2023-01-27 21:48:02

3. 无重复字符的最长子串

解法一 位图

```
func lengthOfLongestSubstring(s string) int {
    if len(s) == 0 {
        return 0
    }
    var bitSet [256]bool
    result, left, right := 0, 0, 0
    for left < len(s) {
        // 右侧字符对应的 bitSet 被标记 true，说明此字符在 X 位置重复，需要左侧向前移动，直到将 X 标记为 false
        if bitSet[s[right]] {
            bitSet[s[left]] = false
            left++
        } else {
            bitSet[s[right]] = true
            right++
        }
        if result < right-left {
            result = right - left
        }
        if left+result >= len(s) || right >= len(s) {
            break
        }
    }
    return result
}
```

解法二 滑动窗口

```
func lengthOfLongestSubstring(s string) int {
    if len(s) == 0 {
        return 0
```

```
    }
    var freq [127]int
    result, left, right := 0, 0, -1
    for left < len(s) {
        if right+1 < len(s) && freq[s[right+1]] == 0 {
            freq[s[right+1]]++
            right++
        } else {
            freq[s[left]]--
            left++
        }
        if result < right-left+1 {
            result = right - left + 1
        }
    }
    return result
}
```

解法三 滑动窗口-哈希桶

```
func lengthOfLongestSubstring(s string) int {
    index := make(map[byte]int, len(s)) // 记录字符对应的下标
    result, left, right := 0, 0, 0
    for right < len(s) {
        if idx, ok := index[s[right]]; ok && idx >= left { // 遇到重复字符,
跳过
            left = idx + 1 // 收缩窗口
        }
        index[s[right]] = right //首次遇见，存储对应下标
        right++                 // 指针继续向后扫描
        result = max(result, right-left)
    }
    return result
}

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

解法四 双指针 O(n)

1. (a)bcabcbb
2. (ab)cabcbb
3. (abc)abcbb
4. (abca)bcbb 当前字符和首字符重复
5. ~~a~~(bca)bcbb 删除首字符（收缩窗口）
6. ~~a~~(bcab)cbb 继续向后扫描（扩展窗口）

7. ~~ab~~(cab)cbb

思路：定义两个指针 i,j(i<=j)，表示当前扫描到的子串是 i,j。扫描过程中维护一个哈希表 hash := map[byte]int{}，表示 [i,j]中每个字符出现的次数。线性扫描时，每次循环的流程如下：

1. 指针 j 向后移一位, 同时将哈希表中 s[j] 的计数加一: hash[s[j]]++;
2. 假设 j 移动前的区间 [i,j] 中没有重复字符，则 j 移动后，只有 s[j] 可能出现2次。因此我们不断向后移动 i，直至区间 [i,j] 中 s[j] 的个数等于1为止；

复杂度分析：由于 i,j 均最多增加n次，且哈希表的插入和更新操作的复杂度都是 O(1)，因此，总时间复杂度 O(n)

```
func lengthOfLongestSubstring(s string) int {
    hash := map[byte]int{} // 哈希集合记录每个字符出现次数
    res := 0
    for i, j := 0, 0; j < len(s); j++ {
        hash[s[j]]++                // 首次存入哈希
        for ; hash[s[j]] > 1; i++ { // 出现字符和首字符重复，i++跳过首字符(收缩窗口)
            hash[s[i]]-- // 哈希记录次数减1
        }
        if res < j-i+1 {
            res = j - i + 1 // 统计无重复字符的最长子串
        }
    }
    return res
}
```

参考

解法五 滑动窗口

思路一：

1. (**a**)bcabcbb
2. (**ab**)cabcbb
3. (**abc**)abcbb
4. (**abca**)bcbb 当前字符和首字符重复
5. ~~a~~(**bca**)bcbb 删除首字符（收缩窗口）
6. ~~a~~(**bcab**)cbb 继续向后扫描（扩展窗口）
7. ~~ab~~(**cab**)cbb

```
func lengthOfLongestSubstring(s string) int {
    m := map[byte]int{} // 哈希集合，记录每个字符出现次数
    right, res, n := -1, 0, len(s)
    for left := 0; left < n; left++ {
        if left != 0 {
            delete(m, s[left-1]) // 左指针向右移动一格，移除一个字符
        }
```

```
        for right+1 < n && m[s[right+1]] == 0 { //右指针指向字符无重复
            m[s[right+1]]++ // 不断地移动右指针
            right++
        }
        res = max(res, right-left+1) // 第 left 到 right 个字符是一个极长的无重复字符子串
    }
    return res
}
func max(x, y int) int {
    if x < y {
        return y
    }
    return x
}
```

参考官方题解

解法六 滑动窗口-哈希桶

1. 思路2：

```
func lengthOfLongestSubstring(s string) int {
    m := map[rune]int{} // 记录以当前字符为终点的无重复字符最大长度
    start := 0          // 无重复字符起始下标
    res := 0            // 目前无重复字符最大长度
    for i, v := range s {
        if _, exists := m[v]; exists { // 遇到重复字符
            start = max(start, m[v]+1) // 取index较大值作为起始下标
        }
        m[v] = i                   // 无重复字符，加入m
        res = max(res, i-start+1) // 统计目前无重复字符最大长度
    }
    return res
}
func max(x, y int) int {
    if x < y {
        return y
    }
    return x
}
```

```
func lengthOfLongestSubstring(s string) int {
    m := map[byte]int{}
    start, res := 0, 0
    for i := 0; i < len(s); i++ {
        if _, exists := m[s[i]]; exists { // 如果出现重复字符，
            start = max(start, m[s[i]]+1) // 收缩窗口
        }
```

```
            m[s[i]] = i                // 没有出现过，加入子串，扩展移动窗口
            res = max(res, i-start+1) // 统计当前最长子串
        }
        return res
    }

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

206. 反转链表

方法一：迭代

假设链表为 1→2→3→∅，我们想要把它改成 ∅←1←2←3。

在遍历链表时，将当前节点的 next 指针改为指向前一个节点。由于节点没有引用其前一个节点，因此必须事先存储其前一个节点。在更改引用之前，还需要存储后一个节点。最后返回新的头引用。

```
    /**
     * Definition for singly-linked list.
     * type ListNode struct {
     *     Val int
     *     Next *ListNode
     * }
     */
    func reverseList(head *ListNode) *ListNode {
        var prev *ListNode // prev -> nil
        curr := head
        for curr != nil { // 当前节点不为空
            next := curr.Next // 存储后续节点
            curr.Next = prev  // 反转
            prev = curr       // 迭代扫描下一对
            curr = next
        }
        return prev
    }
```

复杂度分析

- 时间复杂度：O(n)，其中 n 是链表的长度。需要遍历链表一次。
- 空间复杂度：O(1)。

方法二：递归

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseList(head *ListNode) *ListNode {
    if head == nil || head.Next == nil { // 最小子问题：无 / 只有一个节点
        return head
    }
    newHead := reverseList(head.Next) // 递：1->2->3->4->5->nil
    head.Next.Next = head             // 归：5->4   (1->2->3->  4->5->nil)
    head.Next = nil                   //    4->nil
    return newHead
}
```

方法三：头插法

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseList(head *ListNode) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }
    dummy, curr := &ListNode{Next: head}, head
    for curr.Next != nil {
        next := curr.Next
        curr.Next = next.Next  //后继结点 1——>3
        next.Next = dummy.Next //反转 2->1
        dummy.Next = next      //通知哨兵 dummy-> 2->1->3
    }
    return dummy.Next
}
```

146. LRU 缓存机制

```go
/*
 * @lc app=leetcode.cn id=146 lang=golang
 *
 * [146] LRU 缓存
 */
```

```go
// @lc code=start
type LRUCache struct {
    cache          map[int]*DLinkedNode
    head, tail     *DLinkedNode
    size, capacity int // 忘记 int ❌
}

type DLinkedNode struct { // 多写() ❌
    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
    } else {
        node := this.cache[key]
        // this.addToHead(node) 错误❌
        this.moveToHead(node)
        return node.value
    }
}

func (this *LRUCache) Put(key int, value int) {
    if _, ok := this.cache[key]; !ok {
        node := initDLinkedNode(key, value)
        this.cache[key] = node
        this.addToHead(node) // 缺少操作❌      缺少this❌
        this.size++
        if this.size > this.capacity {
            node := this.removeTail()
            delete(this.cache, node.key)
            this.size--
        }
```

```
    } else {
        node := this.cache[key]
        node.value = value // node.key = value ❌
        this.moveToHead(node)
    }
}

// 双链表操作

func (this *LRUCache) addToHead(node *DLinkedNode) {
    node.prev = this.head      // 寻找前驱节点
    node.next = this.head.next // 寻找后继节点
    node.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) // 拼写错误 removedNode ❌
    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);
 */
// @lc code=end
```

215. 数组中的第K个最大元素

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

func quick_select(A []int, start, end, index int) int {
    piv_pos := partition(A, start, end)
    if piv_pos == index {
```

```
        return A[piv_pos]
    } else if piv_pos < index {
        return quick_select(A, piv_pos+1, end, index)
    } else {
        return quick_select(A, start, piv_pos-1, index)
    }
}

func partition(A []int, start, end int) int {
    i, piv := start, A[end]
    for j := start; j < end; j++ {
        if A[j] < piv {
            if i != j {
                A[i], A[j] = A[j], A[i]
            }
            i++
        }
    }
    A[i], A[end] = A[end], A[i]
    return i
}
```

25. K 个一组翻转链表

```
/**
 * 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 next->3
        tail.Next = nil           // 分段 2->nil
        prev.Next = reverse(head) // 前驱链接反转后的链表 head->1
        head.Next = next          // head->4
        prev = head               // prev->2
        head = next               // head->4
    }
    return dummy.Next
}
```

```go
func reverse(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        next := curr.Next // 存储下一个节点
        curr.Next = prev  // 反转
        prev = curr       // 迭代
        curr = next
    }
    return prev
}
```

补充题4. 手撕快速排序 (912. 排序数组)

解法一

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

func quickSort(A []int, start, end int) {
    if start >= end {
        return
    }
    x := A[(start+end)>>1]  // x := A[(start+end)/2],用j划分递归子区间
    i, j := start-1, end+1 // 循环内直接扫描下一个数，导致多操作1次，所以预处理
    for i < j {
        for i++; A[i] < x; i++ { // 从左向右扫描，找到大于 x 的数，停止
        }
        for j--; A[j] > x; j-- { // 从右向左扫描，找到小于 x 的数，停止
        }
        if i < j {
            A[i], A[j] = A[j], A[i] // 交换，使得左边小于 x，右边大于 x
        }
    }
    quickSort(A, start, j) // 递归处理 x 左边
    quickSort(A, j+1, end) // 递归处理 x 右边
}
```

解法二

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

func quick_sort(A []int, start, end int) {
```

```
    if start < end {
        piv_pos := partition(A, start, end)
        quick_sort(A, start, piv_pos-1)
        quick_sort(A, piv_pos+1, end)
    }
}

func partition(A []int, start, end int) int {
    A[(start+end)>>1], A[end] = A[end], A[(start+end)>>1]
    i, piv := start, A[end]
    for j := start; j < end; j++ {
        if A[j] < piv {
            if i != j {
                A[i], A[j] = A[j], A[i]
            }
            i++
        }
    }
    A[i], A[end] = A[end], A[i]
    return i
}
```

```
func sortArray(nums []int) []int {
    rand.Seed(time.Now().UnixNano())
    quick_sort(nums, 0, len(nums)-1)
    return nums
}

func quick_sort(A []int, start, end int) {
    if start < end {
        piv_pos := random_partition(A, start, end)
        quick_sort(A, start, piv_pos-1)
        quick_sort(A, piv_pos+1, end)
    }
}

func partition(A []int, start, end int) int {
    i, piv := start, A[end] // 从第一个数开始扫描，选取最后一位数字最为对比
    for j := start; j < end; j++ {
        if A[j] < piv {
            if i != j {// 不是同一个数
                A[i], A[j] = A[j], A[i]// A[j] 放在正确的位置
            }
            i++//扫描下一个数
        }
    }
    A[i], A[end] = A[end], A[i] // A[end] 回到正确的位置
    return i
}

func random_partition(A []int, start, end int) int {
```

```
    random := rand.Int()%(end-start+1)+start
    A[random], A[end] = A[end],A[random]
    return partition(A, start, end)
}
```

15. 三数之和

```
func threeSum(nums []int) [][]int {
    sort.Ints(nums)
    res := make([][]int, 0)
    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 && nums[i-1] == n1 { //如果和前一个相同，重复
            continue // 跳过
        }
        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; start++ { //去重移位
                }
                for ; start < end && nums[end] == n3; end-- {
                }
            } else if n1+n2+n3 < 0 {
                start++
            } else {
                end--
            }
        }
    }
    return res
}
```

53. 最大子序和

解法一 dp

- 若前一个元素大于0，将其加到当前元素上 dp

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

```
            if max < nums[i] {
                max = nums[i]
            }
        }
        return max
    }
```

解法二 贪心

- 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
- 将当前值与最大值比较，取最大

```
func maxSubArray(nums []int) int {
    currSub, maxSub := nums[0], nums[0]
    for i := 1; i < len(nums); i++ {
        currSub = max(nums[i], currSub+nums[i]) // nums[i-1]+nums[i]
        maxSub = max(currSub, maxSub)
    }
    return maxSub
}
func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}
```

21. 合并两个有序链表

方法一：递归

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    if l1 == nil {
        return l2
    }
    if l2 == nil {
        return l1
    }
    if l1.Val < l2.Val {
        l1.Next = mergeTwoLists(l1.Next, l2)
        return l1
    } else {
```

```
        l2.Next = mergeTwoLists(l1, l2.Next)
        return l2
    }
}
```

方法二：迭代

当 l1 和 l2 都不是空链表时，判断 l1 和 l2 哪一个链表的头节点的值更小，将较小值的节点添加到结果里， 当一个节点被添加到结果里之后，将对应链表中的节点向后移一位。

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    dummy := new(ListNode)
    prev := dummy
    for l1 != nil && l2 != nil {
        if l1.Val < l2.Val {
            prev.Next = l1
            l1 = l1.Next
        } else {
            prev.Next = l2
            l2 = l2.Next
        }
        prev = prev.Next
    }
    if l1 == nil {
        prev.Next = l2
    } else {
        prev.Next = l1
    }
    return dummy.Next
}
```

1. 两数之和

```
func twoSum(nums []int, target int) []int {
    hash := map[int]int{}
    for i, v := range nums {
        if j, ok := hash[target-v]; ok {
            return []int{i, j}
        }
        hash[v] = i
    }
```

```
        return nil
    }
```