Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-2.22

Let d_{uv} be the diameter of T. First observe that both u and v are tree leaves; if not, we can find a path of higher length, only by considering one of u's or v's children. Moreover, one of u, v has to be a leaf of highest depth. This can be proved by contradiction. Consider a tree and some leaf-to-leaf path P that does not include a leaf of highest depth. Then consider a leaf of greatest depth v; it is always possible to find a new path P' that starts at v and is longer than P. This yields the desired contradiction.

The main idea of the algorithm is to find a leaf v of highest depth and starting from this leaf to keep moving towards the tree's root. At each visited node u (including v, but excluding the tree's root), the height of u's sibling is computed and the current length L_{max} of the longest path in which v belongs is updated accordingly. When we are at T's root, L_{max} contains the diameter of T.

We give the pseudo-code for the above algorithm. We use methods depth(T,v) and height(T,v), which are described in section 2.3.2, pages 80-81, of the textbook, and compute the depth and correspondingly the height of the subtree rooted at v. Method DeepestLeaf(T, v, H, c) returns the deepest node of T or NULL if it is not found in the subtree rooted at v. This is done by essentially performing a pre-order traversal that stops as soon as the deepest leaf is found. H is T's height and c is a counter.

```
Algorithm DeepestLeaf(T,v,H,d):
  Input : a tree T, a node v, the height H, and the current depth d
  Output : The deepest node, or NULL if it is not found

  if T.isExternal(v) then
    if d = H then { Base case: check for deepest node }
      return v
    return NULL
  u ← NULL
  {See if there is a deepest leaf in the left child}
  u ← DeepestLeaf(T,T.leftChild(v),H,d+1)
```

```
  if u ≠ NULL then
    return u
  {See if there is a deepest leaf in the right child}
  u ← DeepestLeaf(T,T.rightChild(v),H,d+1)
  return u


Algorithm findDiameter(T):
  Input : a tree T
  Output : The diameter of T

  H ← depth(T,T.root())
  v ← DeepestLeaf(T,v,H,0)
  if v = T.root() then
    return 0 { if deeps leaf is just the root node, diameter is 0}
  L_max ← 0
  x ← 1   {x: number of nodes visited}
  while v ≠ T.root() do
    {Assign y to be the length of the path going to the sibling}
    y ← height(T,T.sibling(v))
    {See if we have found a new longest path}
    if x+1+y > L_max then
      L_max ← x+1+y
    {Climb up the tree one step}
    x ← x+1
    v ← T.parent(v)
  return L_max
```

The running time of the algorithm is linear. Both depth(T,v) and height(T,v) have complexity $O(S_v)$, where S_v is the size of the subtree rooted at v. depth(T,v) is called from the tree's root once, so its time complexity is $O(n)$. height(T,v) is called for each sibling on our way up to the root, so, in total, it adds an $O(n)$ time complexity. DeepestLeaf also has linear worst case time complexity, for it is essentially a pre-order tree traversal. Finally, the leaf-to-root path traversal adds a linear time complexity.

We can improve on this algorithm, however. Note that the above algorithm can visit all the nodes in the tree twice, once to find the deepest node, and once to

find the height subtrees. We can combine these two operations into one algorithm with a little bit of ingenuity and recursion. We observe that given a binary tree T_v rooted at a node v, if we know the diameters and heights of the two subtrees, we know the diameter of T_v. Imagine we took a path from the deepest node of the left subtree to the deepest node of the other subtree, passing through v. This path would have length $= 2 + \text{height}(T_v, T_v.\text{leftChild}(v)) + \text{height}(T_v, T_v.\text{rightChild}(v))$. Further, note that this is a longest path that runs through the root of T_v, since the height of the left and right subtrees is simply the longest path from their respective roots to any of their leafs. Therefore, the longest path in T_v is simply the maximum of longest path in the found in T_v's left and right subtrees, and the longest path running through v. These observations give rise to the following algorithm. For convenience, we denote the return type of this algorithm to be in the form (h,d) where h and d represent height and diameter respectively, and are accessed using the h() and d() methods.

Algorithm Diameter(T,v)
  Input : a tree T and a node in that tree v
  Output : a (h,d) pair

  if T.isExternal(v) **then** {Base case}
    return $(0,0)$

  {Get the return pairs of the left and right subtrees}
  $L \leftarrow$ Diameter(T, T.leftChild(v))
  $R \leftarrow$ Diameter(T, T.rightChild(v))

  {Find the height of this subtree}
  $H \leftarrow \max(L.h(), R.h()) + 1$

  {Find the longest path length}
  $P \leftarrow L.h() + R.h() + 2$
  return $(H, max(L.d(), R.d(), P))$

Since this algorithm visits each node in the tree exactly once, and performs a constant amount of operations at each node, it follows that this algorithm runs in $O(n)$