```java
// 翻转链表
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode cur = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return cur;
    }
}
```

```java
// 合并两个升序的链表
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // 合并两个升序的链表 1.合并k个 2.链表的归并排序
        if(list1 == null){
            return list2;
        }
        if(list2 == null){
            return list1;
        }
        ListNode l1 = list1;
        ListNode l2 = list2;
        ListNode dummy = new ListNode(-1);
        ListNode pre = dummy;
        while(l1!=null && l2!=null){
            //较小的节点加入
            if(l1.val<l2.val){
                pre.next = l1;
                l1 = l1.next;
            }else{
                pre.next = l2;
                l2 = l2.next;
            }
                pre = pre.next;
        }
        if(l1 == null){
            pre.next = l2;
        }
        if(l2 == null){
            pre.next = l1;
        }
        return dummy.next;
    }
}
```

```
// 合并k个升序链表
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
            //设置递归出口
            if(lists == null || lists.length == 0){
                return lists;
            }
            if(lists.length == 1){
                return lists[0];
            }
            if(lists.length == 2){
                return mergeTwo(lists[0],lists[1]);
            }

            int mid = lists.length/2;
            // 0 - mid-1  mid - length-1
            ListNode[] leftList = new ListNode[mid];
            ListNode[] rightList = new ListNode[lists.length - mid];
            for(int i = 0; i<mid; i++){
                leftList[i] = lists[i];
            }
            for(int i = mid; i<lists.length; i++){
                rightList[i - mid] = lists[i];
            }
            ListNode left = mergeKLists(leftList);
            ListNode right = mergeKLists(rightList);
            return mergeTwo(left,right);
    }

    public ListNode mergeTwo(ListNode list1 , ListNode list2){
            if(list1 == null){
                return list2;
            }
            if(list2 == null){
                return list1;
            }
            ListNode dummy = new ListNode(-1);
            ListNode pre = dummy;
            while(list1!=null && list2!=null){
                if(list1.val<list2.val){
                    pre.next = list1;
                    list1 = list1.next;
                }else{
                    pre.next = list2;
                    list2 = list2.next;
                }
                pre = pre.next;
            }
            if(list1 == null){
                pre.next = list2;
            }
            if(list2 == null){
                pre.next = list1;
            }
            return dummy.next;
    }


}

```

```
// 链表归并排序
class Solution {
    public ListNode sortList(ListNode head) {
        // 归并排序-递归出口
        if(head == null || head.next == null){
            return head;
        }
        //找到中间的节点，不断的分割在合并
        ListNode mid = getMidNode(head);
        ListNode rightStart = mid.next;
        mid.next = null;
        ListNode left = sortList(head);
        ListNode right = sortList(rightStart);
        return mergeTwo(left,right);
    }

    // 快慢指针找到链表的中间节点-偏左
    // 让fast少走一次就行了
    public ListNode getMidNode(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode slow = head;
        ListNode fast = head.next.next;
        while(fast!=null&&fast.next!=null){
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }

    // 合并两个有序节点-模板
    public ListNode mergeTwo(ListNode l1,ListNode l2){
        if(l1 == null){
            return l2;
        }
        if(l2 == null){
            return l1;
        }
        ListNode dummy = new ListNode(-1);
        ListNode pre = dummy;
        while(l1!=null&&l2!=null){
            if(l1.val<l2.val){
                pre.next = l1;
                l1 = l1.next;
            }else{
                pre.next = l2;
                l2 = l2.next;
            }
            pre = pre.next;
        }
        pre.next = l1 == null?l2:l1;
        return dummy.next;
    }
}
```

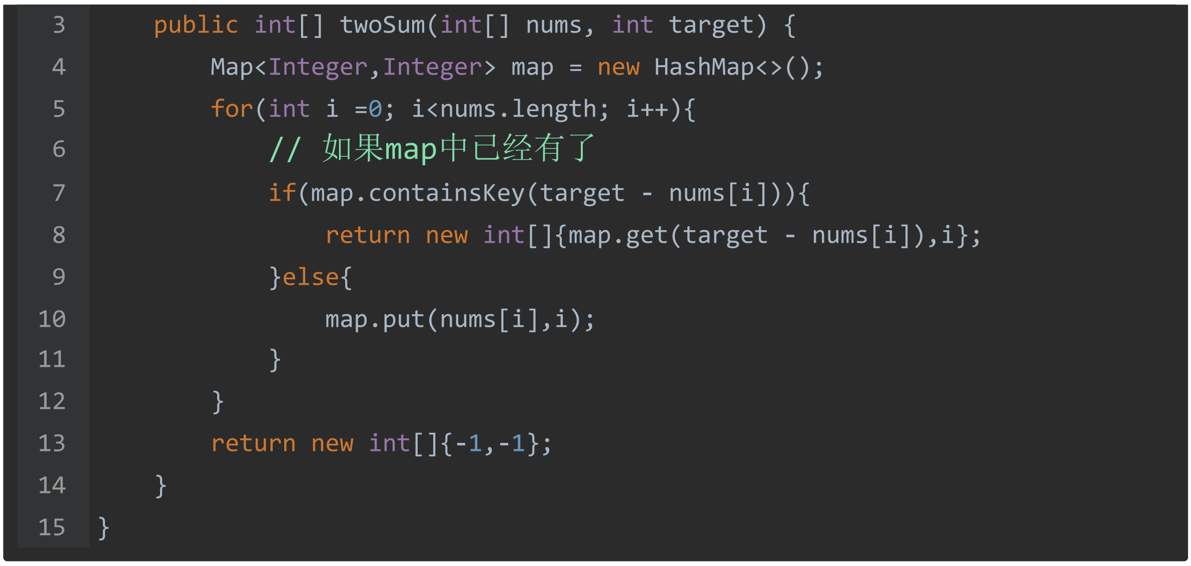
```
// 无重复字符的最长子串
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if(s == null || s.length() == 0){
            return 0;
        }
        char[] str = s.toCharArray();
        boolean[] has = new boolean[128];
        int left = 0;
        int res = 0;
        for(int right = 0; right<str.length;right++){
            char cur = str[right];
            // 如果右节点已经重复，移动左节点
            while(has[cur]){
                has[str[left++]] = false;
            }
            //当前节点加入has
            has[cur] = true;
            res = Math.max(res,right-left+1);
        }
        return res;
    }
}
```

```
// 返回第k大的元素
```

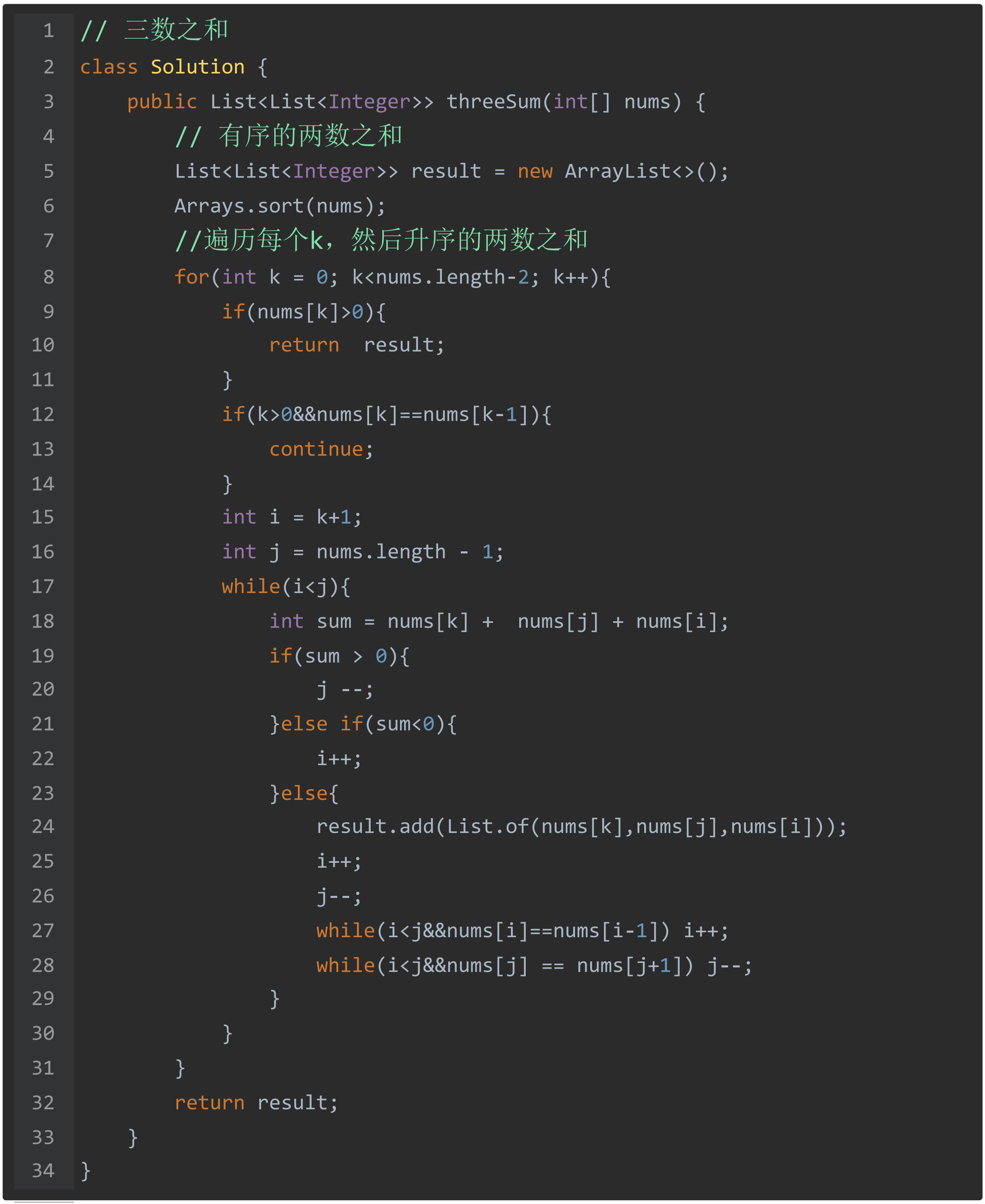
```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        // 维护k大小的小根堆
        PriorityQueue<Integer> minHeap =  new PriorityQueue<>();
        for(int num : nums){
            if(minHeap.size()<k){
                minHeap.add(num);
            }else if(minHeap.peek()<num){
                minHeap.poll();
                minHeap.add(num);

            }
        }
        return minHeap.poll();

    }
}
```

```java
// 返回前k个元素
class Solution {
    public int[] findKthSmallestElements(int[] nums, int k) {
        // 大根堆
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        for (int num : nums) {
            if(maxHeap.size()<k){
                maxHeap.add(num);
            }else if(maxHeap.peek()>num){
                maxHeap.poll();
                maxHeap.add(num);
            }
        }
        // 提取最小堆中的前 k 个元素（由于是最小堆，直接按升序提取）
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = maxHeap.poll(); // 堆顶元素每次提取都会是最小的
        }

        return result;
    }
}
```

```java
// 两数之和
class Solution {
```



```java
    public int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> map = new HashMap<>();
        for(int i =0; i<nums.length; i++){
            // 如果map中已经有了
            if(map.containsKey(target - nums[i])){
                return new int[]{map.get(target - nums[i]),i};
            }else{
                map.put(nums[i],i);
            }
        }
        return new int[]{-1,-1};
    }
}
```

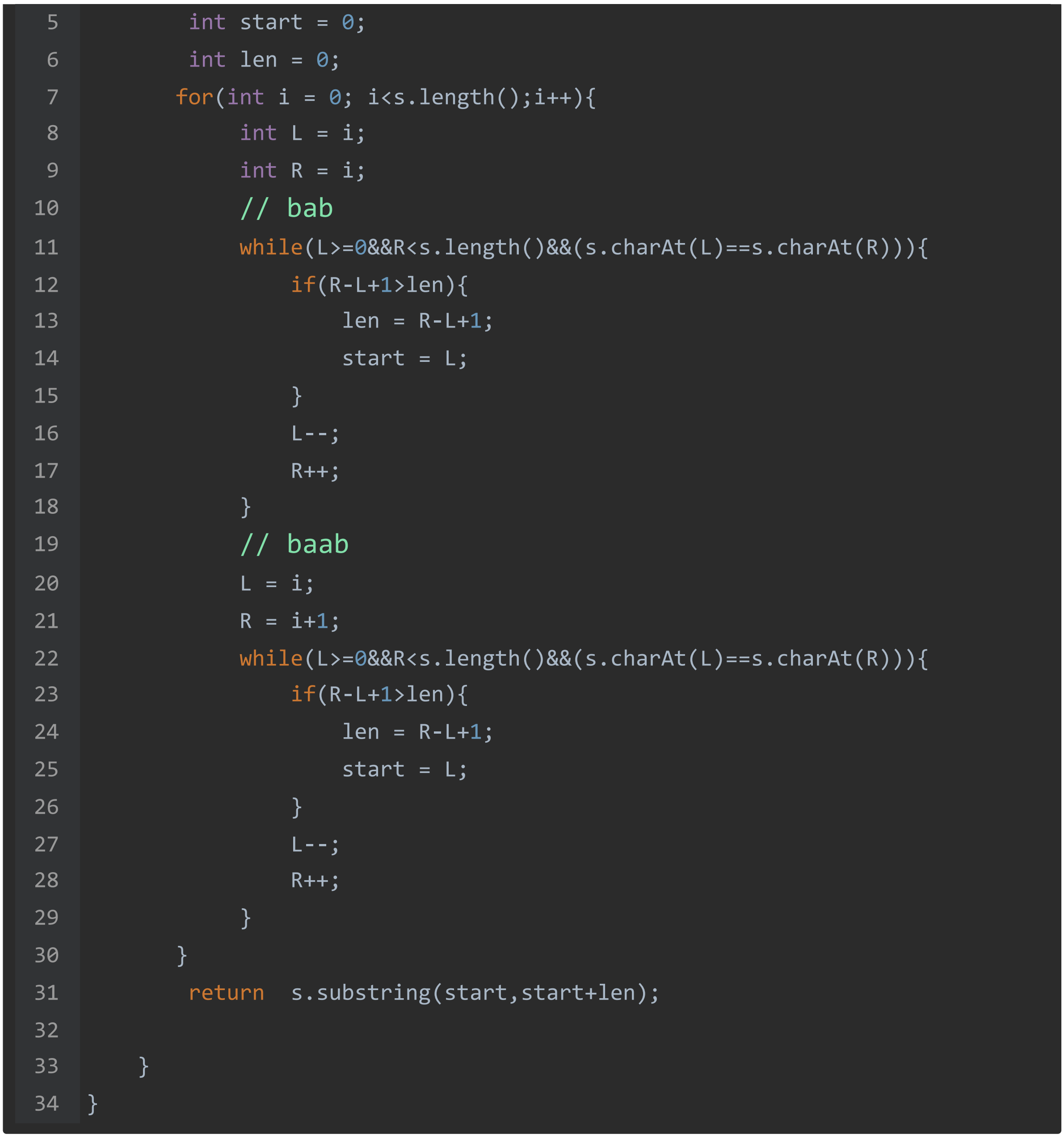


```java
// 三数之和
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        // 有序的两数之和
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);
        //遍历每个k，然后升序的两数之和
        for(int k = 0; k<nums.length-2; k++){
            if(nums[k]>0){
                return  result;
            }
            if(k>0&&nums[k]==nums[k-1]){
                continue;
            }
            int i = k+1;
            int j = nums.length - 1;
            while(i<j){
                int sum = nums[k] +  nums[j] + nums[i];
                if(sum > 0){
                    j --;
                }else if(sum<0){
                    i++;
                }else{
                    result.add(List.of(nums[k],nums[j],nums[i]));
                    i++;
                    j--;
                    while(i<j&&nums[i]==nums[i-1]) i++;
                    while(i<j&&nums[j] == nums[j+1]) j--;
                }
            }
        }
        return result;
    }
}
```

```java
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        // 同向双指针
        int n = numbers.length;
        int left = 0;
        int right = n -1;
        while(true){
            int sum = numbers[left] + numbers[right];
            if(sum == target){
                return new int[]{left+1,right+1};
            }
            if(sum>target){
                right--;
            }else{
                left++;
            }
        }
    }
}
```

```java
// 最大子数组的和
class Solution {
    public int maxSubArray(int[] nums) {
        // 使用dp去做
        if(nums == null || nums.length == 0){
            return 0;
        }
        if(nums.length == 1){
            return nums[0];
        }
        int[] dp = new int[nums.length+1];
        dp[0] = nums[0];
        //dp[i]以nums[i]结尾的最大子数组的值
        int res = nums[0];
        for(int i = 1; i<nums.length; i++){
            dp[i] = Math.max(nums[i],dp[i-1]+nums[i]);
            res = Math.max(res,dp[i]);
        }
        return res;
    }
}
```

```java
// 最长回文子串
class Solution {
    public String longestPalindrome(String s) {

```



```
        int start = 0;
        int len = 0;
       for(int i = 0; i<s.length();i++){
            int L = i;
            int R = i;
            // bab
            while(L>=0&&R<s.length()&&(s.charAt(L)==s.charAt(R))){
                if(R-L+1>len){
                    len = R-L+1;
                    start = L;
                }
                L--;
                R++;
            }
            // baab
            L = i;
            R = i+1;
            while(L>=0&&R<s.length()&&(s.charAt(L)==s.charAt(R))){
                if(R-L+1>len){
                    len = R-L+1;
                    start = L;
                }
                L--;
                R++;
            }
       }
        return  s.substring(start,start+len);

    }
}
```