Recursion.

① description

② Base Case.

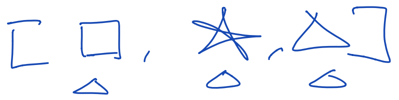
③ Recursive Step.

- Divide
- Combine.

Nested List.

[□ , ☆ , △] .

- Base Case:
  最少层数.
- Recursive.
  - Divide.
    减少层数.
  - combine
    (□ ☆ . △)



```
def depth(obj):
    if not isinstance(obj):
        return 0

    else:
        acc=[]    # [0]
        for sub in obj:
            acc.append(depth(sub)).

        return max(acc) + 1.
```

[[2, 3], 4] .

[[]] [[0]]

```
def contains(obj, item)
        (obj, list):
```

```
if not isinstance(ob),
    return obj == item.

else:
    acc = [ ]
    for sub in obj
        acc.append(contains(sub, item)).
    return any(acc)
                   ← True in acc.

    for sub in obj:
        if contains(sub, item):
            return True.

    return False.



def count(obj).
    if not isinstance(obj, list):
        if ________:
            return 1
        else: return 0
    else:
        acc = 0
        for sub in obj:
            acc += count(sub)
        return acc.
```

```
def equal(obj1, obj2):
    # Base Cases.
    if type(obj1) != type(obj2):
        return False.

    elif not isinstance(obj1, list):
        return obj1 == obj2

    # lists.
    else:
        if len(obj1) != len(obj2):
            return False

        else:
            for i in range(len(obj1)):
                if not equal(obj1[i], obj2[i]):
                    return False

            return True.


def gather(obj):
    if not isinstance(obj, lost):
        return [obj]

                                        a = a + ['1]

    else:
              [ ]
```

```
acc = []
for sub in obj:
    acc += gather(sub)
return acc
```

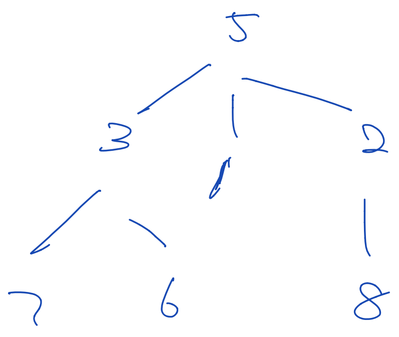
```
def avg(obj):
    return get_sum(obj) / count(obj)
```

helper.

```
def floor_to_int(lst):
    if not isinstance(lst):
        return

    for i in range(len(lst)):
        if not isinstance(lst[i], list):
            lst[i] = int(lst[i])
        else: floor_to_int(lst[i]).
```

```
def list_level(obj, d):      [1, [2, [3,4], 5], 2].
    if d == 0:                  1   2   3 3   2   1
        if isinstance(obj, list):
            return [] .
        else: return [obj]

    else:
        if isinstame(obj, list):
            acc=[]
            for sub in obj:
                acc += list_level(Sub, d-1)
            return acc

        else: return []


def list_level_above(obj, d):
    if d==0:
        return []
    else:    # d >= 0
        if not isinstame(obj, list):
            return [obj]

        else: acc = []
            for sub in obj:
                acc += list_level_above(sub, d -1)
            return acc.
```

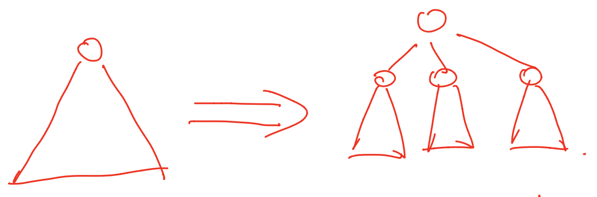
Tree.



root.
leaf
internal node.
height.

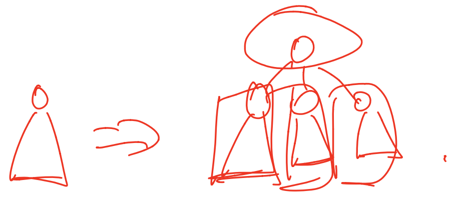
```
class Tree:
    def __init__(self, value=None, children=None):
        self.value = value.
        self.children = children.copy() if children else []
```

Base Case: leaf
Recursive Step:



root represents subtree.

```
def height (t):
    #Base Case: leaf.
    if not t.children:
        return 1

    # Recursion.
    else:
        acc = []
        for c in t.children:
            acc += [ height( c) ] .
        return max(acc) + 1.


def count( t):
    if not t.children:
        return 1.

    else:
        acc = 1.
        for c in t.children:
            acc += count(c)

        return acc.
```

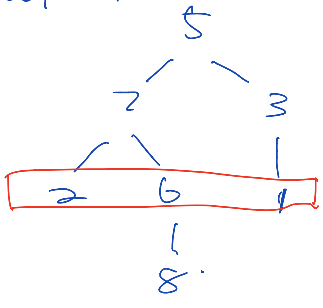


```
# handle root
acc = 1.

# handle subtrees. :
for c in t.children
    acc += count(c).

return acc.
```

```
def count_internal(t):
  if not t.children:
      return 0

  else:
      acc= 1.
      for c in t.children:
          acc += count_internal(c).
      return acc


def gather(t):
  if not t.children:
      return [t.value].

  else:
      acc= [t.value]
      for c in t.children:
          acc += gather(c).
      return acc
```

root at depth 0.

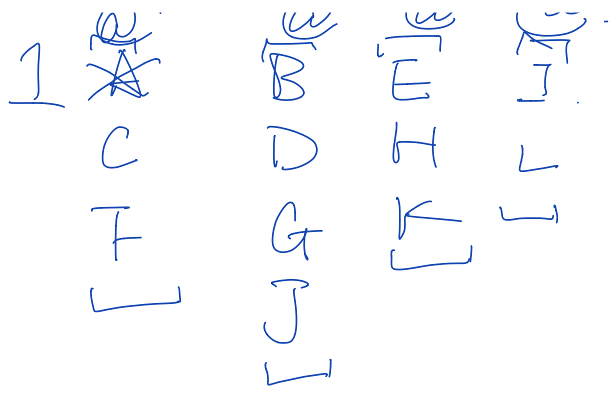
```
def count_at_level(t,d):
  if d==0:
      return 1.
```



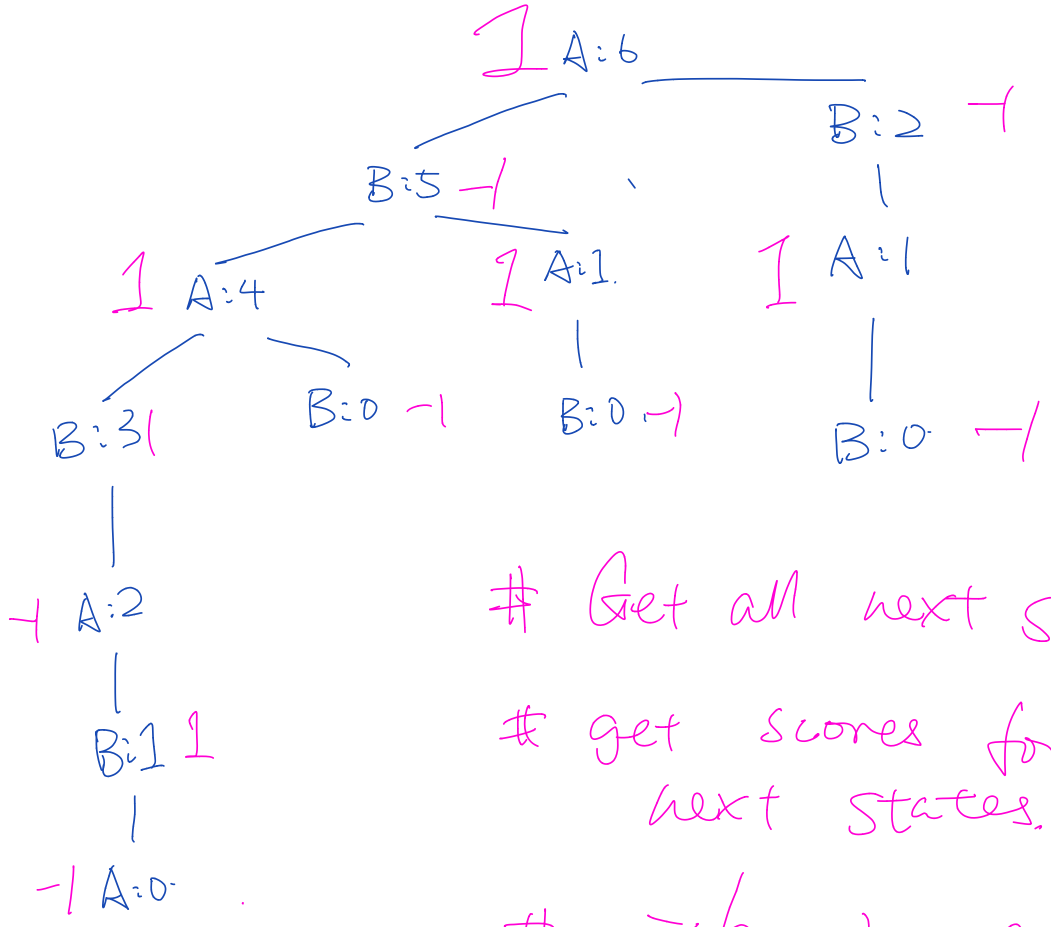
```
else:   # d >= 1.
    acc = 0
    for c  in  t.children.
        acc += count_at_level (t, d-1)
    return  acc

def count_below_level (t, d):
    if not  t.children:
        if d < 0:
            return 1
        else: return 0

    else:
        acc = 0
        if d < 0:
            acc += 1
        for c  in t.children.
            acc += count_below_level(c, d-1)
        return  acc
```

[illegible]

1 ~~@~~[~~A~~, B].
@[C , D , E]
@[F G H I]
@[J , K L]



" {} , {}, {}. // {}\n {} \n"



```
# Get all next states.
# get scores for
  next states. * -1
# pick max
```