146. LRU Cache
We will use a double linked list to store every key-value pair so that delete and insert nodes to maintain that linkedlist. We will use cache to store nodes. So we could check key-node pair. When we want to get key, we will check our cache, if it exists, we will get node and put it in the head of our linked list, if not, we will return -1.. When we want to put key-value, we will check whether it exists, if it does, we will remove this node and add it in the head. We will renew the value at the same time. If it doesn't, we will check whether linked list's length exceeds capacity. If it does, we will remove the last node. In the end, we will create a new node and add it to head. TC is O(1)

```
class Node:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.next = None
        self.prev = None

class LRUCache:

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.dummy = Node(0, 0)
        self.cache = {}
        self.last = None

    def get(self, key: int) -> int:
        print
        if key in self.cache:
            node = self.cache[key]
            if node == self.last and node.prev != self.dummy:
                self.last = node.prev
            self.removeNode(node)
            self.insertNode(node)
            return node.val
        return -1


    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            node = self.cache[key]
            if node == self.last and node.prev != self.dummy:
                self.last = node.prev
            self.removeNode(node)
```

```
            node.val = value
            self.insertNode(node)
        else:
            if len(self.cache.keys()) == self.capacity:
                if self.last:
                    node = self.last
                    if self.last.prev != self.dummy:
                        self.last = self.last.prev
                    else:
                        self.last = None
                    self.removeNode(node)
                    del self.cache[node.key]
            new_node = Node(key, value)
            self.insertNode(new_node)
            self.cache[key] = newNode
            if not self.last:
                self.last = new_node




    def insertNode(self, node):
        node.prev = self.dummy
        node.next = self.dummy.next
        self.dummy.next = node
        if node.next:
            node.next.prev = node

    def removeNode(self, node):
        node.prev.next = node.next
        if node.prev.next:
            node.prev.next.prev = node.prev
```

238. Product of Array Except Self

We will multiply previous one elements in the array and store the result in our res array, Then we will scan from right to left and get muliply right products to current result. TC is O(n), SC is O(1)

```
class Solution(object):
    def productExceptSelf(self, nums):
        """
        :type nums: List[int]
```

```
        :rtype: List[int]
        """
        result = [1]
        product = 1
        for num in nums[:-1]:
            result.append(result[-1] * num)

        for i in range(len(result) - 1, -1, -1):
            result[i] *= product
            product *= nums[i]
        return result
```

34. Find First and Last Position of Element in Sorted Array
We will use binary search to find left index and right index. We will check whether number exists in the array, if not, we will return [-1, -1]. We will return [left_index, right_index - 1]. TC is O(logn)

```
from bisect import *
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        length = len(nums)
        right_index = bisect(nums, target)
        left_index = bisect_left(nums, target)
        if left_index == length or nums[left_index] != target:
            return [-1, -1]
        return [left_index, right_index - 1]
```

708. Insert into a Cyclic Sorted list
If head is None, we will return new node. Then we will use prev = head, next = head.next. When we will iterate through cycled linked list. When we find insertVal > prev.val > next.val or insertVal < prev.val < next.val or prev.val >= insertVal >= next.val. We will insert our node. If not we will pick any place and insert our node. TC is O(n)

```
class Solution:
    def insert(self, head: 'Node', insertVal: int) -> 'Node':
        if not head:
            head = Node(insertVal, None)
            head.next = head
            return head
        prev = head
        next = head.next
        inserted = False

        while True:
            if prev.val <= insertVal <= next.val or prev.val > next.val > insertVal or insertVal > prev.val
> next.val:
```

```
                prev.next = Node(insertVal, next)
                inserted = True
                break

            prev = prev.next
            next = next.next
            if prev == head:
                break

        if not inserted:
            prev.next = Node(insertVal, next)
        return head
```

205. Isomorphic Strings

We will iterate through string and use hashmap to store index, if there is no such key, we will store it, if not, we will set this value to this index. We will compare letter array. Return True or False. TC is O(n)

```
class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:
        index_s, index_t = self.get_index_array(s), self.get_index_array(t)
        return index_s == index_t

    def get_index_array(self, s):
        index_s = []
        map_s = {}
        for idx, w in enumerate(s):
            if w in map_s:
                index_s.append(map_s[w])
            else:
                index_s.append(idx)
                map_s[w] = idx
        return index_s
```