Midterm 1 Take-home

Joe Puccio

October 2, 2014

Collaborators: Sana Imam, Ryan Allan, Rahul Ramkumar, Max Daum, and Matthew Leming, Nathan Weatherly.

1.

a) Assuming that we can't compare bolts with bolts and nuts with nuts, and assuming that it is acceptable to consider the average run time rather than the worst case of randomized Quicksort, we accept for the purposes of this proof that randomized Quicksort performs in $o(k^2)$. Then we may implement Quicksort by choosing a random pivot bolt, and then partitioning the nuts with this bolt into two regions, one with elements less than or equal to the pivot and one with elements just greater than the pivot. This will require k tests to complete. Because we are guaranteed that every bolt has a match and each nut has a distinct diameter, we know that we'll find this bolt's corresponding nut during the operation and it'll be the last element in the less or equal to region. Now, we'll use this corresponding nut to partition the bolts in the same manner that is described above, but because we already know which bolt this nut corresponds to, we will only have to make $k-1$ tests. We know that all of the nuts in the region larger than the original bolt-pivot each have a corresponding bolt in the region larger than the original nut-pivot, and the same is true for the smaller region as well. This means that we can repeat this operation on each of these regions just as we did originally as all of the conditions are the same, simply with a different number of terms. Thus, we can write the tests performed in our algorithm the following recurrence:

$$T(1) = 1$$

$$T(k) = k + (k-1) + 2T(\frac{k}{2})$$

Now, this unpacks to be $ck\log(k)$, which we know falls below dk^2 regardless of the choice of d, $\forall k > k_0$ for some k_0. All that is left is to read off the diagonal which can be done in k time, which also is certainly $o(k^2)$, and thus we have shown that our entire algorithm can be done in $o(k^2)$ time.

b) Now considering the situation where we are not guaranteed that matching pairs exist, let's consider the case where, for every nut there exist *no* bolts that match it. This is possible scenario that may arise and we will show that no algorithm is capable of completing in better than k^2 tests. For an algorithm to be valid, it must halt (complete) on an input and it must produce the valid output. In our scenario, a valid output would be the *comprehensive* set of bolt-nut matches, which means that a valid algorithm must test each bolt per nut (or vice versa) if no such match exists, in order to be valid. But now considering our earlier case where an input is given where *no* matches exist, then we can easily conclude that any valid algorithm must have tested each $(n, b) \in N \times B$, which means it must have performed k^2 tests, and therefore every algorithm must be $\Omega(k^2)$.

2.

a) i) Handling one insertion: we will update the weight values while traversing the tree to insert the new node, and then after the insertion is completed we will splay on the inserted node. We assume that the weights will be properly updated in the rotations involved with splay, and therefore our weight values will be correct once the overall insert and splay operations are done. Our method of updating the weights is to traverse the tree to find the location to insert the new node. During this traversal, add 1 to each of the nodes you examine as they will become an ancestor of this new node once it's inserted (top-down). Alternatively, assuming nodes have parent pointers, once the node has been inserted you may traverse back up to the root from the inserted node's location adding 1 to each of these parents (bottom-up). No other weights will be affected and the worst case run time is $O(h)$, where h denotes the height of the tree.

ii) Handling one deletion: again, splay trees will call splay after a node's deletion so we assume that the subsequent rotations following this deletion will correctly update the relevant weights. Our method of removal is to traverse down to the node to be deleted, subtracting 1 from each examined node as each of them will be losing a descendant once the node is deleted (this is a top-down approach). Alternatively, if the nodes hold parent pointers, we may do this process bottom-up. No other weights will be affected and the worst case run time is again $O(h)$, where h denotes the height of the tree.

iii) Handling one rotation: the rotation, and corresponding weight changes, will depend on the structure of the ancestors. Note: for simplicity in all of the following cases, let the weight of $node.left/right$ actually be the number of nodes including $node.left/right$; this will make summaries simpler. We consider the left "zig" case where a node x is to be swapped with its parent node p, and x is p's left child (without loss of generality, this same operation will be applicable, mirrored, to the right child case). The weight of x would change to $p.weight$. Conversely, p's weight would change to $p.weight - x.left.weight - 1$. These are the only weights that need to be updated as a result of this change. Now consider the left "zig-zig" case, where x is to be swapped with its grandparent node g, and has a parent node p. The weight of x would become $g.weight$, the weight of p would become $p.weight - x.left.weight + g.right.weight$, and the weight of g would become $p.right.weight + g.right.weight$. Now, by symmetry the same for the right "zig-zig" case is trivial. Lastly, we must consider the left "zig-zag" case, again with nodes x, p, and g with identical relationships. The weight of x becomes $g.weight$, p would become $p.weight - (x.left.weight + g.right.weight)$, and g would become $p.right.weight + g.right.weight + 1$. Again, the mirror scenario can be solved trivially. And we have thus showed how to update weights during all rotations. Because these operations are local to an area of the tree (they do not need to examine all nodes in the tree), they require only constant time $O(1)$.

b) It can be calculated top-down while searching for a node. Initialize rank = 1, and call $currentNode$ the node we're currently comparing our key against. As we traverse down the tree in search of the matching key, if we navigate to the left subtree of $currentNode$ we leave the value of rank unchanged, and if we navigate right subtree we add $(1 + currentNode.left.weight)$ to the value of rank because we know that all of these nodes and $currentNode$ (hence +1) will come before our node in an in-order traversal. Lastly, when we finally find the node, we return the current value of rank + $currentNode.left.weight$, which is important when the node being searched for is not a leaf of the BST. The worst case run time is $O(h)$, where h denotes the height of the tree.

c) Initialize *inversions* to 0, then say we set *newRank* = $T.insert(key)$ and after this insertion we incremented a variable *count* which holds the current number of nodes in the tree. Then, assuming the following code had executed on all previous insertions, the variable *inversions* would store the number of inversions up until and including the last insertion:

```
1: if $newRank < count$ then
2:     $inversions$ += $(count - newRank)$
3: end if
```

.

d) First, let's create points (or "flags") for each of the endpoints of the segments where each flag contains its x and y position as well as whether it's the start flag of a segment (lower y coord) or the terminal flag (larger y coord), and put them all into an array. Then we can sort this array based on y coordinate, which can be done using Mergesort, for example, in $O(n \log n)$. Now using the BST data structure from part c (using y-value as the order property), we may navigate through our sorted array and insert the start flags and remove the corresponding start flag when we reach a terminal flag. On these removals, we can run a find on the flag before removing it to check its rank and set *rankToRemove* equal to it. Then, for each removal with *intersections* initialized to 0 we execute the following code:

```
if $rankToRemove > 1$ then
    $intersections$ += $(rankToRemove - 1)$
end if
```

. At the end of all of the insertions and deletions, *intersections* will hold the number of intersections of all the segments.