[a+1 for a in L]
[a for a in L if a>0]

Section 6 Recursion

[a if isinstance(a, int) else Sum(a) for a in L]

Recursion 三部曲：

1. Function description：根据docstring 搞清function的具体作用, input types和 rtype. 在写body的时候 assume function已经写好啦
2. Base Case： 一般来说是各种数据类型的最小值， 比如integer的一般为0或1. String的一般是empty string, nested list的一般是非list或者empty list. 树可以是leaf or None.
3. Recursive step:
   3.1 Divide step： 搞清楚是如何把大问题化小, 是如何把当前问题分解成一个相对较小的问题.
   3.2 Combination step： 搞清楚是如何把子问题的结果组合成原问题的结果.

Nested List recursion

Nested list recursion是我们在148接触的第一种recursion的考点, 每一个nested list的recursion问题简单来说可以通过使用isinstance来分为两种case:

1. 当前非list的情况.
2.. 当前依然是list的情况.

if not isinstance(obj, list):

常见 Recursion 问题中 可能存在filter condition或者来控制base case的结果. 对于这种类型, 我们基本上可以总结成只需要再base case中进行操作, 例如:

[★, 1, 0]

```
def count_odd (obj)
    """ Return number of odd element in obj.

    Assume: obj is an int or non-empty list with finite nesting depth,
    and obj doesn't contain any empty lists
    """
```

```
if not isinstance(obj, list):
    if obj % 2 == 1:
        return 1
    else: return 0
else: acc = 0
    for sub in obj:
        acc += count_odd(sub)
    return acc.
```

并且, 在recursion问题中, accumulation (combination)环节会根据题上不同要求存在不同的方法，例如求depth, 每次是取sub problem 的max 加1,

```
def depth(obj):
    """Return 0 if obj is a non-list, or 1 + maximum
    depth of elements of obj, a possibly nested
    list of objects.
    """
    if not isinstance(obj, list):
        return 0.

    else: acc = [0]   # handle empty list,
        for sub in obj:
            acc.append(depth(sub)).
        return max(acc) + 1
```

```
def count_lists(list_):
    """
    Return the number of lists, including list_ itself, contained
    in list_.

    @param list list_: list to count lists in
    @rtype: int

    >>> count_lists([])
    1
    >>> count_lists([5, [1, [2, 3], 4], 6])
    3
    """
    if not isinstance(list_, list):
        return 0.

    else:
        acc = 1
        for sub in obj
            acc += count_lists(sub)
        return acc
```

在nested list recursion中, 也会出现对list进行modification的题型, 比如:

```
def add_one(obj):
    """Add one to every element in obj.

    >>> L = [3, [7, 1], 5]
    >>> add_one(L)
    >>> L
    [4, [8, 2], 6]
    """
```

```
if isinstance(obj, list):
    for i in range(len(obj)):
        if isinstance(obj[i], list):
            add_one(obj[i])
        else:
            obj[i] += 1
```

```
def concatenate_flat(list_):
    """
    Return the concatenation, from left to right, of strings contained
    in flat (depth 1) sublists contained in list_, but no other strings.

    Assume all non-list elements of list_ or its nested sub-lists are
    strings

    @param list list_: possibly nested sub-list to concatenate from
    @rtype: str

    >>> concatenate_flat(["five", [["four", "by"], "three"], ["two"]])
    'fourbytwo'
    """
```

'-'.join(['a','b'])

'a-b'

```
if helper(list_):
    return ''.join(list_)
else:
    acc = ''
    for sub in list_:
        if isinstance(sub, list):
            acc += concatenate_flat(sub)
    return acc
```

```
def helper(L):
    for item in L:
        if isinstance(item, list):
            return False
    return True
```

```
def width(list_, max_depth):
    """ Return the maximum number of items that occur at the same depth
    in list_ or its sub-lists combined.  These could be list or non-list
    items. Elements may be lists or non-lists.

    @param list[list|object] list_: a possibly nested list
    @param max_depth int: maximum depth of list_
    @rtype: int

    >>> list_ = [0, 1]
    >>> width(list_, 1)
    2
    >>> list_ = [[0, 1], 2, [3, [[], 4]]]
    >>> width(list_, 4)
    4
    >>> # 4 elements: 0, 1, 3, [[], 4]
    """
```

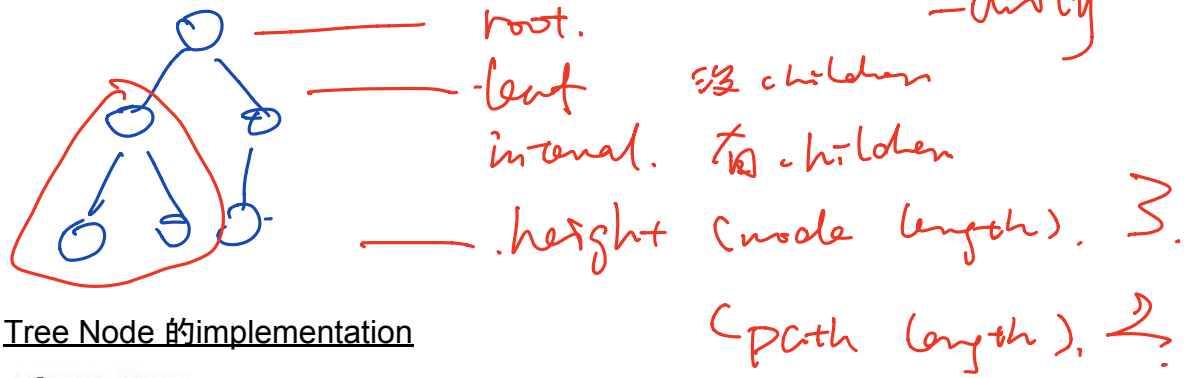
{level: length}

[☆, 0, 0]

```
def helper(list_, max_depth):
    if not isinstance(list_, list):
        return {0:1}.
    else:
        acc = {0:1}.
        for sub in list_:
            temp = helper(sub, max_depth-1).
            for depth in temp:
                acc[depth+1] = acc.get(depth+1,0) +
                               temp[depth].
        return acc

return max(helper(list_, max_depth).values())
```

d.get(key, default).

Section 7 Tree

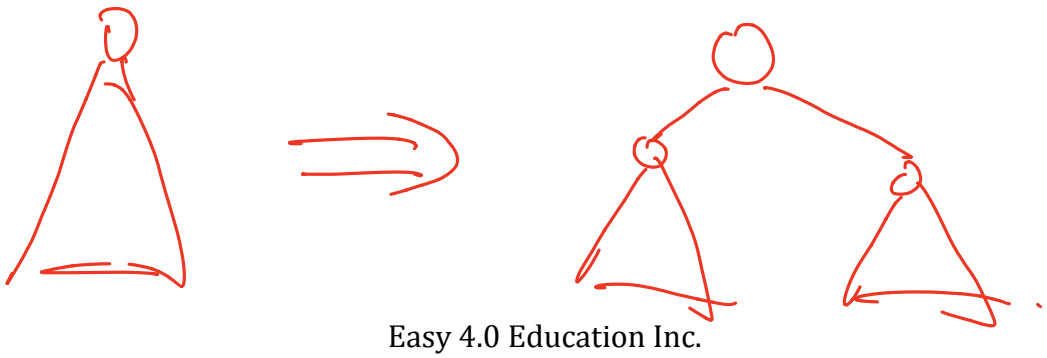


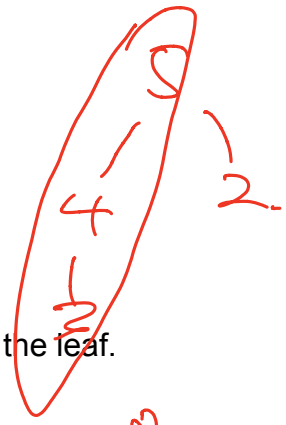
Tree Node 的implementation

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.
    === Attributes ===
    @param object value: value of root node
    @param list[Tree|None] children: child nodes
    """
    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children
        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree|None] children: possibly-empty list of children
        @rtype: None
        """
        self._value = value
        # copy children if not None
        # NEVER have a mutable default parameter...
        self._children = children[:] if children is not None else []
```

Tree, 一个倒过来的树, 每一个tree node都有 value和children, 常见的basecase就是在没有children的时候. 如果有children, 那么要对于每个children进行recursion. 这里要注意, 类似于linkedlist, 每一个node即是一个node, 同时也代表着整个subtree.我们对一个tree object做recursion的时候, 我们可以分为两步:

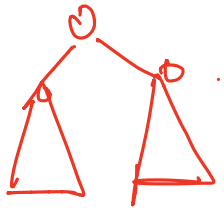
1. 对node的value进行操作.
2. 对 每一个child node都 recursively 操作.





Path 相关题型:

```
def get_longest_path(t: Tree) -> List[obj]:
    """Return a list contains values of the longest path from node t to the leaf.

    >>> get_longest_path(Tree(5, [Tree(4, Tree(3)), Tree(2)]))
    [5, 4, 3]
    """
    if not t.children:
        return [t.value]

    else:
        result = []
        for c in t.children:
            path = get_longest_path(c)
            if len(result) < len(path):
                result = path[:]
        return [t.value] + result
```



```
def get_all_paths(t: Tree) -> List[obj]:
    """Return a list of lists contain values of the longest path from node t to all leaves.

    >>> get_all_paths(Tree(5, [Tree(4, Tree(3)), Tree(2)]))
    [[5, 4, 3], [5, 2]]
    """
    if len(t.children) == 0:
        return [[t.value]]

    else:
        acc = []
        for c in t.children:
            for path in get_all_paths(c):
                acc.append([t.value] + path)

        return acc
```

set() {0} add. intersection difference. union.

```
def pathlength_sets(t):
    """
    Replace the value of each node in Tree t by a set containing all
    path lengths from that node to any leaf.  A path's length is the
    number of edges it contains.

    @param Tree t: tree to record path lengths in
    @rtype: None

    >>> t = Tree(5)
    >>> pathlength_sets(t)
    >>> print(t)
    {0}
    >>> t.children.append(Tree(17))
    >>> t.children.append(Tree(13, [Tree(11)]))
    >>> pathlength_sets(t)
    >>> print(t)
    {1, 2}
      {0}
      {1}
        {0}
    """
```

5 {2, 1}. {0} {1} {0}.

```
if not t.children:
    t.value = {0}
else:
    acc = set()
    for c in t.children:
        pathlength_sets(c).
        for n in c.value:
            acc.add(n+1)
    t.value = acc
```

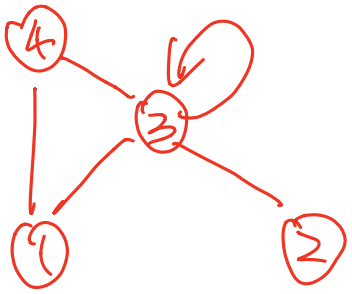
Read the declaration of class Tree and the docstring for function **unique_paths**. Then implement (write the body for) function **unique_paths** on the next page.

Hint: You may conclude there are unique paths in a tree if you traverse (visit) every node without finding a node that has been visited twice. A set is a convenient data structure for recording objects that have been seen.

```
class Tree:
    def __init__(self, value=None, children=None):
        ''' (Tree, object, list of Tree) -> NoneType

        Create Tree(self) with content value and 0 or more Tree children.
        '''
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []


def unique_paths(t):
    ''' (Tree) -> bool

    Return whether there is a unique path from t to each
    of its descendents.

    Assume that two Trees are the same if they have the same
    memory address, that is id(t1) == id(t2)

    >>> t1 = Tree(1)
    >>> t2 = Tree(2)
    >>> t3 = Tree(3, [t1, t2])
    >>> unique_path(t3)
    True
    >>> t4 = Tree(4, [t3, t1])
    >>> unique_path(t4)
    False
    >>> t3.children.append(t3)
    >>> unique_path(t3)
    False
    '''
```

Part (a) [2 MARKS]

Draw the tree rooted at t4, as it is after all the doctest code has been executed.

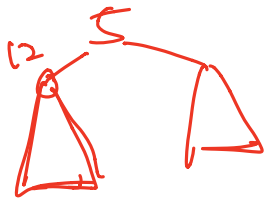


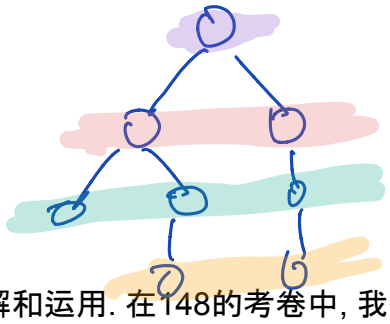
def unique_path(t):

```
return helper(t).
seen = set().
def helper(t):
    if t in seen:
        return False.
    if not t.children:
        seen.add(t).
        return True.
    else: seen.add(t).
        for c in t.children:
            if not helper(c)
                return False
    return True.
```

```
def has_path_sum(t,total):
    '''Given binary tree, t return a boolean value
    indicating whether there is a path in t to any
    leaf whose values add up to int total. Precondition: t
    is not None, total is integer.'''
    if not t.children:
        return t.value == total
    else: for c in t.children:
        if has_path_sum(c, total - t.value):
            return True
    return False.
```

17.





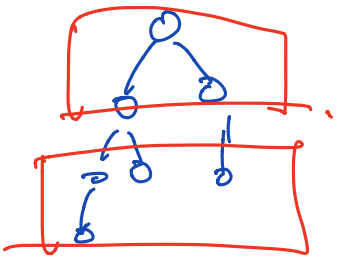
层层叠叠相关题:

在树的问题中, 我们经常会考察关于层数 depth / level 的相关理解和运用. 在148的考卷中, 我们经常默认root所在的是depth 0. 会出现两种常见题型:

1. 针对某一特定depth之上, 之下, 或者这一层进行操作.

```
def count_above_depth(t: Tree, d) -> number:
    """Return the number of nodes above the depth in t.

    >>> count_above_depth(Tree(5, [Tree(4, Tree(3)), Tree(2)]), 2)
    3
    """
    if d <= 0:
        return 0

    else:
        acc = 1
        for c in t.children:
            acc += count_above_depth(c, d-1)

        return acc
```

```
def count_below_depth(t: Tree, d) -> number:
    """Return the number of nodes above the below in t.

    >>> count_above_depth(Tree(5, [Tree(4, Tree(3)), Tree(2)]), 1)
    1
    """
    acc = 0
    if d < 0:
        acc += 1

    for c in t.children:
        acc += count_below_depth(c, d-1)

    return acc
```

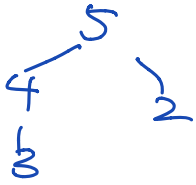


2. 针对奇数或者偶数层进行操作.

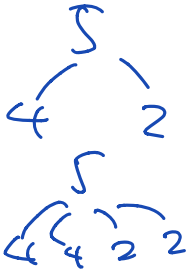
```
def count_on_odd_depth(t: Tree) -> number:
    """Return the number of nodes on odd depth.

    >>> count_above_depth(Tree(5, [Tree(4, Tree(3)), Tree(2)]), 1)
    1
    """
```

```
acc = len(t.children)
for c in t.children:
    for g in c.children
        acc += count_on_odd_depth(g).

return acc
```

```
def duplicate_even_level(t: Tree) -> None:
    """Duplicate all nodes on even level by adding a new one after each of them.

    >>> t = Tree(5, [Tree(4, Tree(3)), Tree(2)])
    >>> duplicate_even_level(t)
    >>> t
    Tree(5, [Tree(4, Tree(3)), Tree(4), Tree(2), Tree(2)])
    """
```

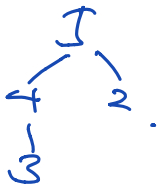


```
def helper(t, do):
    for c in t.children:
        helper(c, not do)

    if do:
        t.children = [Tree(t.value, t.children)]

helper(t, False)
```



3. 针对每一层进行的操作

```
def count_by_level(t: Tree) -> List[int]:
    """Return a list that counts number of nodes on each level, index 0 is the root.

    >>> t = Tree(5, [Tree(4, Tree(3)), Tree(2)])
    >>> count_by_level(t)
    [1, 2, 1]
    """
    if not t.children:
        return [1]
    else: acc=[1]
        for c in t.children:
            temp = count_by_level(c)
            for i in range(len(temp)):
                if i+1 >= len(acc):
                    acc.append(temp[i])
                else:    acc[i+1] += temp[i].
    return acc
```



Modification相关题:

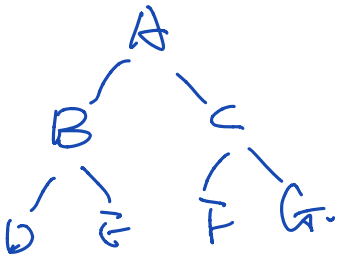
```
def extend_every_third_level(t) -> None:
    """ Add a new node after the node on every third level.

    >>> t = Tree(5, [Tree(4, Tree(3, Tree(7))), Tree(2)])
    >>> extend_every_third_level(t)
    >>> t
    Tree(5, [Tree(5, Tree(4, Tree(3, Tree(7, [Tree(7)])))), Tree(2))])
    """
    def helper(t, level):
        for c in t.children:
            helper(c, level+1)
        if level % 3 == 0:
            t.children = [Tree(t.value, t.children)]

    helper(t, 1).
```

```
def prune(t, predicate):
    """ Return a new tree with the same values as t, except it prunes (omits) all paths of t that start
    with nodes where predicate(node.value) == False. If predicate(t.value) == False, then prune
    returns None.

    Assume that all values in t are ints, and that predicate always takes an int and returns a bool.

    @param Tree t: tree to prune
    @param function[int, bool] predicate: predicate on t's values.
    @rtype: Tree|None

    >>> t1 = Tree(6, [Tree(8), Tree(9)])
    >>> t2 = Tree(4, [Tree(11), Tree(10)])
    >>> t3 = Tree(7, [Tree(3), Tree(12)])
    >>> t = Tree(5, [t1, t2, t3])
    >>> t3_pruned = Tree(7, [Tree(12)])
    >>> def predicate(v): return v > 4
    >>> prune(t, predicate) == Tree(5, [t1, t3_pruned])
    True
    """
```

```
if not predicate(t.value):
    return None.

else:
    acc = []
    for c in t.children:
        new_c = prune(c).
        if new_c:
            acc.append(new_c)

return Tree(t.value, acc)
```

Section 8 Iterative instead of Recursion

我们可以通过使用我们在148中学过的Stack 和 Queue 两种不同的ADT 来对一个Tree进行操作, 在这里要注意两件事情, Stack 会得到一个按照path 顺序进行 recursion的结果, 而 Queue会得到一个按照level 顺序 得到的结果.




```
def mark_tree_by_depth(t) -> None:
    """ Mark the nodes in the tree using numbers by depth

    >>> t = Tree('A', [Tree('B', Tree('C')), Tree('D')])
    >>> mark_tree_by_depth(t)
    >>> t
    Tree(1, [Tree(3, Tree(4)), Tree(2)])
    """
    s = Stack()
    num = 1
    s.add(t)
    while not s.is_empty():
        temp = s.remove()
        temp.value = num
        num = num + 1
        for c in temp.children[::-1]
            s.add(c)
```

```
def mark_tree_by_level(t) -> None:
    """ Mark the nodes in the tree using numbers by level

    >>> t = Tree('A', [Tree('B', Tree('C')), Tree('D')])
    >>> mark_tree_by_depth(t)
    >>> t
    Tree(1, [Tree(2, Tree(4)), Tree(3)])
    """
    q = Queue()
    q.add(t)
    num = 1
    while not q.is_empty():
        temp = q.remove()
        temp.value = num
        num = num + 1
        for c in temp.children:
            q.add(c)
```

Section 4. Binary Tree

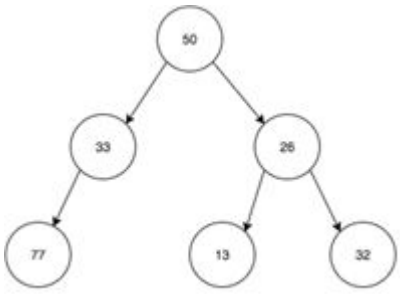
Binary Tree, 顾名思义, arity最大只有2的Tree, 但是, 我们的implementation有改变，不再是node, 变成了 left和right. (虽然我们还是可以 for child in [t.left, t.right]).

需要Check None

```
class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.
    """
    def __init__(self, value, left=None, right=None):
        """
        Create BinaryTree self with value and children left and right.
        @param BinaryTree self: this binary tree
        @param object value: value of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.value, self.left, self.right = value, left, right
```

Traversals:

我们一共接触了三种traversal, pre-order, in-order和post-order. 这里pre, in和post可以理解为 root放的位置. Pre-order也就是先root再children, in-order为root在左右children中间, post-order为root在最后. 注意,



Pre-order: 50 33 77 26 13 32.

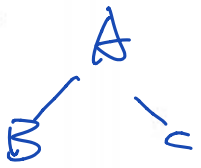
Post-order: 77 33 13 32 26 50.

In-order: 77 33 50 13 26 32.

```
def inorder(node):
    if not node:
        return []
    else:
        return inorder(node.left) +
               [node.value] +
               inorder(node.right).
```

Here is some stack-based code that implements one of the tree traversals. Which traversal is this? Trace the code on a sample tree to support your choice.

```
def mystery_order(bt: BTNode) -> None:
  s = Stack()
  current = bt
  while current or not s.is_empty():
    if current:
      s.push(current)
      current = current.left
    else:
      current = s.pop()
      print(current.item)
      current = current.right
```



B.
A.
C

Inorder.

```
def occurs(root, s):
    ''' (BTNode or None, str) -> bool

    Return whether or not s equals a sequence of characters
    along some path from the root to a leaf, inclusive, and in
    that order. The empty str ("") is considered to occur in
    the empty tree, denoted None.

    Assume each node in the tree rooted at root contains a str of length 1.

    >>> left = BTNode('b', None, BTNode('d', BTNode('e'), None))
    >>> right = BTNode('c', BTNode('e'), BTNode('f', BTNode('h'), BTNode('i')))
    >>> whole = BTNode('a', left, right)
    >>> occurs(whole, 'acfh')
    True
    >>> occurs(whole, 'ace')
    True
    >>> occurs(whole, 'bde')
    False
    '''
```

```
if not root:
    return s == ""

else:
    if root.value == s[:1]:
        return occurs(root.left, s[1:]) or
               occurs(root.right, s[1:])
    return False.
```

A **perfect binary tree** is one in which (a) every non-leaf node has exactly two children, and (b) all leaves occur at the same level. Notice that a tree consisting of a single node is a perfect binary tree of height 1.

An empty tree, represented by None, has is a perfect binary tree of height 0.

Part (a) [3 MARKS]

Recall that we defined height of a tree in such a way that a tree containing just a root node has height one. Draw a perfect binary tree of height 3.

Draw a binary tree of height 4 this time (not 3) that is not perfect because it satisfies condition (a) but not condition (b).

Draw a binary tree of height 4 that is not perfect because it satisfies condition (b) but not condition (a).

Write the body of function TPBT.

```
def TPBT(root):
    ''' (BTNode or None) -> (int, bool)

    Return a tuple containing: (1) the height of the tallest perfect binary
    tree within the tree rooted at root, and (2) whether or not that tallest
    perfect binary tree occurs at the root itself.
    '''
```