Question 1. [5 marks]

Read over the declaration of class BTNode as well as the header and docstring for function has_branch. Then complete the implementation of has_branch.

```python
class BTNode:
    '''A node in a binary tree.'''

    def __init__(self: 'BTNode', item: object, 
                 left: 'BTNode' = None, right: 'BTNode' = None) -> None:
        '''Initialize this node.'''
        self.item, self.left, self.right = item, left, right

    def has_branch(T: BTNode, item1: object, item2: object) -> bool:
        '''Return True if tree rooted at T has some node with node.item == item1 
           that has a child with child.item == item2. Return False otherwise.

           >>> T = BTNode(1, BTNode(2, BTNode(3)), BTNode(4, BTNode(5), 
                       BTNode(6)))
           >>> has_branch(T, 2, 3)
           True
           >>> has_branch(T, 4, 7)
           False
           '''

        if T is None:
            return False
        else:
            return ((T.item == item1 and
                     ((T.left and T.left.item == item2) or
                      (T.right and T.right.item == item2))) or
                     has_branch(T.left, item1, item2) or
                     has_branch(T.right, item1, item2))
```

Marking notes: 1 mark for None base case, 2 marks for checking whether T has a branch with appropriate values, 2 marks for checking T.left and T.right for branches.

A, A1: check children before recursion -0.5
B, B1: check existence of children (lines 5, 6) -1
D: check further than current parent/child if they don't match -1
E: omitting 'or' and ris returning False prematurely -0.5
F: not returning -0.5
G: missing args in recursive calls -0.5
H: compares node with item -0.5
I: incorrect connectives -0.5, missing connectives -1
J: used self instead of T -0.5

Question 2. [5 MARKS]

Read over the declarations of classes BTNode and LLNode, as well as the header and docstring for function root_to_leaves. Then implement the function root_to_leaves.

class BTNode:
    """A node in a binary tree.""
    def __init__(self: 'BTNode', item: object,
                 left: 'BTNode' =None, right: 'BTNode' =None) -> None:
        """Initialize this node.
        ""
        self.item, self.left, self.right = item, left, right

class LLNode:
    """A node in a linked list.""
    def __init__(self: 'LLNode', item: object, link: 'LLNode' =None) -> None:
        """Initialize this node.
        ""
        self.item, self.link = item, link
    def __repr__(self: 'LLNode') -> str:
        """Return a string that represents self in constructor (initializer) form.
        ""
        return ('LLNode({}, {})'.format(repr(self.item), repr(self.link))
                if self.link else 'LLNode({})'.format(repr(self.item)))
    def __eq__(self: 'LLNode', other: 'LLNode') -> bool:
        """Return whether LLNode self is equivalent to LLNode other"
        return (isinstance(other, LLNode) and
                self.item == other.item and self.link == other.link)
def root_to_leaves(T: BTNode) -> list:
    """
    Return list of paths from T to each of its leaves, or []
    if T is None. Each path is a linked list formed from LLNodes.
    You should return a list containing a single-node linked list
    when T has no children.
    >>> T = BTNode(1, BTNode(2, None, BTNode(3)), BTNode(4, BTNode(5), BTNode(6)))
    >>> L1 = root_to_leaves(T)
    >>> L2 = [LLNode(1, LLNode(2, LLNode(3))), LLNode(1, LLNode(4, LLNode(5))), 
           LLNode(1, LLNode(4, LLNode(6)))]
    >>> len(L1) == len(L2) and all([p in L2 for p in L1])
    True
    """
    if T is None:
        return []
    elif T.left is None and T.right is None:
        return [(LLNode(T.item,None))]
    else:
        leftchpaths = root_to_leaves(T.left)
        rightchpaths = root_to_leaves(T.right)
        leftpaths = [LLNode(T.item, P) for P in leftchpaths]
        rightpaths = [LLNode(T.item, P) for P in rightchpaths]
        return leftpaths + rightpaths

Marking notes: 1 mark for None base case, 1 mark for leaf base case, 3 marks for computing lists of child
paths and combining them into list of paths from this node.

Question 3. [5 MARKS]

Read over the class declaration for BTNode and the docstring for function ordered_and_bounded. Then
implement ordered_and_bounded.

class BTNode:
    """A node in a binary tree."""

def __init__(self: 'BTNode', item: object,
             left: 'BTNode' =None, right: 'BTNode' =None) -> None:
    """Initialize this node.
    """
    self.item, self.left, self.right = item, left, right

def ordered_and_bounded(T: BTNode, lower: int, upper: int) -> list:
"""Return a list of items, in ascending order, from nodes of T, with all items no less than lower and no greater than upper. Return [] if T is None. You are *not* allowed to sort any list, and you should visit as few nodes as possible.

preconditions:  -- node items in T are comparable,  -- T is a binary search tree in ascending order,    that is, all items in every left sub-tree are less than the sub-tree's root and all items in every right sub-tree are more than the sub-tree's root

if T is None:
   return []
else:
   return ((ordered_and_bounded(T.left, lower, upper)
     if lower < T.item else []) +
     ([T.item] if lower <= T.item <= upper else []) +
     (ordered_and_bounded(T.right, lower, upper)
     if upper > T.item else []))

Marking notes: 1 mark for None base case. 1 mark for getting list from left subtree if lower <= T.item. 1 mark for getting list from right subtree if upper >= T.item. 2 marks for adding T.item to list if it is in interval [lower, upper]. 1 mark off if extra nodes are visited, that is BST property not used. 1 mark off if list is sorted.

Question 4.  [6 MARKS]

Read the functions hybrid_search and hybrid_search2. For each function, decide which of the following complexity classes best describe that function's worst-case performance on a list of n elements:

\[
\mathcal{O}(1) \quad \mathcal{O}(\log n) \quad \mathcal{O}(n) \quad \mathcal{O}(n \log n) \quad \mathcal{O}(n^2)
\]

For each function, explain why your choice of big-Oh complexity makes sense. Also explain what behaviour you expect hybrid_search and hybrid_search2 should exhibit when run on a computer on a list of size 2n versus a list of size n.

def hybrid_search(x:int,L:list) -> bool:
   """precondition: L is sorted
   >>> L = [1,5,9, 9, 9, 12, 12, 15, 19,20,40,41,42,43,50,100,500]
   >>> hybrid_search(21,L)
>>> hybrid_search(100,L)
True

"""
def helper(i,j) -> bool:
    # precondition: 0 <= i <= j < len(L)
    if (j-i) < len(L)/10:
        return any([y == x for y in L[i:j+1]])
    if x < L[(i+j)//2]:
        return helper(i, (i+j)//2-1)
    elif x > L[(i+j)//2]:
        return helper((i+j)//2+1, j)
    else:
        return True
return helper(0,len(L)-1)
"""

O(n). The call to the helper methods occur at most 4 times before \( j - i < \frac{\text{len}(L)}{10} \), and then we must search a slice of size between \( n/10 \) and \( n/20 \). Then each element in a slice with at least \( n/20 \) elements must be inspected. I expect the running time to roughly double if I increase the size of the list from \( n \) to \( 2n \).

Marking notes: 2 marks for choosing \( O(n) \) with a suitable explanation. 1 mark if their expectation of how running time scales with doubling the list size is consistent with (whatever) choice of complexity class they make.

---

```python
def hybrid_search2(x:int,L:list) -> bool:
    """precondition: L is sorted
    >>> L = [1,5,9, 9, 9, 12, 12, 15, 19,20,40,41,42,43,50,100,500]
    >>> hybrid_search(21,L)
    False
    >>> hybrid_search(100,L)
    True
    """
def helper(i,j) -> bool:
    # precondition: 0 <= i <= j < len(L)
    if (j-i) < 10:
        return any([y == x for y in L[i:j+1]])
    if x < L[(i+j)//2]:
        return helper(i, (i+j)//2-1)
    elif x > L[(i+j)//2]:
        return helper((i+j)//2+1, j)
    else:
        return True
    return helper(0,len(L)-1)
```

---

O(n). The call to the helper methods occur at most 4 times before \( j - i < \frac{\text{len}(L)}{10} \), and then we must search a slice of size between \( n/10 \) and \( n/20 \). Then each element in a slice with at least \( n/20 \) elements must be inspected. I expect the running time to roughly double if I increase the size of the list from \( n \) to \( 2n \).

Marking notes: 2 marks for choosing \( O(n) \) with a suitable explanation. 1 mark if their expectation of how running time scales with doubling the list size is consistent with (whatever) choice of complexity class they make.
$O(lg\ n)$. The helper method is called approximately $lg\ n - 3$ times, and then a linear search of no more than 10 items is performed, so the complexity is proportional to $lg\ n$. I expect that the running time would increase by a constant as the size of the input list was doubled.

Marking notes: 2 marks for indicating $O(lg\ n)$ and giving a suitable explanation. 1 mark for indicating that run time would increase by a constant if the length of the input list were doubled.