because we really can’t assume anything about the structure of our tree – does it strictly decrease? Are the values of nodes randomized? Is it sorted in some way? – describing an average-case runtime is difficult. If there exists just one local minimum, it’s likely near the bottom of the tree, and so the runtime is near . Because there’s often more than one local minimum, we can’t define much more than the previously-mentioned upper- and lower-bounded runtimes.

**Chapter 5, Problem 7**

****

****

**Solution**

*Overall, this problem took me 30 minutes. I’d rate it a 5 on the difficulty scale; a correct algorithm was readily apparent, but it took some effort to devise one that performs probes. The proof, too, was nontrivial.*

To solve this problem, we need to utilize a technique known as *rolling downhill*. Effectively, when looking at an element in the graph, we can

1. Decide that is a local minimum and return, or
2. Roll downhill to an element smaller than .

In the case of (b), instead of always rolling to the smallest adjacent element, though, we’ll roll into the side of the graph that *contains* that element smaller than ; from there, we can divide into pieces and try again. Let’s formalize the algorithm:

1 FindLocalMinimum()

2 Let be the index of the middle column; break ties arbitrarily

3 Let be the subgraph consisting of all columns

left of column

4 Let be the subgraph consisting of all columns

right of column

5 Let be the smallest element in column

6 If has a smaller value than that of each neighbor

7 Return as a local minimum

8 Else

9 If

10 Return FindLocalMinimum()

11 Else

12 Return FindLocalMinimum()

13 Endif

14 Endif

Because the labels in the graph are all distinct, we must have a global minimum. Because a global minimum exists, at least one local minimum always exists.

This algorithm will only ever roll *downhill*; that is, this algorithm will never enter a half of in which it could roll back out. This indicates that the number of columns in our search space decreases by a factor of (roughly) two at every recursive call. Because the search space decreases at every step, we must – in the worst case – necessarily reach a point where we have one final column to examine.

We know that we rolled into this column because it contains at least one element with a value less than any of the values in the two neighboring columns. We also know that, if we consider solely this column, it too must possess a global minimum. Because the algorithm selects the minimum element when examining a column, we know the algorithm will select *this column’s* global minimum, and it will thus select a local minimum in .

Of course, the algorithm may not need to search until it reaches a search space of a single column; it is certainly possible – likely, too, depending on the size of – that the algorithm will identify a local minimum *before* reaching this last column. I only mean to show that, in the absence of any other local minima, the algorithm will at least find one in its last column. This concludes the proof of correctness.

In this algorithm, we only probe elements of in lines 5, 6, and 9. In line 5, we perform probes because we must search every element in column for the minimum value. In line 6, we perform probes because can have no more than four neighbors. In line 9, we perform probes because can have no more than one left neighbor and one right neighbor. Because we perform – in total – probes, we perform probes, where is some constant. We can simplify this to an upper bound of probes.

In the best case, the minimum element in the middle column of is itself a local minimum. Here, we perform probes to determine that is a local minimum. This, of course, also simplifies to probes.

However (depending on the size of , of course), a local minimum likely won’t exist in the middle column. In this average case, we must again perform probes at each step until we reach a local minimum. We can see now that our algorithm will always require probes to identify a local minimum.