1. Two Sum

The basic thought is that we only need to store element and their indice. Then when we iterate through elements later, we only need to check in memo to see whether the difference between target and current number exist. If so, we find what we want. Time complexity is O(n) and space complexity is O(n) because of memo.

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        memo = {}
        for i, num in enumerate(nums):
            dif = target - num
            if dif in memo:
                return [memo[dif], i]
            else:
                memo[num] = i
```

2. Add Two Numbers

This question actually mimic how we add two numbers. We only need to add two numbers one digit by one digit and never forget the carry digit. We will set reminder as new node's value and quotient as carry digit's number.

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        l3 = ListNode(0)
        l3_memo = l3
        carry = 0
        while l1 and l2:
            carry, res = divmod(l1.val + l2.val + carry, 10)
            l3.next = ListNode(res)
            l3 = l3.next
            l1 = l1.next
            l2 = l2.next

        restNode = l1 if l1 else l2
        while restNode:
            carry, res = divmod(restNode.val + carry, 10)
            l3.next = ListNode(res)
            l3 = l3.next
            restNode = restNode.next
        if carry > 0:
            l3.next = ListNode(carry)

        return l3_memo.next
```

146.LRU

For this question, we could take advantage of OrderedDict in python, which will remember the order of node inserting to it. When we want to get key, we just remove that key, value pair and readd it. When we want to set it, we do the same way if there is the same key available. In other way, we will check the whether the length has passed the capacity, if it does, we delete the first one we added. Then add a new key, value pair.

If we don't use OrderedDict, we could use doubleLinked List and map to store linked nodes. Every we want to get key, we will retrieve node through map, and delete that node and append it to the original linkedlist. When we want to set new key, value pair, we would check whether that key exists, if it does, we do the same way as get, if not, we will delete the first node if its length exceeds the capacity. In the end append new node in the original Linkedlist and store its key, node in the map.

```
from collections import OrderedDict
class LRUCache:

    def __init__(self, capacity: int):
        self.memo = OrderedDict()
        self.capacity = capacity

    def get(self, key: int) -> int:
        if key in self.memo:
            temp_result = self.memo[key]
            del self.memo[key]
            self.memo[key] = temp_result
            return self.memo[key]
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.memo:
            del self.memo[key]
            self.memo[key] = value
        else:
            if len(self.memo) == self.capacity:
                self.memo.popitem(last=False)
            self.memo[key] = value
```

5. Longest Palindromic Substring

Basically, we just iterate from beginning to end, for every letter, we are gonna check from center to both sides to find longest palindrome. And store the longest palindrome in our result. Also we need to consider both situations when string's length is odd or even.

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        max_length = 0
        result = ""
        for i in range(len(s)):
            longest_palindrome = self.helper(s, i, i)
            if len(longest_palindrome) > max_length:
                result = longest_palindrome
                max_length = len(longest_palindrome)
            longest_palindrome = self.helper(s, i, i + 1)
            if len(longest_palindrome) > max_length:
                result = longest_palindrome
                max_length = len(longest_palindrome)
        return result

    def helper(self, s, start, end):
        while start >= 0 and end < len(s) and s[start] == s[end]:
            start -= 1
            end += 1
        return s[start + 1:end]
```

973. K Closet Points to Origin

For this question, we only need to sort the original array and return the first K elements. Or we could use heapq.nlargest to get K closet points.

```
class Solution:
    def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
        points.sort(key=lambda a: a[0] ** 2 + a[1] ** 2)
        return points[0:K]
```