449. Serialize and Deserialize BST

We will use pre-order to traverse all nodes and append their value as string, which will be separated by commas. In deserialize process, we will split data and transform each element into integer. Then we will take advantage of bst’s left tree is always less than right tree. So every time we will check each first element in queue is within the boundary, and then if it is, we will append this new node to the current node.

```
from collections import deque
class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        self.res, sep = '', ','
        def pre_order(node):
            if node:
                self.res += str(node.val) + sep
                pre_order(node.left)
                pre_order(node.right)

        pre_order(root)
        return self.res[:-1]


    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        if not data:
            return None
        q = deque(int(i) for i in data.split(','))
        def build(min_val, max_val):
            if q and min_val < q[0] < max_val:
                val = q.popleft()
                node = TreeNode(val)
                node.left = build(min_val, val)
                node.right = build(val, max_val)
                return node
            return None
        return build(-float('inf'), float('inf'))
```

167. Two Sum II - Input array is sorted

We will use two pointers way starting from two ends. If the sum of two numbers is less than target, we will move left to right by 1, if larger, we will move right index to left by one. Once they are equal, we will return [left + 1, right + 1], TC is O(n)

```
class Solution(object):
    def twoSum(self, numbers, target):
        """
        :type numbers: List[int]
        :type target: int
        :rtype: List[int]
        """
        left, right = 0, len(numbers) - 1
        while left < right:
            cur = numbers[left] + numbers[right]
            if cur == target:
                return [left + 1, right + 1]
            elif cur > target:
                right -= 1
            else:
                left += 1
```

364. We will use dfs to traverse all nested integers. We will use a dict to store each element presenting time. Key is (num, layer), value is presenting time. In the end, we will find max_layer and get the final layer by max_layer - layer. And it would be very easy to get the result.

```
from collections import defaultdict
class Solution:
    def depthSumInverse(self, nestedList: List[NestedInteger]) -> int:
        visited = defaultdict(int)
        result = 0
        for l in nestedList:
            if l.isInteger():
                visited[(l.getInteger(), 0)] += 1
            else:
                self.dfs(visited, l, 0)

        max_layer = max(map(lambda a: a[1], visited.keys())) + 1 if len(visited.keys()) > 0 else 0
        for k, v in visited.items():
            num, layer = k
            result += (max_layer - layer) * num * v
        return result
    def dfs(self, visited, l, layer):
        if l.isInteger():
            visited[(l.getInteger(), layer)] += 1
        else:
            for n in l.getList():
                self.dfs(visited, n, layer + 1)
```

706. Design HashMap

We will use chain to implement this hashmap. We will use an array to record these key-value pair. We will divide key by 1000(assume), and get the index of array that we will put our node, It there is one ,we will append it behind this node. The same as get and remove. TC is O(n) almost.

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
class MyHashMap:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.m = 1000
        self.map = [None] * self.m

    def put(self, key: int, value: int) -> None:
        """
        value will always be non-negative.
        """
        my_key = key % self.m
        if self.map[my_key]:
            node = self.map[my_key]
            if node.key == key:
                node.value = value
                return
            while node.next:
                if node.next.key == key:
                    node.next.value = value
                    return
                node = node.next
            node.next = Node(key, value)
        else:
            self.map[my_key] = Node(key, value)

    def get(self, key: int) -> int:
        """
        Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key
        """
        my_key = key % self.m
        node = self.map[my_key]
        if not node:
            return -1
        else:
            while node:
                if node.key == key:
                    return node.value
                node = node.next
            return -1

    def remove(self, key: int) -> None:
        """
        Removes the mapping of the specified value key if this map contains a mapping for the key
        """
        my_key = key % self.m
        node = self.map[my_key]
        if node:
            if node.key == key:
                self.map[my_key] = node.next
            else:
                while node.next:
                    if node.next.key == key:
                        node.next = node.next.next
                        return
                    node = node.next
```

1162. As Far from Land as Possible

We will start from all lands and using bfs to seek seas around(top, down, left, right). In each round, we will add our distance by 1. When there is no nodes available, We will return our distance. TC is O(n)

```
class Solution:
    def maxDistance(self, grid: List[List[int]]) -> int:
        visited = set()
        cur = set()
```

```
rows, cols = len(grid), len(grid[0])
length = 0

for i in range(rows):
  for j in range(cols):
    if grid[i][j] == 1:
      cur.add((i, j))
      visited.add((i, j))
if len(cur) == rows * cols or len(cur) == 0:
  return -1

while cur:
  next_ite = set()
  for x, y in cur:
    for d_x, d_y in [[0, -1], [0, 1], [1, 0], [-1, 0]]:
      new_x, new_y = x + d_x, y + d_y
      if 0 <= new_x < rows and 0 <= new_y < cols and (new_x, new_y) not in visited:
        next_ite.add((new_x, new_y))
        visited.add((new_x, new_y))
  if len(next_ite) == 0:
    return length
  length += 1
  cur = next_ite
```