Algorithms Assignment 6

Andrew Osborne

March 18, 2019

14.1-3

```
OS-Select(x,i) // finds ith OS in tree rooted by x
  y = x // dummy variables
  j = i
  while j != y.left.size+1:

    if j < y.left.size+1
      y = y.left

    elseif j > y.left.size+1:
      j = j - y.left.size - 1
      y = y.right

  return y
```

14.1-5

This routine will find the ith successor of a node x in an n element order–statistic tree, T.

```
Ith-Successor(T,x,i)
  j = OS-RANK(T,x) // O(lg n) runtime
  return OS-Select(T.root,j+i) // O(lg n) runtime
```

It is clear that the routine posed above runs in $c_1 lg\,(n) + c_2\,lg(n) = (c_1 + c_2)\,lg\;n = O(lg\;n)$ time in the worst case, so this is satisfactory for our purposes.

14.1-6

We must write some subroutines to modify existing logic for the purpose of this problem.

```
// to be called AFTER a right rotation
Right-Rotate-Fix(x)
  x.rank = x.rank - x.p.rank

// to be called AFTER a left rotation
Left-Rotate-Fix(x)
  x.p.rank = x.rank + x.p.rank
```

These two routines will be used to be used to preserve rank–information under rotations. That is, when preforming a left–rotation on a node, say x, then x becomes the left child of x.right, whose left subtree is transplanted onto the right subtree of x. In this case, the rank of x will not be altered under a left rotation, but (x.p after rotation or x.right before rotation) needs to have its rank increased by the rank of x. On the other hand, under right–rotation, the rank of x will change, because of the replacement of the left subtree of x with the right subtree of x.left (before rotation).

That may be unclear, but we may simply append `Right-Rotate-Fix(x)` to the last line of `RIGHT-ROTATE`, and we may append `Left-Rotate-Fix(x)` to the last line of `LEFT-ROTATE` in order to prevent rotations from damaging our rank information.

On the other hand, at some point during the insertion process, any node to be inserted will be a leaf node in some tree. At this point, after the initial binary search tree insertion, before calling any FIXUP functions, we may call

```
Increment-Insert(x)
  if x.p != NIL
    if x == x.p.left
      x.p.rank = x.p.rank + 1
    Increment-Insert(x.p)
```

on the inserted node in order to maintain rank information during insertion, while the rotation operations above maintain rank information during the fixup process which comes after insertion.

During deletion, we need only reduce the rank of elements between the element to be deleted and the root of our tree.

```
Decrement-Delete(x)
  if x.p != NIL
    if x == x.p.left
      x.p.rank = x.p.rank 1 1
    Increment-Insert(x.p)
```

That is, before deleting the specified element from the tree, call `Decrement-Delete(x)` on the element to be deleted.

14.1-7

Consider an array of real valued constants, $A = [a_1, a_2, a_3, \ldots, a_n]$. Then for $1 \leq k \leq n$, let r_k be the rank of a_k in A, and let $R = [r_1, r_2, r_3, \ldots, r_n]$. Using one's intuition it is clear that r_1 is the rank of a_1 and also the number of elements beyond a_1 which are less than a_1 minus one! That is, there are $r_1 - 1$ values in $A[2..n]$ which are less than or equal to a_1, and if there are m values in A equal to a_1, then there are $r_1 - m$ elements in $A[2..n]$ which are less than a_1. Then we may repeat this process on $A[2..n]$ to continue finding the number of inversions of A. The above mentioned subtraction by m has the effect of ensuring that we always treat each nonunique item as lowest rank value which any element of that value can have. In our discussion of this chapter, we have split nonunique items to the right. Essentially, if we search for some element in an order–statistic tree, the first node with the value of our element will be the one with the lowest rank and all subsequent values will lie in the right subtree of the one with the lowest rank. Then, with all of this in mind, the solution to this problem is clear

```
COUNT-INVERSIONS(Array)
  let T be a new order-statistic tree
  n = Array.length

  for i = 1 to n      // n
    OS-insert(T,A[i]) // O(lg n)

  inversions = 0
  for j = 1 to n      // n
    x = RB-SEARCH(T.root,A[j]) // O(lg n)
    inversions = inversions + OS-RANK(T,x) - 1 // O(lg n)
    OS-delete(T,x) // O(lg n)
  return inversions
// O(n lg n)
```

15.1-1

Let

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \ \ \& \ \ T(0) = 1$$

Then consider

$$T(n) - T(n-1) = \sum_{j=0}^{n-1} T(j) - \sum_{j=0}^{n-2} T(n) = T(n-1)$$

So then $T(n) - T(n-1) = T(n-1)$ which implies that $T(n) = 2T(n-1)$. Then, quite clearly, $T(n) = 2^m T(n-m)$ which implies that $T(n) = 2^n T(n-n) = 2^n T(0) = 2^n$

15.1-2

Let $p = [1, 10, 1, 1, 24]$ and let $n = 5$. Our "density" array, $d = [1, 5, 0.3333, 0.25, 4.8]$ has a maximum which occurs at 2. Then the most dense length that we can create is a rod of length 2. Our greedy process will produce two rods of length 2 and a rod of length 1 for a total revenue of 21. However, if we had not cut the rod at all, and left it as a rod of length 5, we could have sold it for a revenue of 25 which is higher than 21, so our greedy strategy is not optimal.

15.1-3

I shall deliver this algorithm in the form of pseudocode.

```
MEMOIZED-CUT-ROD-MODIFIED(p,n,c)
  let r[0..n] be a new array
  for i = 0 to n
    r[i] = -infinity
  return MEMOIZED-CUT-ROD-AUX-MODIFIED(p,n,c,r)

MEMOZED-CUT-ROD-AUX-MODIFIED(p,n,c,r)
  // don't solve same problem twice
  if r[n] >= 0
    return r[n]

  // don't make money off of nothing
  if n == 0
    q = 0
  else
    q = -infinity
    for i = 1 to n-1
      // notice that we must modify the upper limit on our for loop
      // to avoid deducting c in the situation where we make no cuts
      q = max(q,p[i] + MEMOIZED-CUT-ROD-AUX-MODIFIED(p,n-i,c,r) - c)

  // nth step of for loop done outside the loop with no cuts
  r[n] = max(q,p[n])
  return r[n]
```

Note that we still retain the property that we are never indebted by selling rods. The most important change in this algorithm is that the final step of the for loop cannot be executed with the other cases because, if it were, we would be penalized by c without making any cuts. Moreover, the subtraction of x within the max inside the for loop reflects the penalty for making cuts.