d) What would be the overall, worst-case $O(\cdot)$ for Quick Sort? Repeatedly pick pivot that's largest

\[ \begin{align*}
&\text{Print} \quad n-1 \text{ compares} \\
&\text{Print} \quad n-2 \text{ compares} \\
&\text{Print} \quad n-3 \text{ compares} \\
&\text{Print} \quad \frac{n(n-1)}{2} = O(n^2)
\end{align*} \]

\[ \begin{align*}
&\text{Print} \quad \frac{n(n-1)}{2}
\end{align*} \]

\[ \begin{align*}
\text{Call}
\end{align*} \]

e) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

\[ \begin{align*}
\log n \text{ levels}
\end{align*} \]

\[ \begin{align*}
&\text{Call} \\
&\text{Call} \\
&\text{Call}
\end{align*} \]

\[ \begin{align*}
&n \text{ compares} \\
&n \text{ compares} \\
&n \text{ compares}
\end{align*} \]

\[ \begin{align*}
&O(n \log_2 n)
\end{align*} \]

f) What would be the big-oh for Quick Sort in the average case? Picking a pivot at random should roughly partition list into equal pieces, so $O(n \log_2 n)$ on average.

g) The textbook's partition code (Listing 5.15 on page 225) selects the first item in the list as the pivot item. However, the above partition code selects the middle item of the list to be the pivot. What advantage does selecting the middle item as the pivot have over selecting the first item as the pivot?

If list already sorted, picking first item as pivot leads to worst case $O(n^2)$ compares since pivot always on left end.
1. Consider the parse tree for $(9 + (5 * 3)) / (8 + 4)$:

![Parse Tree Diagram]

a) Identify the following items in the above tree:
- node containing "*"
- edge from node containing "+" to node containing "8"
- root node
- children of the node containing "+"
- parent of the node containing "3"
- siblings of the node containing "*
- leaf nodes of the tree
- subtree who's root is node contains "+
- path from node containing "+" to node containing "5"
- branch from root node to "3"

b) Mark the levels of the tree (level is the number of edges on the path from the root)

c) What is the height (max. level) of the tree?

2. In Python an easy way to implement a tree is as a list of lists where a tree look like:

```
[ "node value", remaining items are subtrees for the node each implemented as a list of lists]
```

Complete the list-of-lists representation look like for the above parse tree:

```
["9", ["+", ["5", ["*", ["8", []], ["4", []]]], ["3", []]]], ["/", ["+", ["5", ["*", ["8", []], ["4", []]]], ["3", []]]], ["8", []]
```

3. Consider a "linked" representation of a BinaryTree. For the expression $((4 + 5) * 7)$, the binary tree would be:

```python
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

![Linked Binary Tree Diagram]
import operator

class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.left = self.leftChild
            self.leftChild = t

    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.right = self.rightChild
            self.rightChild = t

    def isLeaf(self):
        return (not self.leftChild) and (not self.rightChild)

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self):
        return self.key

    def inorder(self):
        if self.leftChild:
            self.leftChild.inorder()
        print(self.key)
        if self.rightChild:
            self.rightChild.inorder()

    def postorder(self):
        if self.leftChild:
            self.leftChild.postorder()
        if self.rightChild:
            self.rightChild.postorder()
        print(self.key)

    def preorder(self):
        print(self.key)
        if self.leftChild:
            self.leftChild.preorder()
        if self.rightChild:
            self.rightChild.preorder()

    def printexp(self):
        if self.leftChild:
            print('(', end=' ')
            self.leftChild.printexp()
            print(')', end=' ')
        if self.rightChild:
            print(')', end=' ')
        if self.leftChild:
            self.leftChild.printexp()
        if self.rightChild:
            self.rightChild.printexp()

    def postordereval(self):
        opers = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
        res1 = None
        res2 = None
        if self.leftChild:
            res1 = self.leftChild.postordereval()
        if self.rightChild:
            res2 = self.rightChild.postordereval()
        if res1 and res2:
            return opers[self.key](res1, res2)
        else:
            return self.key

def preorder(tree):
    sVal = ''
    if tree:
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')
        return sVal

def postordereval(tree):
    opers = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
    if res1 and res2:
        return opers[tree.getRootVal()](res1, res2)
    else:
        return tree.getRootVal()
b) If myTree is the BinaryTree object for the expression: \((4 + 5) \times 7\), what gets printed by a calls to:

<table>
<thead>
<tr>
<th>myTree.inorder()</th>
<th>myTree.preorder()</th>
<th>myTree.postorder()</th>
<th>inorder(myTree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4)</td>
<td>(5)</td>
<td>(7)</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>(5)</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>


c) If myTree is the BinaryTree object for the expression: \((4 + 5) \times 7\), what gets printed by a call to myTree.printexp()?

d) If myTree is the BinaryTree object for the expression: \((4 + 5) \times 7\), what gets returned by a call to myTree.postordereval()?

e) Write the height method for the BinaryTree class.

\[
\text{height} = \max(\text{height left subtree}, \text{height right subtree})
\]

4. Consider the Binary Search Tree (BST). For each node, all values in the left-subtree are < the node and all values in the right-subtree are > the node.

```
      50
     /   \
   30     70
  /   \   /   \
 9     34 58   80
 / \   /   \
18 32 47
```

a. What is the order of node processing in a preorder traversal of the above BST?

b. What is the order of node processing in a postorder traversal of the above BST?

c. What is the order of node processing in an inorder traversal of the above BST?

d. Starting at the root, how would you find the node containing “32”? 

e. Starting at the root, when would you discover that “65” is not in the BST?

f. Starting at the root, where would be the “easiest” place to add “65”?

g. Where would we add “5” and “33”?
class BinaryTree

def height(self):
    if self == None:
        return -1
    return 1 + max(self.leftChild.height(), self.rightChild.height())

self
height self
0 = 1 + max(-1, 1)