Sliding Windows SubLists

```
for i in range(len(nums)):
    for j in range(i+1,len(nums)):
```

Get all Subsets

```
#Time Space = O(N×2^N)
for i in range(0, len(nums) + 1):  # to get all lengths: 0 to 3
    for subset in itertools.combinations(nums, i):
        print(list(subset))
```

sortTable BigtoSmall

```
   values = list(self.scoreBoard.values())
    return sum(sorted(values, reverse=True)[:K])
```

MergeIntervals

```
intervals.sort(key=lambda x:x[0])
```

rotate around array

```
 index  = (currentidx + increment - 1) % len(array)
```

Sorting dict

```
dict(sorted(x.items(), key=lambda item: item[1])) #Sort Values
dict(sorted(x.items(), key=lambda item: item[0])) #Sort Keys
```

Class DLL

```
class DLLNode:
    def __init__(self, val=0, prev=None, next= None):
        self.val = val
        self.prev = prev
        self.next = next
```

Class Tree

```
class Node:
    def __init__(self, val=0, left=None, right=None):#, random=None
        self.val = val
        self.left = left
        self.right = right
        # self.random = random
```

Graph Node

```
class Node:
    def __init__(self, val):
        self.val = val
        self.directions = [] #self.chidren = []
```

LRU Cache Helper

```
    def _add_node(self, node): #Always add the new node right after head.
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):# Remove an existing node from the linked
list.
        prev = node.prev
        new = node.next
        prev.next = new
        new.prev = prev

    def _pop_tail(self):#Pop the current tail.
        res = self.tail.prev
        self._remove_node(res)
        return res
```

Binary Tree

```
def isValidBSTRecursive(root):

    def validate(node, low=-math.inf, high=math.inf):
        # Empty trees are valid BSTs.
        if not node:
            return True
        # The current node's value must be between low and high.
        # i.e for right side lower will be -inf unless it has a left
        if node.val <= low or node.val >= high:
            return False

        # The left and right subtree must also be valid.
        return (validate(node.right, node.val, high) and
                validate(node.left, low, node.val))

    return validate(root)

def isValidBSTIterative(root):
    if not root:
        return True

    stack = [(root, -math.inf, math.inf)]
    while stack:
        node, lower, upper = stack.pop()
        if not node:
            continue
        val = node.val
        if val <= lower or val >= upper:
            return False
        stack.append((node.right, val, upper))
        stack.append((node.left, lower, val))
    return True
```

```
Counter(string)
Counter(string).items()
Counter(string).keys()
Counter(string).values()
Counter(string).most_common(3)
Counter(string).most_common()[:-3-1:-1]
sorted(nums, key=lambda x: (count[x], x), reverse=True)
######################
```

```
    def solve(s, nums):
         total = sum(nums)
         max_val = 0
         curr = None
         for idx, num in enumerate(nums):
             if curr != s[idx]:
                 total -= max_val
                 max_val = nums[i]
                 curr = s[i]
             else:
                 max_val = max(max_val, nums[i])
         total -= max_val

         return total
```

```
# class Tree:
#     def __init__(self, val, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
from collections import defaultdict

def solve(self, root):
    seen = defaultdict(list)
    def level(root, row=0, col=0):
        if root:
            seen[col].append(row)
            level(root.left, 2 * row + 1, col+1)
            level(root.right, 2 * row + 2, col+1)

    level(root)
    best = 0
```

```
        for row in seen.values():
            m = max(row) - min(row) + 1
            best = max(best, m)
        return best
```