3. 无重复字符的最长子串

```
func lengthOfLongestSubstring(s string) int {
    longest, n := 0, len(s)
    freq := make(map[byte]int, n) // 哈希集合记录每个字符出现次数
    for i, j := 0, 0; j < n; j++ {
        freq[s[j]]++         // 首次出现存入哈希
        for freq[s[j]] > 1 { // 当前字符与首字符重复
            freq[s[i]]-- // 收缩窗口，跳过重复首字符
            i++          // 向后扫描
            if freq[s[j]] == 1 { // 优化：如果无重复退出循环
                break
            }
        }
        if longest < j-i+1 { // 统计无重复字符的最长子串
            longest = j - i + 1
        }
    }
    return longest
}
```

206. 反转链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */

func reverseList(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        next := curr.Next
        curr.Next = prev
        prev = curr
        curr = next
    }
    return prev
}

func reverseList1(head *ListNode) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }
    dummy := &ListNode{Next: head}
    curr := head
    for curr.Next != nil {
        next := curr.Next
        curr.Next = next.Next  // 连接后继节点
        next.Next = dummy.Next // 反转（头插）
        dummy.Next = next      // 通知哨兵节点（前驱）
    }
    return dummy.Next
}


func reverseList1(head *ListNode) *ListNode {
    dummy := &ListNode{Next: head}
    curr := head
    for curr != nil && curr.Next != nil {
        next := curr.Next
        curr.Next = next.Next
        next.Next = dummy.Next
        dummy.Next = next
    }
    return dummy.Next
}


func reverseList2(head *ListNode) *ListNode {
    if head == nil || head.Next == nil { // 没有节点或只有一个节点
        return head // 递归出口
    }
    newHead := reverseList(head.Next)
    head.Next.Next = head // 反转
    head.Next = nil
    return newHead
}
```

146. LRU 缓存机制

```
type LRUCache struct {
    cache          map[int]*DLinkedNode
    head, tail     *DLinkedNode
    size, capacity int
}
```

```go
type DLinkedNode struct {
    key, value int
    prev, next *DLinkedNode
}

func initDLinkedNode(key, value int) *DLinkedNode {
    return &DLinkedNode{
        key:   key,
        value: value,
    }
}

func Constructor(capacity int) LRUCache {
    l := LRUCache{
        cache:    map[int]*DLinkedNode{},
        head:     initDLinkedNode(0, 0),
        tail:     initDLinkedNode(0, 0),
        capacity: capacity,
    }
    l.head.next = l.tail
    l.tail.prev = l.head
    return l
}

func (this *LRUCache) Get(key int) int {
    if _, ok := this.cache[key]; !ok {
        return -1
    }
    node := this.cache[key] // 如果 key 存在，先通过哈希表定位，再移到头部
    this.moveToHead(node)
    return node.value
}

func (this *LRUCache) Put(key int, value int) {
    if _, ok := this.cache[key]; !ok { // 如果 key 不存在，创建一个新的节点
        node := initDLinkedNode(key, value)
        this.cache[key] = node // 添加进哈希表
        this.addToHead(node)   // 添加至双向链表的头部
        this.size++
        if this.size > this.capacity {
            removed := this.removeTail()    // 如果超出容量，删除双向链表的尾部节点
            delete(this.cache, removed.key) // 删除哈希表中对应的项
            this.size--
        }
    } else { // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
        node := this.cache[key]
        node.value = value
        this.moveToHead(node)
    }
}

func (this *LRUCache) addToHead(node *DLinkedNode) {
    node.prev = this.head
    node.next = this.head.next
    this.head.next.prev = node
    this.head.next = node
}

func (this *LRUCache) removeNode(node *DLinkedNode) {
    node.prev.next = node.next
    node.next.prev = node.prev
}

func (this *LRUCache) moveToHead(node *DLinkedNode) {
    this.removeNode(node)
    this.addToHead(node)
}

func (this *LRUCache) removeTail() *DLinkedNode {
    node := this.tail.prev
    this.removeNode(node)
    return node
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * obj := Constructor(capacity);
 * param_1 := obj.Get(key);
 * obj.Put(key,value);
 */
```

215. 数组中的第K个最大元素

1.最优解：快速选择

```
func findKthLargest(A []int, k int) int {
    return quickSelect(A, 0, len(A)-1, len(A)-k) // 第k大 == n-k 小
}

func quickSelect(A []int, low, high, index int) int {
    pos := partition(A, low, high)
    if pos == index {
        return A[index]
    } else if index < pos {
        return quickSelect(A, low, pos-1, index)
    } else {
        return quickSelect(A, pos+1, high, index)
    }
}

func partition(A []int, low, high int) int {
    A[high], A[low+(high-low)>>1] = A[low+(high-low)>>1], A[high]
    i, j := low, high
    for i < j {
        for i < j && A[i] <= A[high] {
            i++
        }
        for i < j && A[j] >= A[high] {
            j--
        }
        A[i], A[j] = A[j], A[i]
    }
    A[i], A[high] = A[high], A[i]
    return i
}
```

```
func findKthLargest(A []int, k int) int {
    return quickSelect(A, 0, len(A)-1, k-1)
}

func quickSelect(A []int, low, high, index int) int {
    j := partition(A, low, high)
    if low == high {
        return A[index]
    } else if index <= j {
        return quickSelect(A, low, j, index)
    } else {
        return quickSelect(A, j+1, high, index)
    }
}

func partition(A []int, low, high int) int {
    x := A[low+(high-low)>>1]
    i, j := low-1, high+1
    for i < j {
        for i++; A[i] > x; i++ { // 降序
        }
        for j--; A[j] < x; j-- {
        }
        if i < j {
            A[i], A[j] = A[j], A[i]
        }
    }
    return j
}
```

解法二：基于堆排序的选择方法

```
// 在大根堆中、最大元素总在根上，堆排序使用堆的这个属性进行排序
func findKthLargest(A []int, k int) int {
    heapSize, n := len(A), len(A)
    buildMaxHeap(A, heapSize)
    for i := heapSize - 1; i >= n-k+1; i-- {
        A[0], A[i] = A[i], A[0]    // 交换堆顶元素 A[0] 与堆底元素 A[i]，最大值 A[0] 放置在数组末尾
        heapSize--                 // 删除堆顶元素 A[0]
        maxHeapify(A, heapSize, 0) // 堆顶元素 A[0] 向下调整
    }
    return A[0]
}

// 建堆 O(n)
func buildMaxHeap(A []int, heapSize int) {
    for i := heapSize >> 1; i >= 0; i-- { // heap_size>>1 后面都是叶子节点，不需要向下调整
        maxHeapify(A, heapSize, i)
    }
}

// 调整大根堆 O(n)
func maxHeapify(A []int, heapSize, i int) {
    for i<<1+1 < heapSize {
        lson, rson, large := i<<1+1, i<<1+2, i
        if lson < heapSize && A[large] < A[lson] { // 左儿子存在并大于根
            large = lson
        }
        if rson < heapSize && A[large] < A[rson] { // 右儿子存在并大于根
            large = rson
        }
        if large != i { // 找到左右儿子的最大值
            A[i], A[large] = A[large], A[i] // 堆顶调整为最大值
            maxHeapify(A, heapSize, large)  // 递归调整子树
        } else {
            break
        }
    }
}

// 调整大根堆 O(nlogn)
func maxHeapify1(A []int, heapSize, i int) {
    lson, rson, large := i<<1+1, i<<1+2, i
    if lson < heapSize && A[large] < A[lson] { // 左儿子存在并大于根
        large = lson
    }
    if rson < heapSize && A[large] < A[rson] { // 右儿子存在并大于根
        large = rson
    }
    if large != i { // 找到左右儿子的最大值
        A[i], A[large] = A[large], A[i] // 堆顶调整为最大值
        maxHeapify(A, heapSize, large)  // 递归调整子树
    }
}
```

25. K 个一组翻转链表

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseKGroup(head *ListNode, k int) *ListNode {
    dummy := &ListNode{Next: head}
    prev := dummy
    for head != nil {
        tail := prev
        for i := 0; i < k; i++ {
            tail = tail.Next
            if tail == nil {
                return dummy.Next
            }
        }
        next := tail.Next         // 存储后继节点
        tail.Next = nil           // 断开链表
        prev.Next = reverse(head) // 连接前驱
        head.Next = next          // 连接后继
        prev = head               // 扫描下一组
        head = next
    }
    return dummy.Next
}

func reverse(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        next := curr.Next
        curr.Next = prev
        prev = curr
        curr = next
    }
    return prev
}
```

232. 用栈实现队列

```go
type MyQueue struct {
    inStack  []int
    outStack []int
}

func Constructor() MyQueue {
    return MyQueue{}
}
```

```
func (q *MyQueue) Push(x int) {
    q.inStack = append(q.inStack, x)
}

func (q *MyQueue) in2out() {
    for len(q.inStack) > 0 {
        q.outStack = append(q.outStack, q.inStack[len(q.inStack)-1])
        q.inStack = q.inStack[:len(q.inStack)-1]
    }
}

func (q *MyQueue) Pop() int {
    if len(q.outStack) == 0 {
        q.in2out()
    }
    val := q.outStack[len(q.outStack)-1]
    q.outStack = q.outStack[:len(q.outStack)-1]
    return val
}

func (q *MyQueue) Peek() int {
    if len(q.outStack) == 0 {
        q.in2out()
    }
    return q.outStack[len(q.outStack)-1]
}

func (q *MyQueue) Empty() bool {
    return len(q.inStack) == 0 && len(q.outStack) == 0
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * obj := Constructor();
 * obj.Push(x);
 * param_2 := obj.Pop();
 * param_3 := obj.Peek();
 * param_4 := obj.Empty();
 */
```

15. 三数之和

```
func threeSum(nums []int) [][]int {
    sort.Ints(nums)
    res := [][]int{}
    for i := 0; i < len(nums)-2; i++ {
        n1 := nums[i]
        if n1 > 0 { //如果最小的数大于0，break
            break
        }
        if i > 0 && n1 == nums[i-1] { //如果和前一个相同，跳过
            continue
        }
        start, end := i+1, len(nums)-1 //转换为两数之和，双指针解法
        for start < end {
            n2, n3 := nums[start], nums[end]
            if n1+n2+n3 == 0 {
                res = append(res, []int{n1, n2, n3})
                for start < end && nums[start] == n2 { //去重移位
                    start++
                }
                for start < end && nums[end] == n3 {
                    end--
                }
            } else if n1+n2+n3 < 0 {
                start++
            } else {
                end--
            }
        }
    }
    return res
}
```

53. 最大子数组和

```
func maxSubArray(nums []int) int {
    max, preSum := math.MinInt32, 0 // max = nums[0] OK
    for _, x := range nums {
        if preSum < 0 {
            preSum = 0
        }
        preSum += x
        if max < preSum {
            max = preSum
        }
    }
    return max
}
```

```
func maxSubArray(nums []int) int {
    pre, maxSum := 0, nums[0]
    for _, x := range nums {
        // 若当前指针所指元素之前的和小于0，则丢弃当前元素之前的数列
        pre = max(pre+x, x)
        maxSum = max(maxSum, pre) // 将当前值与最大值比较，取最大
    }
    return maxSum
}
```

```go
func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}
```

```go
func maxSubArray(nums []int) int {
    max := nums[0]
    for i := 1; i < len(nums); i++ {
        // 若前一个元素大于0，将其加到当前元素上
        if nums[i-1]+nums[i] > nums[i] { // nums[i-1] > 0
            nums[i] += nums[i-1]
        }
        if max < nums[i] {
            max = nums[i]
        }
    }
    return max
}
```

补充题4. 手撕快速排序

```go
func sortArray(nums []int) []int {
    quickSort(nums, 0, len(nums)-1)
    return nums
}

func quickSort(A []int, low, high int) {
    if low >= high {
        return
    }
    pos := partition(A, low, high)
    quickSort(A, low, pos-1)
    quickSort(A, pos+1, high)
}

func partition(A []int, low, high int) int {
    A[high], A[low+(high-low)>>1] = A[low+(high-low)>>1], A[high] // 以 A[high] 作为基准数
    i, j := low, high
    for i < j {
        for i < j && A[i] <= A[high] { // 从左向右找首个大于基准数的元素
            i++
        }
        for i < j && A[j] >= A[high] { // 从右向左找首个小于基准数的元素
            j--
        }
        A[i], A[j] = A[j], A[i] // 元素交换
    }
    A[i], A[high] = A[high], A[i] // 将基准数交换至两子数组的分界线
    return i                      // 返回基准数的索引
}
```

```
func sortArray(nums []int) []int {
    quickSort(nums, 0, len(nums)-1)
    return nums
}

func quickSort(A []int, low, high int) {
    if low >= high {
        return
    }
    j := partition(A, low, high)
    quickSort(A, low, j)
    quickSort(A, j+1, high)
}

func partition(A []int, low, high int) int {
    pivot := A[low+(high-low)>>1]
    i, j := low-1, high+1
    for i < j {
        for {
            i++
            if A[i] >= pivot {
                break
            }
        }
        for {
            j--
            if A[j] <= pivot {
                break
            }
        }
        if i < j {
            A[i], A[j] = A[j], A[i]
        }
    }
    return j
}

func Partition(A []int, low, high int) int {
    pivot := A[low+(high-low)>>1]
    i, j := low-1, high+1
    for i < j {
        for i++; A[i] < pivot; i++ {
        }
        for j--; A[j] > pivot; j-- {
        }
        if i < j {
            A[i], A[j] = A[j], A[i]
        }
    }
    return j
}
```