数组

189、旋转数组

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

进阶：

尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
你可以使用空间复杂度为 O(1) 的 原地 算法解决这个问题吗？

示例 1:

```
输入: nums = [1,2,3,4,5,6,7], k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右旋转 1 步: [7,1,2,3,4,5,6]
向右旋转 2 步: [6,7,1,2,3,4,5]
向右旋转 3 步: [5,6,7,1,2,3,4]
```

解题思路:

操作	结果
原始数组	1 2 3 4 5 6 7
翻转所有元素	7 6 5 4 3 2 1
翻转 [0,k mod n−1] 区间的元素	5 6 7 4 3 2 1
翻转 [k mod n−1，n−1] 区间的元素	5 6 7 1 2 3 4

题解代码:

```java
class Solution {
    public void rotate(int[] nums, int k) {
        k %= nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }

    public void reverse(int[] nums, int start, int end) {
        while (start < end) {
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start += 1;
            end -= 1;
        }
    }
}
```

457、环形数组是否存在循环

存在一个不含 0 的 环形 数组 nums ，每个 nums[i] 都表示位于下标 i 的角色应该向前或向后移动的下标个数：

- 如果 nums[i] 是正数，向前 移动 nums[i] 步
- 如果 nums[i] 是负数，向后 移动 nums[i] 步

因为数组是 环形 的，所以可以假设从最后一个元素向前移动一步会到达第一个元素，而第一个元素向后移动一步会到达最后一个元素。

数组中的 循环 由长度为 k 的下标序列 seq ：

- 遵循上述移动规则将导致重复下标序列 seq[0] -> seq[1] -> ... -> seq[k - 1] -> seq[0] -> ...
- 所有 nums[seq[j]] 应当不是 全正 就是 全负
- k > 1

如果 nums 中存在循环，返回 true ； 否则，返回 false 。

示例 1：

```
输入：nums = [2,-1,1,2,2]
输出：true
解释：存在循环，按下标 0 -> 2 -> 3 -> 0 。循环长度为 3 。
```

示例 2：

```
输入：nums = [-1,2]
输出：false
解释：按下标 1 -> 1 -> 1 ... 的运动无法构成循环，因为循环的长度为 1 。根据定义，循环的长度必须大于 1 。
```

解题思路：

DFS

做深度优先搜素，利用字典visited来标记已经搜索过的节点。
利用numSet记录在一个方向上遇到的节点，如果新节点在numSet就有环。但需要在三个情况清空numSet：

- 从i节点开始DFS到底了，从i+1节点开始搜素时清空numSet.
- 当搜索方向direction改变符号，按题意要求同方向，清空numSet.
- 当一个节点的下一个节点是自身，清空numSet.

快慢指针

题解代码：

```python
#暴力法
class Solution:
    def circularArrayLoop(self, nums: List[int]) -> bool:
        n = len(nums)
        visited = set()
        for i in range(n):
            if i not in visited:
                direction = nums[i]
                visited.add(i)
                numSet = set()
                numSet.add(i)
                j = (i + nums[i]) % n
                while j not in visited:
                    visited.add(j)
                    if nums[j] * direction < 0: #方向改变，numSet清空
                        direction = nums[j]
```

```
                        numSet = set()
                        numSet.add(j)
                        j = (n + j + nums[j]) % n
                    else:
                        numSet.add(j)
                        j = (n + j + nums[j]) % n
                    if (n + j + nums[j]) % n == j: #下一步回到当前位置，numSet清空
                        numSet = set()
                    elif j in numSet and len(numSet) >= 2:
                        return True
        return False
```

```
//快慢指针
class Solution {
    public boolean circularArrayLoop(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            if (nums[i] == 0) {
                continue;
            }
            int slow = i, fast = next(nums, i);
            // 判断非零且方向相同
            while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[next(nums,
fast)] > 0) {
                if (slow == fast) {
                    if (slow != next(nums, slow)) {
                        return true;
                    } else {
                        break;
                    }
                }
                slow = next(nums, slow);
                fast = next(nums, next(nums, fast));
            }
            int add = i;
            while (nums[add] * nums[next(nums, add)] > 0) {
                int tmp = add;
                add = next(nums, add);
                nums[tmp] = 0;
            }
        }
        return false;
    }

    public int next(int[] nums, int cur) {
        int n = nums.length;
        return ((cur + nums[cur]) % n + n) % n; // 保证返回值在 [0,n) 中
    }
}
```

918、环形子数组的最大和

给定一个由整数数组 A 表示的环形数组 C，求 C 的非空子数组的最大可能和。

在此处，环形数组意味着数组的末端将会与开头相连呈环状。（形式上，当0 <= i < A.length 时 C[i] = A[i]，且当 i >= 0 时 C[i+A.length] = C[i]）

此外，子数组最多只能包含固定缓冲区 A 中的每个元素一次。（形式上，对于子数组 C[i], C[i+1], ..., C[j]，不存在 i <= k1, k2 <= j 其中 k1 % A.length = k2 % A.length）

示例 1：

```
输入：[1,-2,3,-2]
输出：3
解释：从子数组 [3] 得到最大和 3
```

解题思路：

- 环形子数组的最大和具有两种可能，一种是不使用环的情况，另一种是使用环的情况
- 不使用环的情况时，直接通过53题的思路，逐步求出整个数组中的最大子序和即可
- 使用到了环，则必定包含 A[n-1]和 A[0]两个元素且说明从A[1]到A[n-2]这个子数组中必定包含负数
- 【否则只通过一趟最大子序和就可以的出结果】
- 因此只需要把A[1]-A[n-2]间这些负数的最小和求出来
- 用整个数组的和 sum减掉这个负数最小和即可实现原环型数组的最大和

题解代码：

```
class Solution {
    public int maxSubarraySumCircular(int[] A) {
        int[] dp = new int[A.length];   //dp[i]用来记录以nums[i]结尾的最大子序列和
        dp[0] = A[0];                   //初始化dp
        int max = dp[0];                //最大子序列和
        int sum = dp[0];                //整个数组的和

        //求最大子序列和，见53题
        for (int i = 1; i < dp.length; i++) {
            sum += A[i];
            dp[i] = A[i] + Math.max(dp[i - 1], 0);
            max = Math.max(dp[i], max);
        }

        int min = 0;    //开始求A[1]~A[n-1]上的最小子序列和
        for (int i = 1; i < dp.length - 1; i++) {
            dp[i] = A[i] + Math.min(0, dp[i - 1]);
            min = Math.min(dp[i], min);
        }
        return Math.max(sum - min, max);
    }
}
```

ps：对于环形数组，可以考虑有环和无环的情况

881、救生艇

第 i 个人的体重为 people[i]，每艘船可以承载的最大重量为 limit。

每艘船最多可同时载两人，但条件是这些人的重量之和最多为 limit。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1：

```
输入：people = [1,2], limit = 3
输出：1
解释：1 艘船载 (1, 2)
```

示例 2：

```
输入：people = [3,2,2,1], limit = 3
输出：3
解释：3 艘船分别载 (1, 2), (2) 和 (3)
```

解题思路：

排序 + 双指针

题解代码：

```java
class Solution {
    public int numRescueBoats(int[] people, int limit) {
        int n = people.length;
        Arrays.sort(people);
        int l = 0, h = n-1;
        int ans = 0;
        while(l <= h){
            if(people[l] + people[h] <= limit){
                ++l;
            }
            --h;
            ++ans;
        }
        return ans;
    }
}
```

剑指offer 03、数组中重复的数字

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

```
输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3
```

解题思路：

考虑数组中数字的范围[0,n-1]，所以有以下两种方式

- 哈希表
- 原地置换，使得 $nums[i] == i$

题解代码：

```java
class Solution {
    public int findRepeatNumber(int[] nums) {
        int n = nums.length;
        int i = 0;
        while(i < n){
            if(nums[i] == i){
                ++i;
                continue;
            }
            if(nums[nums[i]] == nums[i]) return nums[i];
            int tmp = nums[i];
            nums[i] = nums[tmp];
            nums[tmp] = tmp;
        }
        return -1;
    }
```

```
}
```

链表

206、反转链表

给你单链表的头节点 `head` ，请你反转链表，并返回反转后的链表。

示例 1：

```
输入：head = [1,2,3,4,5]
输出：[5,4,3,2,1]
```

解题思路：

迭代：头插法

递归:

递归版本稍微复杂一些，其关键在于反向工作。假设链表的其余部分已经被反转，现在应该如何反转它前面的部分？

假设链表为：

$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \rightarrow \ldots \rightarrow n_m \rightarrow \varnothing n1 \rightarrow \ldots \rightarrow nk-1 \rightarrow nk \rightarrow nk+1 \rightarrow \ldots \rightarrow nm \rightarrow \emptyset$

若从节点 n_{k+1}到 n_m已经被反转，而我们正处于 n_k。

$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \leftarrow \ldots \leftarrow n_m n1 \rightarrow \ldots \rightarrow nk-1 \rightarrow nk \rightarrow nk+1 \leftarrow \ldots \leftarrow nm$

我们希望 n_{k+1} 的下一个节点指向 n_k 。

所以，$n_k.next.next = n_k$ 。

需要注意的是 n_1 的下一个节点必须指向 $\varnothing$。如果忽略了这一点，链表中可能会产生环。

题解代码：

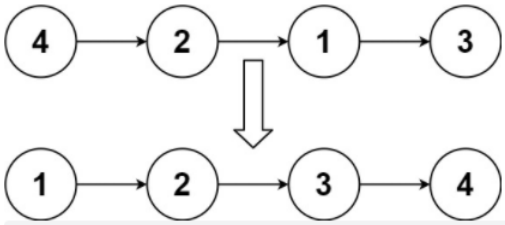
```java
class Solution {  //头插法
    public ListNode reverseList(ListNode head) {
        ListNode pre = null;
        ListNode tmp = head;
        while(tmp != null){
            ListNode next = tmp.next;
            tmp.next = pre;
            pre = tmp;
            tmp = next;
        }
        return pre;
    }
}
```

```java
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
}
```

148、排序链表

给你链表的头结点head，请将其按升序排列并返回排序后的链表。

示例 1：



输入： head = [4,2,1,3]
输出： [1,2,3,4]

解题思路：

归并排序，快慢指针找中点

题解代码：

```java
class Solution {
    public ListNode sortList(ListNode head) {
        return sortList(head, null);
    }

    public ListNode sortList(ListNode head, ListNode tail) {
        if (head == null) {
            return head;
        }
        if (head.next == tail) {
            head.next = null;
            return head;
        }
        ListNode slow = head, fast = head;
        while (fast != tail) {
            slow = slow.next;
            fast = fast.next;
            if (fast != tail) {
                fast = fast.next;
            }
        }
        ListNode mid = slow;
        ListNode list1 = sortList(head, mid);
        ListNode list2 = sortList(mid, tail);
        ListNode sorted = merge(list1, list2);
        return sorted;
    }
```

```
28
29      public ListNode merge(ListNode head1, ListNode head2) {
30          ListNode dummyHead = new ListNode(0);
31          ListNode temp = dummyHead, temp1 = head1, temp2 = head2;
32          while (temp1 != null && temp2 != null) {
33              if (temp1.val <= temp2.val) {
34                  temp.next = temp1;
35                  temp1 = temp1.next;
36              } else {
37                  temp.next = temp2;
38                  temp2 = temp2.next;
39              }
40              temp = temp.next;
41          }
42          if (temp1 != null) {
43              temp.next = temp1;
44          } else if (temp2 != null) {
45              temp.next = temp2;
46          }
47          return dummyHead.next;
48      }
49  }
```

栈和队列

232、栈实现队列

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（`push`、`pop`、`peek`、`empty`）

```
1   //弹出时考虑先先将1中数据转移到2中
2   class MyQueue {
3
4       Stack<Integer> p;
5       Stack<Integer> q;
6
7       /** Initialize your data structure here. */
8       public MyQueue() {
9           p = new Stack<>();
10          q = new Stack<>();
11      }
12
13      /** Push element x to the back of queue. */
14      public void push(int x) {
15          p.push(x);
16      }
17
18      /** Removes the element from in front of queue and returns that element. */
19      public int pop() {
20          if(q.isEmpty()){
21              while(!p.isEmpty()){
22                  q.push(p.pop());
23              }
24          }
25          return q.pop();
26      }
27
28      /** Get the front element. */
29      public int peek() {
30          if(q.isEmpty()){
```

```
            while(!p.isEmpty()){
                q.push(p.pop());
            }
        }
        return q.peek();
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return p.isEmpty() && q.isEmpty();
    }
}
```

225、用队列实现栈

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（`push`、`top`、`pop` 和 `empty`）。

```
//添加时考虑将队列1中数据转入2中，然后两者互换
class MyStack {
    Queue<Integer> queue1;
    Queue<Integer> queue2;

    /** Initialize your data structure here. */
    public MyStack() {
        queue1 = new LinkedList<Integer>();
        queue2 = new LinkedList<Integer>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        queue2.offer(x);
        while (!queue1.isEmpty()) {
            queue2.offer(queue1.poll());
        }
        Queue<Integer> temp = queue1;
        queue1 = queue2;
        queue2 = temp;
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        return queue1.poll();
    }

    /** Get the top element. */
    public int top() {
        return queue1.peek();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
        return queue1.isEmpty();
    }
}
```

150、逆波兰表达式求值

根据 逆波兰表示法，求表达式的值。

有效的算符包括 +、-、*、/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

整数除法只保留整数部分。
给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
输入：tokens = ["2","1","+","3","*"]
输出：9
解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9
```

题解代码：

```
class Solution {
    public int evalRPN(String[] tokens) {
        Deque<Integer> stack = new LinkedList<Integer>();
        int n = tokens.length;
        for (int i = 0; i < n; i++) {
            String token = tokens[i];
            if (isNumber(token)) {
                stack.push(Integer.parseInt(token));
            } else {
                int num2 = stack.pop();
                int num1 = stack.pop();
                switch (token) {
                    case "+":
                        stack.push(num1 + num2);
                        break;
                    case "-":
                        stack.push(num1 - num2);
                        break;
                    case "*":
                        stack.push(num1 * num2);
                        break;
                    case "/":
                        stack.push(num1 / num2);
                        break;
                    default:
                }
            }
        }
        return stack.pop();
    }

    public boolean isNumber(String token) {
        return !("+".equals(token) || "-".equals(token) || "*".equals(token) ||
"/".equals(token));
    }
}
```

ps：中缀式转后缀式也采用栈，对于操作符赋予不同的优先级

堆

堆排序

```java
/**
 * 堆排序
 * 有效数据从0开始，
 * 所以一个节点i，其对应二叉树左右子节点下标分别为2*i+1以及2*i+2
 */
public class MaxHeapSort {

    @Test
    public void test(){
        int[] array= {2,8,14,4,16,7,1,10,9,3};
        heapSort(array);
        //输出堆排序结果
        for(int i:array){
            println(i);
        }
    }

    /**
     * 堆排序
     * @param array
     */
    public void heapSort(int[] array){
        //初始化大顶堆
        buildMaxHeap(array);
        //堆排序
        int heapSize = array.length;
        //最外层是循环次数，循环到最后大顶堆只有一个元素时停止，所以循环次数为array.length-1
        for(int i=0;i<array.length-1;i++){
            //交换a[0]与大顶堆最后一个元素(不包括已排好序的节点)
            swap(array,0,heapSize-1);
            //大顶堆数据减少一个
            heapSize--;
            //我这里array[0]也是有效数据，所以maxHeepify的第二个参数一致是0
            maxHeepify(array,0,heapSize);
        }
    }

    /**
     * 初始化大顶堆
     */
    private void buildMaxHeap(int[] array){
        int len = array.length;
        for(int i= (array.length-2)/2;i>=0;i--){
            maxHeepify(array,i,len);
        }
    }
    /**
     *
     * @param arr
     * @param i
     */
    private void maxHeepify(int[] arr,int i,int len){
        //println("i="+i);
        //有效数据下标从0开始
        //左子节点
        int left = 2*i+1;
```

```
        //右子节点
        int right = 2*i+2;
        //初始化最大值节点为当前节点
        int largest = i;
        //左节点不超出数组范围且比较大节点值大，则更新较大值下标
        if(left <len && arr[left] > arr[largest]){
            //左节点比该节点大
            largest = left;
        }
        //右节点不超出数组范围且比较大节点值大，则更新较大值下标
        if(right <len && arr[right] > arr[largest]){
            //左节点比该节点大
            largest = right;
        }
        //如果子节点有一个比当前节点大，则进行数据呼唤，同时向下递归
        if(largest != i){
            //交换节点i与较大子节点数据
            swap(arr,i,largest);
            //经过上面的调整后节点i与其两个子节点满足大顶堆条件
            //但是需要判断调整后的节点largest位置以及其子节点是否还满足大顶堆特性
            maxHeepify(arr,largest,len);
        }
    }

    private void swap(int[] arr,int i,int j){
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

502、IPO

假设 力扣（LeetCode）即将开始 IPO 。为了以更高的价格将股票卖给风险投资公司，力扣 希望在 IPO 之前开展一些项目以增加其资本。 由于资源有限，它只能在 IPO 之前完成最多 k 个不同的项目。帮助 力扣 设计完成最多 k 个不同项目后得到最大总资本的方式。

给你 n 个项目。对于每个项目 i ，它都有一个纯利润 profits[i] ，和启动该项目需要的最小资本 capital[i] 。

最初，你的资本为 w 。当你完成一个项目时，你将获得纯利润，且利润将被添加到你的总资本中。

总而言之，从给定项目中选择 最多 k 个不同项目的列表，以 最大化最终资本 ，并输出最终可获得的最多资本。

答案保证在 32 位有符号整数范围内。

示例 1：

```
输入：k = 2, w = 0, profits = [1,2,3], capital = [0,1,1]
输出：4
解释：
由于你的初始资本为 0，你仅可以从 0 号项目开始。
在完成后，你将获得 1 的利润，你的总资本将变为 1。
此时你可以选择开始 1 号或 2 号项目。
由于你最多可以选择两个项目，所以你需要完成 2 号项目以获得最大的资本。
因此，输出最后最大化的资本，为 0 + 1 + 3 = 4。
```

解题思路：

贪心 + 堆

思路与算法

我们首先思考，如果不限制次数下我们可以获取的最大利润，我们应该如何处理？我们只需将所有的项目按照资本的大小进行排序，依次购入项目 i，同时手中持有的资本增加 $profits[i]$，直到手中的持有的资本无法启动当前的项目为止。

- 如果初始资本 $w \geq \max(capital)$，我们直接返回利润中的 k 个最大元素的和即可。
- 当前的题目中限定了可以选择的次数最多为 k 次，这就意味着我们应该贪心地保证选择每次投资的项目获取的利润最大，这样就能保持选择 k 次后获取的利润最大。
- 我们首先将项目按照所需资本的从小到大进行排序，每次进行选择时，假设当前手中持有的资本为 w，我们应该从所有投入资本小于等于 w 的项目中选择利润最大的项目 j，然后此时我们更新手中持有的资本为 $w + profits[j]$，同时再从所有花费资本小于等于 $w + profits[j]$ 的项目中选择，我们按照上述规则不断选择 k 次即可。
- 我们利用大根堆的特性，我们将所有能够投资的项目的利润全部压入到堆中，每次从堆中取出最大值，然后更新手中持有的资本，同时将待选的项目利润进入堆，不断重复上述操作。
- 如果当前的堆为空，则此时我们应当直接返回。

题解代码：

```java
class Solution {
    public int findMaximizedCapital(int k, int w, int[] profits, int[] capital) {
        int n = profits.length;
        int curr = 0;
        int[][] arr = new int[n][2];

        for (int i = 0; i < n; ++i) {
            arr[i][0] = capital[i];
            arr[i][1] = profits[i];
        }
        Arrays.sort(arr, (a, b) -> a[0] - b[0]);

        PriorityQueue<Integer> pq = new PriorityQueue<>((x, y) -> y - x);
        for (int i = 0; i < k; ++i) {
            while (curr < n && arr[curr][0] <= w) {
                pq.add(arr[curr][1]);
                curr++;
            }
            if (!pq.isEmpty()) {
                w += pq.poll();
            } else {
                break;
            }
        }

        return w;
    }
}
```

二叉树

参考链接：二叉树所有遍历模板及知识点总结

前、中、后序迭代遍历模板

```python
# 迭代：前、中、后序遍历通用模板（只需一个栈的空间）
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        res = []
        stack = []
```

```
        cur = root
        # 中序，模板：先用指针找到每颗子树的最左下角，然后进行进出栈操作
        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            res.append(cur.val)
            cur = cur.right
        return res

        # # 前序，相同模板
        # while stack or cur:
        #     while cur:
        #         res.append(cur.val)
        #         stack.append(cur)
        #         cur = cur.left
        #     cur = stack.pop()
        #     cur = cur.right
        # return res

        # # 后序，相同模板
        # while stack or cur:
        #     while cur:
        #         res.append(cur.val)
        #         stack.append(cur)
        #         cur = cur.right
        #     cur = stack.pop()
        #     cur = cur.left
        # return res[::-1] #反转链表
```

二叉搜索树和中序遍历

二叉搜索树的中序遍历为递增序列

98、验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含大于当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入:
    2
   / \
  1   3
输出: true
```

解题思路：

中序遍历，满足序列保持递增的性质，分为递归和迭代的方法

题解代码：

```
//递归
class Solution {
```

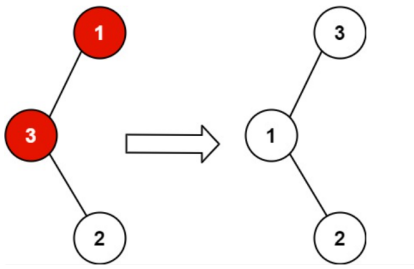
```
    TreeNode pre = null;
    public boolean isValidBST(TreeNode root) {
        if(root == null){
            return true;
        }
        boolean left = isValidBST(root.left);
        if(pre != null && pre.val >= root.val){
            return false;
        }else{
            pre = root;
        }
        boolean right = isValidBST(root.right);
        return left && right;
    }
}

//迭代
class Solution {
    public boolean isValidBST(TreeNode root) {
        Deque<TreeNode> stack = new LinkedList<TreeNode>();
        double inorder = -Double.MAX_VALUE;

        while (!stack.isEmpty() || root != null) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
              // 如果中序遍历得到的节点的值小于等于前一个 inorder，说明不是二叉搜索树
            if (root.val <= inorder) {
                return false;
            }
            inorder = root.val;
            root = root.right;
        }
        return true;
    }
}
```

99、恢复二叉搜索树

给你二叉搜索树的根节点 root ，该树中的两个节点被错误地交换。请在不改变其结构的情况下，恢复这棵树。

进阶：使用 O(n) 空间复杂度的解法很容易实现。你能想出一个只使用常数空间的解决方案吗？

示例1




```
输入：root = [1,3,null,null,2]
输出：[3,1,null,null,2]
解释：3 不能是 1 左孩子，因为 3 > 1 。交换 1 和 3 使二叉搜索树有效。
```

解题思路：

方法一是显式地将中序遍历的值序列保存在一个 nums 数组中，然后再去寻找被错误交换的节点，但我们也可以隐式地在中序遍历的过程就找到被错误交换的节点 x 和 y。

具体来说，由于我们只关心中序遍历的值序列中每个相邻的位置的大小关系是否满足条件，且错误交换后最多两个位置不满足条件，因此在中序遍历的过程我们只需要维护当前中序遍历到的最后一个节点 pred，然后在遍历到下一个节点的时候，看两个节点的值是否满足前者小于后者即可，如果不满足说明找到了一个交换的节点，且在找到两次以后就可以终止遍历。

这样我们就可以在中序遍历中直接找到被错误交换的两个节点 x 和 y，不用显式建立 nums 数组。

中序遍历的实现有迭代和递归两种等价的写法，在本方法中提供迭代实现的写法。使用迭代实现中序遍历需要手动维护栈。

题解代码：

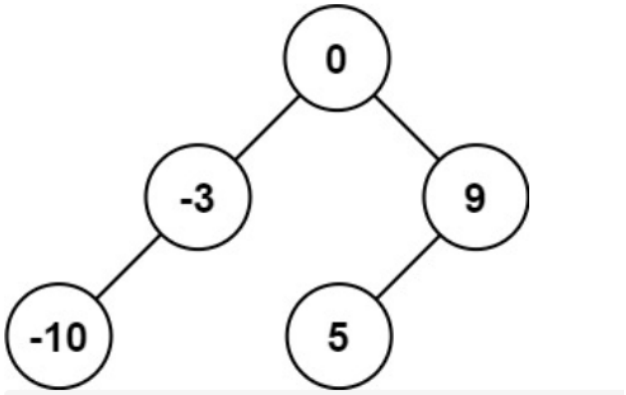
```java
class Solution {
    public void recoverTree(TreeNode root) {
        Deque<TreeNode> stack = new ArrayDeque<TreeNode>();
        TreeNode x = null, y = null, pred = null;

        while (!stack.isEmpty() || root != null) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            if (pred != null && root.val < pred.val) {
                y = root;
                if (x == null) {
                    x = pred;
                } else {
                    break;
                }
            }
            pred = root;
            root = root.right;
        }

        swap(x, y);
    }

    public void swap(TreeNode x, TreeNode y) {
        int tmp = x.val;
        x.val = y.val;
        y.val = tmp;
    }
}
```

108、将有序数组转换为二叉搜索树

给你一个整数数组 nums ，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

示例 1：




```
输入：nums = [-10,-3,0,5,9]
输出：[0,-3,9,-10,null,5]
解释：[0,-10,5,null,-3,null,9] 也将被视为正确答案：
```

解题思路：

给定二叉搜索树的中序遍历，是否可以唯一地确定二叉搜索树？答案是否定的。如果没有要求二叉搜索树的高度平衡，则任何一个数字都可以作为二叉搜索树的根节点，因此可能的二叉搜索树有多个。

如果增加一个限制条件，即要求二叉搜索树的高度平衡，是否可以唯一地确定二叉搜索树？答案仍然是否定的。

直观地看，我们可以选择中间数字作为二叉搜索树的根节点，这样分给左右子树的数字个数相同或只相差 11，可以使得树保持平衡。如果数组长度是奇数，则根节点的选择是唯一的，如果数组长度是偶数，则可以选择中间位置左边的数字作为根节点或者选择中间位置右边的数字作为根节点，选择不同的数字作为根节点则创建的平衡二叉搜索树也是不同的。

题目109、有序链表转换为二叉树 利用快慢指针找到链表的中点

题解代码：

```java
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }

    public TreeNode helper(int[] nums, int left, int right) {
        if (left > right) {
            return null;
        }

        // 总是选择中间位置左边的数字作为根节点
        int mid = (left + right) / 2;

        TreeNode root = new TreeNode(nums[mid]);
        root.left = helper(nums, left, mid - 1);
        root.right = helper(nums, mid + 1, right);
        return root;
    }
}
```

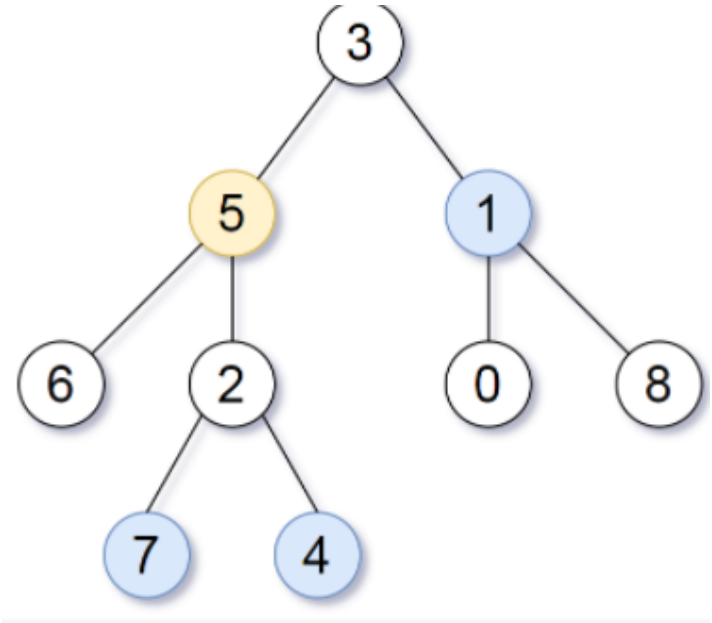
863、二叉树中所有距离为K的结点

给定一个二叉树（具有根结点 root），一个目标结点 target，和一个整数值 K 。

返回到目标结点 target 距离为 K 的所有结点的值的列表。 答案可以以任何顺序返回。

示例 1：

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2
输出：[7,4,1]
解释：
所求结点为与目标结点（值为 5）距离为 2 的结点，
值分别为 7，4，以及 1
```



解题思路：

深度优先搜索 + 哈希表

若将 target 当作树的根结点，我们就能从 target 出发，使用深度优先搜索去寻找与 target 距离为 k 的所有结点，即深度为 k 的所有结点。

由于输入的二叉树没有记录父结点，为此，我们从根结点 root 出发，使用深度优先搜索遍历整棵树，同时用一个哈希表记录每个结点的父结点。

然后从 target 出发，使用深度优先搜索遍历整棵树，除了搜索左右儿子外，还可以顺着父结点向上搜索。

题解代码：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    Map<Integer,TreeNode> map = new HashMap<>();
    List<Integer> ans = new ArrayList<>();
    public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {
        findParent(root);
        findAns(target,null,0,k);
        return ans;
    }
    public void findParent(TreeNode node){
        if(node.left != null){
            map.put(node.left.val,node);
            findParent(node.left);
        }
        if(node.right != null){
```

```
            map.put(node.right.val,node);
            findParent(node.right);
        }
    }
    public void findAns(TreeNode node, TreeNode pre, int depth, int k){
        if(node == null){
            return;
        }
        if(depth == k){
            ans.add(node.val);
            return;
        }
        if(node.left != pre){
            findAns(node.left,node,depth+1,k);
        }
        if(node.right != pre){
            findAns(node.right,node,depth+1,k);
        }
        if(map.get(node.val) != pre){
            findAns(map.get(node.val),node,depth+1,k);
        }
    }
}
```

102、二叉树的层序遍历

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

示例：
二叉树：[3,9,20,null,null,15,7],

返回其层序遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

解题思路：

BFS 和 层序遍历的差别

```
//BFS
void bfs(TreeNode root) {
    Queue<TreeNode> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll(); // Java 的 pop 写作 poll()
        if (node.left != null) {
            queue.add(node.left);
        }
        if (node.right != null) {
            queue.add(node.right);
        }
    }
}
```

BFS输出的为一维数组，层序遍历为各层的数组

题解代码：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if(root == null){
            return res;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while(!queue.isEmpty()){
            int n = queue.size();
            List<Integer> level = new ArrayList<>();
            for(int i = 0; i < n; ++i){
                TreeNode node = queue.poll();
                level.add(node.val);
                if(node.left != null){
                    queue.offer(node.left);
                }
                if(node.right != null){
                    queue.offer(node.right);
                }
            }
            res.add(level);
        }
        return res;
    }
}
```

987、二叉树的垂序遍历

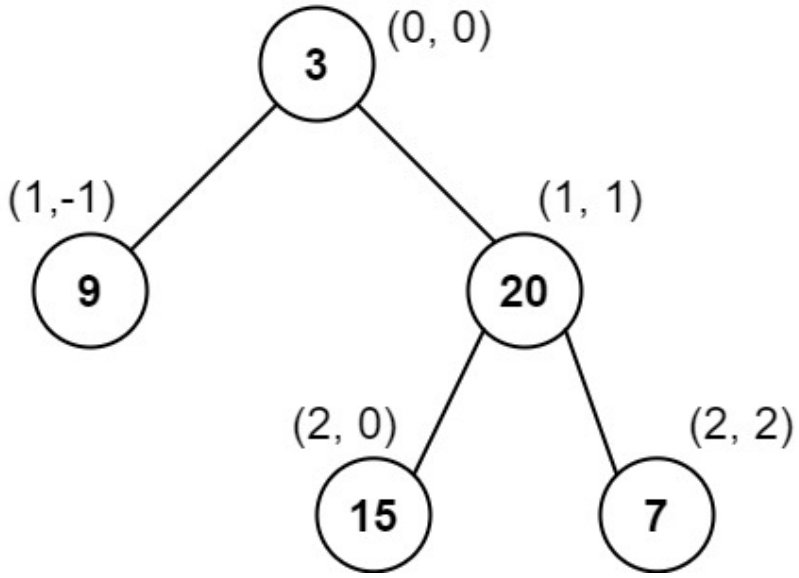
给你二叉树的根结点 root ，请你设计算法计算二叉树的 垂序遍历 序列。

对位于 (row, col) 的每个结点而言，其左右子结点分别位于 (row + 1, col - 1) 和 (row + 1, col + 1) 。树的根结点位于 (0, 0) 。

二叉树的 垂序遍历 从最左边的列开始直到最右边的列结束，按列索引每一列上的所有结点，形成一个按出现位置从上到下排序的有序列表。如果同行同列上有多个结点，则按结点的值从小到大进行排序。

返回二叉树的 垂序遍历 序列。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：[[9],[3,15],[20],[7]]
解释：
列 -1 ：只有结点 9 在此列中。
列  0 ：只有结点 3 和 15 在此列中，按从上到下顺序。
列  1 ：只有结点 20 在此列中。
列  2 ：只有结点 7 在此列中。
```

解题思路：

我们可以从根节点开始，对整棵树进行一次遍历，在遍历的过程中使用数组 nodes 记录下每个节点的行号 row，列号 col 以及值 value。在遍历完成后，我们按照 col 为第一关键字升序，row 为第二关键字升序，value 为第三关键字升序，对所有的节点进行排序即可。

在排序完成后，我们还需要按照题目要求，将同一列的所有节点放入同一个数组中。因此，我们可以对 nodes 进行一次遍历，并在遍历的过程中记录上一个节点的列号 lastcol。如果当前遍历到的节点的列号 col 与 lastcol 相等，则将该节点放入与上一个节点相同的数组中，否则放入不同的数组中。

题解代码：

```java
class Solution {
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        List<int[]> nodes = new ArrayList<int[]>();
        dfs(root, 0, 0, nodes);
        Collections.sort(nodes, new Comparator<int[]>() {
            public int compare(int[] tuple1, int[] tuple2) {
                if (tuple1[0] != tuple2[0]) {
                    return tuple1[0] - tuple2[0];
                } else if (tuple1[1] != tuple2[1]) {
                    return tuple1[1] - tuple2[1];
                } else {
                    return tuple1[2] - tuple2[2];
                }
            }
        });
        List<List<Integer>> ans = new ArrayList<List<Integer>>();
        int size = 0;
        int lastcol = Integer.MIN_VALUE;
        for (int[] tuple : nodes) {
            int col = tuple[0], row = tuple[1], value = tuple[2];
            if (col != lastcol) {
                lastcol = col;
                ans.add(new ArrayList<Integer>());
```

```
                size++;
            }
            ans.get(size - 1).add(value);
        }
        return ans;
    }

    public void dfs(TreeNode node, int row, int col, List<int[]> nodes) {
        if (node == null) {
            return;
        }
        nodes.add(new int[]{col, row, node.val});
        dfs(node.left, row + 1, col - 1, nodes);
        dfs(node.right, row + 1, col + 1, nodes);
    }
}
```

105. 从前序与中序遍历序列构造二叉树（map结合）

给定一棵树的前序遍历 `preorder` 与中序遍历 `inorder`。请构造二叉树并返回其根节点。

示例 1:

```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

解题思路:

递归解法，使用哈希表快速定位节点位置

迭代解法

题解代码:

```
class Solution {
    private Map<Integer, Integer> indexMap;

    public TreeNode myBuildTree(int[] preorder, int[] inorder, int preorder_left, int preorder_right, int inorder_left, int inorder_right) {
        if (preorder_left > preorder_right) {
            return null;
        }

        // 前序遍历中的第一个节点就是根节点
        int preorder_root = preorder_left;
        // 在中序遍历中定位根节点
        int inorder_root = indexMap.get(preorder[preorder_root]);

        // 先把根节点建立出来
        TreeNode root = new TreeNode(preorder[preorder_root]);
        // 得到左子树中的节点数目
        int size_left_subtree = inorder_root - inorder_left;
        // 递归地构造左子树，并连接到根节点
        // 先序遍历中「从 左边界+1 开始的 size_left_subtree」个元素就对应了中序遍历中「从 左边界 开始到 根节点定位-1」的元素
        root.left = myBuildTree(preorder, inorder, preorder_left + 1, preorder_left + size_left_subtree, inorder_left, inorder_root - 1);
        // 递归地构造右子树，并连接到根节点
        // 先序遍历中「从 左边界+1+左子树节点数目 开始到 右边界」的元素就对应了中序遍历中「从 根节点定位+1 到 右边界」的元素
```

```
        root.right = myBuildTree(preorder, inorder, preorder_left +
size_left_subtree + 1, preorder_right, inorder_root + 1, inorder_right);
        return root;
    }

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int n = preorder.length;
        // 构造哈希映射，帮助我们快速定位根节点
        indexMap = new HashMap<Integer, Integer>();
        for (int i = 0; i < n; i++) {
            indexMap.put(inorder[i], i);
        }
        return myBuildTree(preorder, inorder, 0, n - 1, 0, n - 1);
    }
}
```

前缀和

523、连续的子数组和

给你一个整数数组 nums 和一个整数 k ，编写一个函数来判断该数组是否含有同时满足下述条件的连续子数组：

- 子数组大小 至少为 2 ，且
- 子数组元素总和为 k 的倍数。

如果存在，返回 true ；否则，返回 false 。

如果存在一个整数 n ，令整数 x 符合 x = n * k ，则称 x 是 k 的一个倍数。

示例 1：

```
输入：nums = [23,2,4,6,7], k = 6
输出：true
解释：[2,4] 是一个大小为 2 的子数组，并且和为 6 。
```

示例 2：

```
输入：nums = [23,2,6,4,7], k = 6
输出：true
解释：[23, 2, 6, 4, 7] 是大小为 5 的子数组，并且和为 42 。
42 是 6 的倍数，因为 42 = 7 * 6 且 7 是一个整数。
```

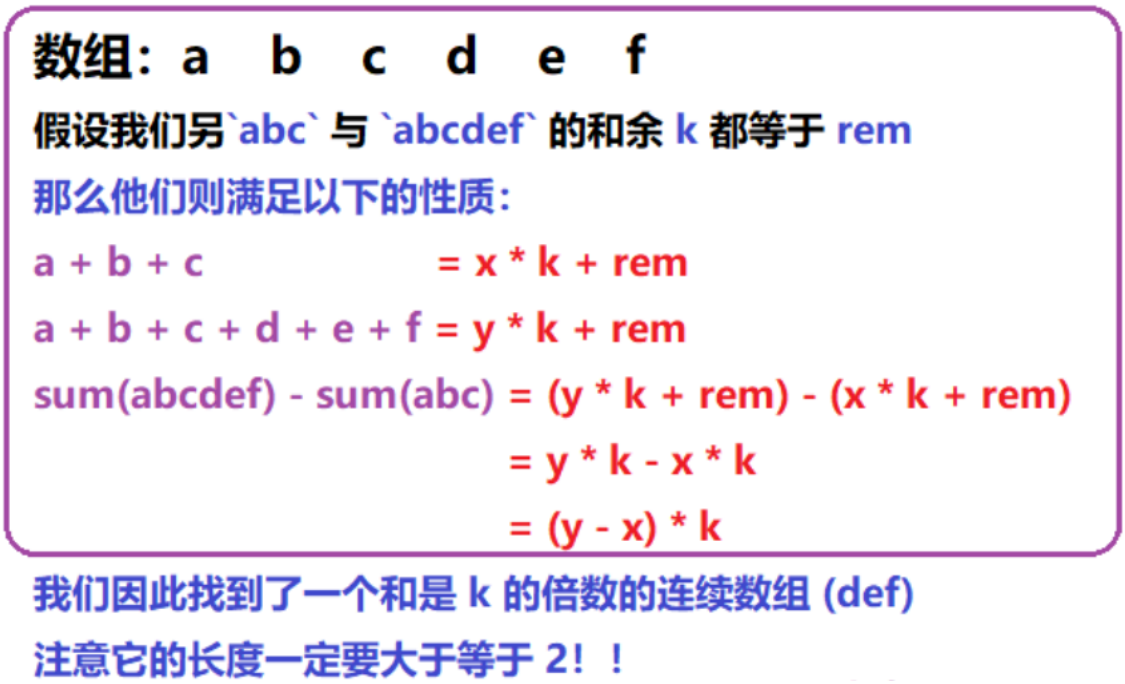
解题思路：

(1) 直接前缀和，时间复杂度为O(n^2)

(2) 前缀和+哈希表

- 创建一个哈希表，key来储存当前前缀和的余数，value则储存对应的index
- 如果哈希表中存在其对应的余数，我们则取出其pos，看当前的下标 index 到 pos的距离是否大于2.（题目要求）如果是则返回true。不是我们则继续遍历。不要更新哈希表中的下标！(贪心的思维)
- 如果不存在则将当前余数与其对应的下标储存在哈希表中。

你问我答：为什么找到了相同的余数就相当于找到了一个连续的前缀和是 k*k* 的倍数?



题解代码:

```java
class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        int m = nums.length;
        if (m < 2) {
            return false;
        }
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
        map.put(0, -1);
        int remainder = 0;
        for (int i = 0; i < m; i++) {
            remainder = (remainder + nums[i]) % k;
            if (map.containsKey(remainder)) {
                int prevIndex = map.get(remainder);
                if (i - prevIndex >= 2) {
                    return true;
                }
            } else {
                map.put(remainder, i);
            }
        }
        return false;
    }
}
```

525、连续数组

给定一个二进制数组 nums , 找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。

示例 1:

```
输入: nums = [0,1]
输出: 2
说明: [0, 1] 是具有相同数量0和1的最长连续子数组。
```

示例 2:

```
输入: nums = [0,1,0]
输出: 2
说明: [0, 1] (或 [1, 0]) 是具有相同数量0和1的最长连续子数组。
```

解题思路:

前缀和 + 哈希表，利用counter变量记录1和0的数目，如果为1，则加1，如果为0，则减1。遍历过程寻找map中与当前counter值相同的key，相减求取最长连续数组。

题解代码:

```java
class Solution {
    public int findMaxLength(int[] nums) {
        int n = nums.length;
        Map<Integer,Integer> map = new HashMap<>();
        int maxLen = 0;
        int counter = 0;
        map.put(counter,-1);
        for(int i = 0; i < n; ++i){
            int num = nums[i];
            if(num == 1){
                counter ++;
            }else{
                counter --;
            }
            if(map.containsKey(counter)){
                int pre = map.get(counter);
                maxLen = Math.max(maxLen,i - pre);
            }else{
                map.put(counter,i);
            }
        }
        return maxLen;
    }
}
```

1893、检查是否区域内所有整数都被覆盖

给你一个二维整数数组 ranges 和两个整数 left 和 right 。每个 ranges[i] = [starti, endi] 表示一个从 starti 到 endi 的 闭区间 。

如果闭区间 [left, right] 内每个整数都被 ranges 中 至少一个 区间覆盖，那么请你返回 true ，否则返回 false 。

已知区间 ranges[i] = [starti, endi] ，如果整数 x 满足 starti <= x <= endi ，那么我们称整数x 被覆盖了。

示例 1：

```
输入: ranges = [[1,2],[3,4],[5,6]], left = 2, right = 5
输出: true
解释: 2 到 5 的每个整数都被覆盖了:

- 2 被第一个区间覆盖。
- 3 和 4 被第二个区间覆盖。
- 5 被第三个区间覆盖。
```

解题思路:

- 排序
- 差分数组，前缀和思想

差分数组diff表示相邻格之间，是否被覆盖的变化量。
diff[i]++,代表在i位置上有新的覆盖
若覆盖到j结束了呢？此时j依然是覆盖，但是j+1不在覆盖状态，所以在j+1处 -1；
即diff[j+1]--;
当我们把差分数组求前缀和，就很直观把这种变化量转化为不变的，可以理解的。

题解代码：

```
//排序
class Solution {
    public boolean isCovered(int[][] ranges, int left, int right) {
        Arrays.sort(ranges,(a1,a2) -> a1[0] - a2[0]);
        for(int []range : ranges){
            if(range[0] <= left && range[1] >= left){
                left = range[1] + 1;
            }
        }
        return left > right;
    }
}
//差分数组
class Solution {
    public boolean isCovered(int[][] ranges, int left, int right) {
        int[] diff = new int[52];
        //对差分数组进行处理
        for(int i = 0; i < ranges.length; i++){
            diff[ranges[i][0]]++;
            diff[ranges[i][1]+1]--;
        }
        //根据差分数组处理前缀和，为理解方便单独定义sum，可以原地做
        int[] sum = new int[52];
        for(int i = 1; i <= 51; i++){
            sum[i] = sum[i-1] + diff[i];
        }
        //从left到right判断是否满足sum > 0
        for(int i = left; i <= right; i++){
            if(sum[i] <= 0) return false;
        }
        return true;
    }
}
```

1588、所有奇数长度子数组的和

给你一个正整数数组 arr ，请你计算所有可能的奇数长度子数组的和。

子数组 定义为原数组中的一个连续子序列。

请你返回 arr 中 所有奇数长度子数组的和 。

示例 1：

```
输入：arr = [1,4,2,5,3]
输出：58
解释：所有奇数长度子数组和它们的和为：
[1] = 1
[4] = 4
[2] = 2
[5] = 5
[3] = 3
[1,4,2] = 7
[4,2,5] = 11
[2,5,3] = 10
[1,4,2,5,3] = 15
我们将所有值求和得到 1 + 4 + 2 + 5 + 3 + 7 + 11 + 10 + 15 = 58
```

解题思路：

前缀和，再统计奇数子数组的和

题解代码：

```java
class Solution {
    public int sumOddLengthSubarrays(int[] arr) {
        int n = arr.length;
        int []sum = new int[n + 1];
        sum[0] = 0;
        for(int i = 1; i <= n; ++i){
            sum[i] = sum[i - 1] + arr[i - 1];
        }
        int res = 0;
        for(int i = 0; i < n; ++i){
            for(int j = 1; i + j - 1 < n; j += 2){
                res += sum[i+j] - sum[i];
            }
        }
        return res;
    }
}
```

差分数组

1109、航班预定统计

这里有 n 个航班，它们分别从 1 到 n 进行编号。

有一份航班预订表 bookings ，表中第 i 条预订记录 bookings[i] = [firsti, lasti, seatsi] 意味着在从 firsti 到 lasti （包含 firsti 和 lasti ）的 每个航班 上预订了 seatsi 个座位。

请你返回一个长度为 n 的数组 answer，其中 answer[i] 是航班 i 上预订的座位总数。

示例 1：

```
输入：bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
输出：[10,55,45,25,25]
解释：
航班编号        1   2   3   4   5
预订记录 1 ：   10  10
预订记录 2 ：       20  20
预订记录 3 ：       25  25  25  25
总座位数：      10  55  45  25  25
因此，answer = [10,55,45,25,25]
```

解题思路：

差分数组 + 前缀和

差分数组对应的概念是前缀和数组，对于数组 $[1,2,2,4]$，其差分数组为 $[1,1,0,2]$，差分数组的第 i 个数即为原数组的第 i-1 个元素和第 i 个元素的差值，也就是说我们对差分数组求前缀和即可得到原数组。

差分数组的性质是，当我们希望对原数组的某一个区间 $[l,r]$ 施加一个增量inc 时，差分数组 d 对应的改变是：$d[l]$ 增加 inc，$d[r+1]$ 减少 inc。这样对于区间的修改就变为了对于两个位置的修改。并且这种修改是可以叠加的，即当我们多次对原数组的不同区间施加不同的增量，我们只要按规则修改差分数组即可。

在本题中，我们可以遍历给定的预定记录数组，每次 O(1)地完成对差分数组的修改即可。当我们完成了差分数组的修改，只需要最后求出差分数组的前缀和即可得到目标数组。

解题思路：

1. 换一种思路理解题意，将问题转换为：某公交车共有 n 站，第 `i` 条记录 `bookings[i] = [i, j, k]` 表示在 `i` 站上车 `k` 人，乘坐到 `j` 站，在 `j+1` 站下车，需要按照车站顺序返回每一站车上的人数
2. 根据 1 的思路，定义 `counter[]` 数组记录每站的人数变化，`counter[i]` 表示第 `i+1` 站。遍历 `bookings[]`：`bookings[i] = [i, j, k]` 表示在 `i` 站增加 `k` 人即 `counters[i-1] += k`，在 `j+1` 站减少 `k` 人即 `counters[j] -= k`
3. 遍历（整理）`counter[]` 数组，得到每站总人数：每站的人数为前一站人数加上当前人数变化 `counters[i] += counters[i - 1]`

题解代码：

```java
class Solution {
    public int[] corpFlightBookings(int[][] bookings, int n) {
        int []res = new int[n];
        for(int []booking : bookings){
            res[booking[0] - 1] += booking[2];
            if(booking[1] < n){
                res[booking[1]] -= booking[2];
            }
        }
        for(int i = 1; i < n; ++i){
            res[i] += res[i-1];
        }
        return res;
    }
}
```

哈希表

哈希表与树结合，可以存储父节点（二叉树中所有距离为K的节点）

17.11、大餐计数

大餐 是指 恰好包含两道不同餐品 的一餐，其美味程度之和等于 2 的幂。

你可以搭配 任意 两道餐品做一顿大餐。

给你一个整数数组 deliciousness ，其中 deliciousness[i] 是第 i 道餐品的美味程度，返回你可以用数组中的餐品做出的不同 大餐 的数量。结果需要对 109 + 7 取余。

注意，只要餐品下标不同，就可以认为是不同的餐品，即便它们的美味程度相同。

示例 1：

```
输入：deliciousness = [1,3,5,7,9]
输出：4
解释：大餐的美味程度组合为 (1,3) 、(1,7) 、(3,5) 和 (7,9) 。
它们各自的美味程度之和分别为 4 、8 、8 和 16 ，都是 2 的幂。
```

解题思路：

两层遍历时间复杂度为O(n^2)

使用哈希表

题解代码：

```
class Solution {
    public int countPairs(int[] deliciousness) {
       final int mod = 1000000007;
       int maxVal = -1;
       for(int val : deliciousness){
           maxVal = Math.max(maxVal,val);
       }
       int sum = maxVal * 2;
       int pairs = 0;
       Map<Integer,Integer> map = new HashMap<>();
       for(int i  = 0; i < deliciousness.length; ++i){
           int del = deliciousness[i];
           for(int j = 1; j <= sum; j <<= 1){
               int count = map.getOrDefault(j - del, 0);
               pairs = (pairs + count) % mod;
           }
           map.put(del,map.getOrDefault(del,0) + 1);
       }
       return pairs;
    }
}
```

摩尔投票

17.10、主要元素

数组中占比超过一半的元素称之为主要元素。给你一个 整数 数组，找出其中的主要元素。若没有，返回 -1 。请设计时间复杂度为 O(N) 、空间复杂度为 O(1) 的解决方案。

示例 1：

```
输入：[1,2,5,9,5,9,5,5,5]
输出：5
```

示例 2：

```
输入：[3,2]
输出：-1
```

解题思路：

一次遍历，cur为上一步的数，count表示cur的数量，当前数如果和cur相同则count++，否则count--。当count为0时，替换cur。遍历结束后，再遍历一次，统计最终cur的数量是否大于数组长度的一半。

题解代码：

```

```

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        length = len(nums)
        count = 1
        cur = nums[0]
        for i in range(1,length):
            if count == 0:
                cur = nums[i]
                count = 1
                continue
            if nums[i] != cur:
                count -= 1
            else:
                count += 1
        acc = 0
        for num in nums:
            if num == cur:
                acc += 1
        return cur if acc * 2 >length else -1
```

排序

快速排序

```java
public class QuickSort {
    public static void main(String[] args) {
        int[] arr = {5,3,8,1,-1};
        Quick_Sort(arr,0,arr.length-1);
        for(int num : arr){
            System.out.print(num + " ");
        }
    }

    public static void Quick_Sort(int[] a, int low, int high){
        if(low > high){
            return;
        }
        int l,r,temp;
        l = low;
        r = high;
        temp = a[l];

        //循环的作用，保证基准值左边都小于，右边都大于
        while(l < r){
            while(l < r && temp <= a[r]){
                --r;
            }
            a[l] = a[r];
            while(l < r && temp >= a[l]){
                ++l;
            }
            a[r] = a[l];
        }
        a[l] = temp;
        Quick_Sort(a, low, l-1);
        Quick_Sort(a, l+1, high);
    }
}
```

归并排序

```java
public class MergeSort {
    public static void main(String[] args) {
        int[] arr = {11,44,23,67,88,65,34,48,9,12};
        int[] tmp = new int[arr.length];    //新建一个临时数组存放
        mergeSort(arr,0,arr.length-1,tmp);
        for(int i=0;i<arr.length;i++){
            System.out.print(arr[i]+" ");
        }
    }

    public static void merge(int[] arr,int low,int mid,int high,int[] tmp){
        int i = 0;
        int j = low,k = mid+1;  //左边序列和右边序列起始索引
        while(j <= mid && k <= high){
            if(arr[j] < arr[k]){
                tmp[i++] = arr[j++];
            }else{
                tmp[i++] = arr[k++];
            }
        }
        //若左边序列还有剩余，则将其全部拷贝进tmp[]中
        while(j <= mid){
            tmp[i++] = arr[j++];
        }

        while(k <= high){
            tmp[i++] = arr[k++];
        }

        for(int t=0;t<i;t++){
            arr[low+t] = tmp[t];
        }
    }

    public static void mergeSort(int[] arr,int low,int high,int[] tmp){
        if(low<high){
            int mid = (low+high)/2;
            mergeSort(arr,low,mid,tmp); //对左边序列进行归并排序
            mergeSort(arr,mid+1,high,tmp);  //对右边序列进行归并排序
            merge(arr,low,mid,high,tmp);    //合并两个有序序列
        }
    }

}
```

75、颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

示例 1：

```
输入: nums = [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]
```

示例 2：

```
输入：nums = [2,0,1]
输出：[0,1,2]
```

解题思路：

- 双指针，交换
- 统计0，1，2的数量，填数

题解代码：

```
class Solution {
    public void sortColors(int[] nums) {
        int n = nums.length;
        int p0 = 0, p1 = 0;
        for (int i = 0; i < n; ++i) {
            if (nums[i] == 1) {
                int temp = nums[i];
                nums[i] = nums[p1];
                nums[p1] = temp;
                ++p1;
            } else if (nums[i] == 0) {
                int temp = nums[i];
                nums[i] = nums[p0];
                nums[p0] = temp;
                if (p0 < p1) {
                    temp = nums[i];
                    nums[i] = nums[p1];
                    nums[p1] = temp;
                }
                ++p0;
                ++p1;
            }
        }
    }
}
```

滑动窗口

1838、最高频元素的频数

元素的 频数 是该元素在一个数组中出现的次数。

给你一个整数数组 nums 和一个整数 k 。在一步操作中，你可以选择 nums 的一个下标，并将该下标对应元素的值增加 1 。

执行最多 k 次操作后，返回数组中最高频元素的 最大可能频数 。

示例 1：

```
输入：nums = [1,2,4], k = 5
输出：3
解释：对第一个元素执行 3 次递增操作，对第二个元素执 2 次递增操作，此时 nums = [4,4,4] 。
4 是数组中最高频元素，频数是 3 。
```

解题思路：

排序 + 滑动窗口

题解代码：

```

```

```java
class Solution {
    public int maxFrequency(int[] nums, int k) {
        Arrays.sort(nums);
        int tmp = 0;
        int max = 1;
        int start = 0;
        for(int end = 0; end < nums.length; ++end){
            tmp += nums[end];
            if((end-start+1)*nums[end] - tmp <= k){
                max = Math.max(max,end-start+1);
            }else{
                tmp -= nums[start];
                start ++;
            }
        }
        return max;
    }
}
```

567、字符串的排列

给你两个字符串 s1 和 s2 ，写一个函数来判断 s2 是否包含 s1 的排列。

换句话说，s1 的排列之一是 s2 的 子串 。

示例 1：

```
输入: s1 = "ab" s2 = "eidbaooo"
输出: true
解释: s2 包含 s1 的排列之一 ("ba").
```

示例 2：

```
输入: s1= "ab" s2 = "eidboaoo"
输出: false
```

解题思路：

滑动窗口，数组统计s1的字符情况，具体见代码。

题解代码：

```java
class Solution {
    public boolean checkInclusion(String s1, String s2) {
        int n = s1.length(), m = s2.length();
        if(n > m){
            return false;
        }
        int []cnt = new int[26];
        for(int i = 0; i < n; ++i){
            --cnt[s1.charAt(i) - 'a'];
        }
        int left = 0, right = 0;
        while(right < m){
            int idx = s2.charAt(right) - 'a';
            ++cnt[idx];
            while(cnt[idx] > 0){
                --cnt[s2.charAt(left) - 'a'];
                ++left;
            }
```

```
            if(right - left + 1 == n){
                return true;
            }
            ++right;
        }
        return false;
    }
}
```

二分查找

```
    //普通二分查找 left <= right
    public static int binary_search1(int[] a,int n,int target){
        int left = 0, right = n-1;

        while(left <= right){
            int mid = (left + right) / 2;
            if(a[mid] == target){
                return mid;
            }
            if(a[mid] > target){
                right = mid - 1;
            }
            if(a[mid] < target){
                left = mid + 1;
            }
        }
        return -1;
    }

    //普通二分查找 left < right
    public static int binary_search2(int[] a,int n,int target){
        int left = 0, right = n-1;

        while(left < right){
            int mid = (left + right) / 2;
            if(a[mid] == target){
                return mid;
            }
            if(a[mid] > target){
                right = mid;
            }
            if(a[mid] < target){
                left = mid + 1;
            }
        }
        return -1;
    }

    //寻找左边界
    public static int left_binary_search(int[] a,int n,int target){
        int left = 0, right = n-1;
        int ans = 0;
        while(left <= right){
            int mid = (left + right) / 2;
            if(a[mid] >= target){
                right = mid - 1;
                ans = mid;
            }else{
                left = mid + 1;
```

```
            }
        }
        return ans;
    }

    //寻找右边界
    public static int right_binary_search(int[] a,int n,int target){
        int left = 0, right = n-1;
        int ans = 0;
        while(left <= right){
            int mid = (left + right) / 2;
            if(a[mid] > target){
                right = mid - 1;
                ans = mid;
            }else{
                left = mid + 1;
            }
        }
        return ans;
    }
```

374、猜数字大小

猜数字游戏的规则如下：

每轮游戏，我都会从 1 到 n 随机选择一个数字。 请你猜选出的是哪个数字。
如果你猜错了，我会告诉你，你猜测的数字比我选出的数字是大了还是小了。
你可以通过调用一个预先定义好的接口 int guess(int num) 来获取猜测结果，返回值一共有 3 种可能的情况（-1，1 或 0）：

-1：我选出的数字比你猜的数字小 pick < num
1：我选出的数字比你猜的数字大 pick > num
0：我选出的数字和你猜的数字一样。恭喜！你猜对了！pick == num
返回我选出的数字。

示例 1：

```
输入：n = 10, pick = 6
输出：6
```

示例 2：

```
输入：n = 1, pick = 1
输出：1
```

解题思路：

二分查找，注意二分查找边界条件

- $left < right$
- $left <= right$

题解代码：

```
/**
 * Forward declaration of guess API.
 * @param  num   your guess
 * @return       -1 if num is lower than the guess number
 *                1 if num is higher than the guess number
 *               otherwise return 0
```

```
 * int guess(int num);
 */

public class Solution extends GuessGame {
    public int guessNumber(int n) {
        int left = 1, right = n;
        while (left < right) { // 循环直至区间左右端点相同
            int mid = left + (right - left) / 2; // 防止计算时溢出
            if (guess(mid) <= 0) {
                right = mid; // 答案在区间 [left, mid] 中
            } else {
                left = mid + 1; // 答案在区间 [mid+1, right] 中
            }
        }
        // 此时有 left == right，区间缩为一个点，即为答案
        return left;
    }
}
```

852、山脉数组的峰顶索引

符合下列属性的数组 arr 称为 山脉数组：
arr.length >= 3
存在 i（0 < i < arr.length - 1）使得：
arr[0] < arr[1] < ... arr[i-1] < arr[i]
arr[i] > arr[i+1] > ... > arr[arr.length - 1]
给你由整数组成的山脉数组 arr，返回任何满足 arr[0] < arr[1] < ... arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1] 的下标 i。

示例 1：

```
输入：arr = [0,1,0]
输出：1
```

示例 2：

```
输入：arr = [0,2,1,0]
输出：1
```

解题思路：

- 一次遍历
- 二分查找

题解代码：

```
//一次遍历
class Solution {
    public int peakIndexInMountainArray(int[] arr) {
        int n = arr.length;
        int ans = -1;
        for (int i = 1; i < n - 1; ++i) {
            if (arr[i] > arr[i + 1]) {
                ans = i;
                break;
            }
        }
        return ans;
    }
```

```
}
```

```java
//二分查找
class Solution {
    public int peakIndexInMountainArray(int[] arr) {
        int n = arr.length;
        int left = 1, right = n - 2, ans = 0;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] > arr[mid + 1]) {
                ans = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}
```

1818、绝对差值和

给你两个正整数数组 nums1 和 nums2 ， 数组的长度都是 n 。

数组 nums1 和 nums2 的 绝对差值和 定义为所有 |nums1[i] - nums2[i]|（0 <= i < n）的 总和（下标从 0 开始）。

你可以选用 nums1 中的 任意一个 元素来替换 nums1 中的 至多 一个元素，以 最小化 绝对差值和。

在替换数组 nums1 中最多一个元素 之后 ， 返回最小绝对差值和。因为答案可能很大，所以需要对 109 + 7 取余 后返回。

|x| 定义为：

如果 x >= 0 ， 值为 x ， 或者
如果 x <= 0 ， 值为 -x

示例 1：

```
输入：nums1 = [1,7,5], nums2 = [2,3,5]
输出：3
解释：有两种可能的最优方案：
将第二个元素替换为第一个元素：[1,7,5] => [1,1,5] ，或者
将第二个元素替换为第三个元素：[1,7,5] => [1,5,5]
两种方案的绝对差值和都是 |1-2| + (|1-3| 或者 |5-3|) + |5-5| = 3
```

示例 2：

```
输入：nums1 = [2,4,6,8,10], nums2 = [2,4,6,8,10]
输出：0
解释：nums1 和 nums2 相等，所以不用替换元素。绝对差值和为 0
```

解题思路：

错误思路（找到最大差值，减小最小差值）

```
/*
最后执行的输入：
[1,28,21]
[9,21,20]
```

```
*/
class Solution {
    public int minAbsoluteSumDiff(int[] nums1, int[] nums2) {
        int n = nums1.length;
        int idx = 0;
        int count = 0;
        int max = 0;
        for(int i = 0; i < n; ++i){
            if(Math.abs(nums1[i] - nums2[i]) > max){
                idx = i;
                max = Math.abs(nums1[i] - nums2[i]);
            }
            count += Math.abs(nums1[i] - nums2[i]);
        }
        if(count == 0) return 0;
        int min = max;
        for(int i = 0; i < n; ++i){
            if(i == idx) continue;
            min = Math.min(Math.abs(nums1[i] - nums2[idx]),min);
        }
        return (count - max + min) % 1000000007;
    }
}
```

正确思路

排序+二分查找，最大化以下差值

$|nums_1[i] - nums_2[i]| - |nums_1[j] - nums_2[i]|$

我们希望能最大化该差值，这样可以使得答案尽可能小。因为我们只能修改一个位置，所以我们需要检查每一个 i 对应的差值的最大值。当 i 确定时，该式的前半部分的值即可确定，而后半部分的值取决于 j 的选择。观察该式，我们只需要找到和 $nums_2[i]$尽可能接近的$nums_1[j]$ 即可。

题解代码：

```
class Solution {
    public int minAbsoluteSumDiff(int[] nums1, int[] nums2) {
        final int MOD = 1000000007;
        int n = nums1.length;
        int[] rec = new int[n];
        System.arraycopy(nums1, 0, rec, 0, n);
        Arrays.sort(rec);
        int sum = 0, maxn = 0;
        for (int i = 0; i < n; i++) {
            int diff = Math.abs(nums1[i] - nums2[i]);
            sum = (sum + diff) % MOD;
            int j = binarySearch(rec, nums2[i]);
            if (j < n) {
                maxn = Math.max(maxn, diff - (rec[j] - nums2[i]));
            }
            if (j > 0) {
                maxn = Math.max(maxn, diff - (nums2[i] - rec[j - 1]));
            }
        }
        return (sum - maxn + MOD) % MOD;
    }

    public int binarySearch(int[] rec, int target) {
        int low = 0, high = rec.length - 1;
        if (rec[high] < target) {
```

```
            return high + 1;
        }
        while (low < high) {
            int mid = (high - low) / 2 + low;
            if (rec[mid] < target) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
}
```

34、在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 target，返回 [-1, -1]。

进阶：

你可以设计并实现时间复杂度为 O(log n) 的算法解决此问题吗？

示例 1：

```
输入：nums = [5,7,7,8,8,10], target = 8
输出：[3,4]
```

示例 2：

```
输入：nums = [5,7,7,8,8,10], target = 6
输出：[-1,-1]
```

解题思路：

二分查找，找寻左右边界

二分查找中，寻找 $leftIdx$ 即为在数组中寻找第一个大于等于 $target$ 的下标，寻找 $rightIdx$ 即为在数组中寻找第一个大于 $target$ 的下标，然后将下标减一。两者的判断条件不同，为了代码的复用，我们定义 $binarySearch(nums, target, lower)$ 表示在 $nums$ 数组中二分查找 $target$ 的位置，如果 $lower$ 为 true，则查找第一个大于等于 $target$ 的下标，否则查找第一个大于 $target$ 的下标

题解代码：

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int leftIdx = binarySearch(nums, target, true);
        int rightIdx = binarySearch(nums, target, false) - 1;
        if (leftIdx <= rightIdx && rightIdx < nums.length && nums[leftIdx] ==
target && nums[rightIdx] == target) {
            return new int[]{leftIdx, rightIdx};
        }
        return new int[]{-1, -1};
    }

    public int binarySearch(int[] nums, int target, boolean lower) {
        int left = 0, right = nums.length - 1, ans = nums.length;
        while (left <= right) {
            int mid = (left + right) / 2;
```

```
            if (nums[mid] > target || (lower && nums[mid] >= target)) {
                right = mid - 1;
                ans = mid;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}
```

1713、得到子序列的最少操作次数

给你一个数组 target，包含若干 互不相同 的整数，以及另一个整数数组 arr，arr 可能 包含重复元素。

每一次操作中，你可以在 arr 的任意位置插入任一整数。比方说，如果 arr = [1,4,1,2]，那么你可以在中间添加 3 得到 [1,4,3,1,2]。你可以在数组最开始或最后面添加整数。

请你返回 最少 操作次数，使得 target 成为 arr 的一个子序列。

一个数组的 子序列 指的是删除原数组的某些元素（可能一个元素都不删除），同时不改变其余元素的相对顺序得到的数组。比方说，[2,7,4] 是 [4,2,3,7,2,1,4] 的子序列（加粗元素），但 [2,4,2] 不是子序列。

示例 1：

```
输入：target = [5,1,3], arr = [9,4,2,3,4]
输出：2
解释：你可以添加 5 和 1 ，使得 arr 变为 [5,9,4,1,2,3,4] ，target 为 arr 的子序列。
```

示例 2：

```
输入：target = [6,4,8,1,3,2], arr = [4,7,6,2,3,8,6,1]
输出：3
```

解题思路：

转换为求最长上升子序列。

将arr中的元素转换成该元素在target中的下标（去掉不在target中的元素），可以得到一个新数组，举示例2进行说明

$arr' = [1, 0, 5, 4, 0, 3]$

若将target也做上述转换，这相当与将每个元素变为其下标，得

$target' = [0, 1, 2, 3, 4, 5]$

则求原数组的最长公共子序列等价于求上述转换后的两数组的最长公共子序列。参考题目300得贪心+二分查找得解法。

题解代码：

```
class Solution {
    public int minOperations(int[] target, int[] arr) {
        int n = target.length;
        Map<Integer, Integer> pos = new HashMap<Integer, Integer>();
        for (int i = 0; i < n; ++i) {
            pos.put(target[i], i);
        }
        List<Integer> d = new ArrayList<Integer>();
        for (int val : arr) {
            if (pos.containsKey(val)) {
```

```
                int idx = pos.get(val);
                int it = binarySearch(d, idx);
                if (it != d.size()) {
                    d.set(it, idx);
                } else {
                    d.add(idx);
                }
            }
        }
        return n - d.size();
    }

    public int binarySearch(List<Integer> d, int target) {
        int size = d.size();
        if (size == 0 || d.get(size - 1) < target) {
            return size;
        }
        int low = 0, high = size - 1;
        while (low < high) {
            int mid = (high - low) / 2 + low;
            if (d.get(mid) < target) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
}
```

数学

877、石子游戏

亚历克斯和李用几堆石子在做游戏。偶数堆石子排成一行，每堆都有正整数颗石子 piles[i] 。

游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。

亚历克斯和李轮流进行，亚历克斯先开始。 每回合，玩家从行的开始或结束处取走整堆石头。 这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 true ，当李赢得比赛时返回 false 。

示例：

```
输入：[5,3,4,5]
输出：true
解释：
亚历克斯先开始，只能拿前 5 颗或后 5 颗石子 。
假设他取了前 5 颗，这一行就变成了 [3,4,5] 。
如果李拿走前 3 颗，那么剩下的是 [4,5]，亚历克斯拿走后 5 颗赢得 10 分。
如果李拿走后 5 颗，那么剩下的是 [3,4]，亚历克斯拿走后 4 颗赢得 9 分。
这表明，取前 5 颗石子对亚历克斯来说是一个胜利的举动，所以我们返回 true 。
```

解题思路：

- 动态规划

  定义二维数组 dp，其行数和列数都等于石子的堆数，$dp[i][j]$ 表示当剩下的石子堆为下标 i 到下标 j 时，当前玩家与另一个玩家的石子数量之差的最大值，注意当前玩家不一定是先手 Alex。

  只有当 $i \leq j$ 时，剩下的石子堆才有意义，因此当 $i > j$ 时，$dp[i][j] = 0$。

当 $i = j$ 时，只剩下一堆石子，当前玩家只能取走这堆石子，因此对于所有 $0 \le i < nums.\text{length}$，都有 $dp[i][i] = piles[i]$。

当 $i < j$ 时，当前玩家可以选择取走 $piles[i]$ 或 $piles[j]$，然后轮到另一个玩家在剩下的石子堆中取走石子。在两种方案中，当前玩家会选择最优的方案，使得自己的石子数量最大化。因此可以得到如下状态转移方程：

$dp[i][j] = \max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1])$

最后判断 $dp[0][piles.\text{length} - 1]$ 的值，如果大于 0，则 Alex 的石子数量大于 Lee 的石子数量，因此 Alex 赢得比赛，否则 Lee 赢得比赛。

- 数学

  先手总是赢

题解代码：

```
class Solution {
    public boolean stoneGame(int[] piles) {
        int length = piles.length;
        int[][] dp = new int[length][length];
        for (int i = 0; i < length; i++) {
            dp[i][i] = piles[i];
        }
        for (int i = length - 2; i >= 0; i--) {
            for (int j = i + 1; j < length; j++) {
                dp[i][j] = Math.max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
            }
        }
        return dp[0][length - 1] > 0;
    }
}
```

```
class Solution {
    public boolean stoneGame(int[] piles) {
        return true;
    }
}
```

65、有效数字

有效数字（按顺序）可以分成以下几个部分：

一个 小数 或者 整数
（可选）一个 'e' 或 'E' ，后面跟着一个 整数
小数（按顺序）可以分成以下几个部分：

（可选）一个符号字符（'+' 或 '-'）
下述格式之一：
至少一位数字，后面跟着一个点 '.'
至少一位数字，后面跟着一个点 '.' ，后面再跟着至少一位数字
一个点 '.' ，后面跟着至少一位数字
整数（按顺序）可以分成以下几个部分：

（可选）一个符号字符（'+' 或 '-'）
至少一位数字
部分有效数字列举如下：

["2", "0089", "-0.1", "+3.14", "4.", "-.9", "2e10", "-90E3", "3e+7", "+6e-1", "53.5e93", "-123.456e789"]
部分无效数字列举如下：

["abc", "1a", "1e", "e3", "99e2.5", "--6", "-+3", "95a54e53"]
给你一个字符串 s ， 如果 s 是一个 有效数字 ， 请返回 true 。

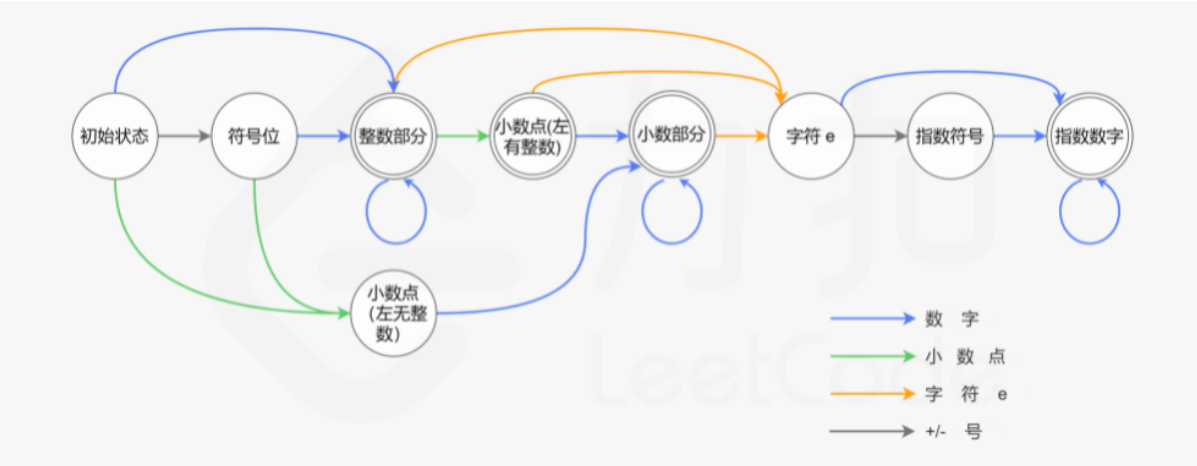
示例 1：

```
输入：s = "0"
输出：true
```

解题思路：

确定有限状态自动机

> 初始状态
>
> 符号位
>
> 整数部分
>
> 左侧有整数的小数点
>
> 左侧无整数的小数点（根据前面的第二条额外规则，需要对左侧有无整数的两种小数点做区分）
>
> 小数部分
>
> 字符e
>
> 指数部分的符号位
>
> 指数部分的整数部分



题解代码：

```java
class Solution65 {
    public boolean isNumber(String s) {
        boolean ans = true, occur = false;
        int n = s.length();
        for (int i=0; i<n; i++) {
            char ch = s.charAt(i);
            if (ch == '+' || ch == '-') {
                if (!(i < n-1 && ((s.charAt(i+1)-'0' >= 0 && s.charAt(i+1)-'0' <=
9) || s.charAt(i+1) == '.')))
                    return false;
            }
            else if (ch == '.') {
                if (!((i > 0 && s.charAt(i-1)-'0'>=0 && s.charAt(i-1)-'0'<=9) ||
                        (i < n-1 && s.charAt(i+1)-'0'>=0 && s.charAt(i+1)-'0'<=9))
|| occur)
                    return false;
                occur = true;
```

```
            }
            else if (ch == 'e' || ch == 'E') {
                if (i == 0 || i == n-1)
                    return false;
                else {
                    for (int j=i+1; j<n; j++) {
                        char c = s.charAt(j);
                        if ((c == '+' || c == '-') && !(j == i+1 && j != n-1))
                            return false;
                        if (c == '.' || (c-'a'>=0 && c-'a'<=25) || (c-'A'>=0 && c-'A'<=25))
                            return false;
                    }
                    break;
                }
            }
            else if (ch-'0' >= 0 && ch-'0'<=9) {
                if (i < n-1 && (s.charAt(i+1) == '+' || s.charAt(i+1) == '-'))
                    return false;
            }
            else
                return false;
        }
        return ans;
    }
}
```

483、最小好进制

对于给定的整数 n, 如果n的k（k>=2）进制数的所有数位全为1，则称 k（k>=2）是 n 的一个好进制。

以字符串的形式给出 n, 以字符串的形式返回 n 的最小好进制。

示例 1：

```
输入："13"
输出："3"
解释：13 的 3 进制是 111。
```

示例 2：

```
输入："4681"
输出："8"
解释：4681 的 8 进制是 11111。
```

解题思路：

假设正整数 n 在 $k\ (k \geq 2)$ 进制下的所有数位都为 1，且位数为 m + 1，那么有：

$n = k^0 + k^1 + k^2 + \cdots + k^m$

首先讨论两种特殊情况：

m=0，此时 $n = 1$，而题目保证 $n \geq 3$，所以本题中 $m > 0$。
m=1，此时 $n = (11)_k$，即 $k = n - 1 \geq 2$，这保证了本题有解。

然后我们分别证明一般情况下的两个结论，以帮助解决本题。

结论一：$m < log_k n$ (等比数列求和)

结论二：$k < \sqrt[m]{n} < k + 1$ (二项式定理)

题解代码：

```java
class Solution {
    public String smallestGoodBase(String n) {
        long nVal = Long.parseLong(n);
        int mMax = (int) Math.floor(Math.log(nVal) / Math.log(2));
        for (int m = mMax; m > 1; m--) {
            int k = (int) Math.pow(nVal, 1.0 / m);
            long mul = 1, sum = 1;
            for (int i = 0; i < m; i++) {
                mul *= k;
                sum += mul;
            }
            if (sum == nVal) {
                return Integer.toString(k);
            }
        }
        return Long.toString(nVal - 1);
    }
}
```

470、用rand7()实现rand10()

已有方法 rand7 可生成 1 到 7 范围内的均匀随机整数，试写一个方法 rand10 生成 1 到 10 范围内的均匀随机整数。

不要使用系统的 Math.random() 方法。

示例 1:

```
输入: 1
输出: [7]
```

示例 2:

```
输入: 2
输出: [8,4]
```

解题思路:

拒绝采样

可以调用两次 $Rand7()$，那么可以生成 $[1, 49]$ 之间的随机整数，我们只用到其中的前 40 个用来实现 $Rand10()$，而拒绝剩下的 99 个数

题解代码:

```java
class Solution extends SolBase {
    public int rand10() {
        int row, col, idx;
        do {
            row = rand7();
            col = rand7();
            idx = col + (row - 1) * 7;
        } while (idx > 40);
        return 1 + (idx - 1) % 10;
    }
}
```

回溯

1239、串联字符串的最长长度

给定一个字符串数组 arr，字符串 s 是将 arr 某一子序列字符串连接所得的字符串，如果 s 中的每一个字符都只出现过一次，那么它就是一个可行解。

请返回所有可行解 s 中最长长度。

示例 1：

```
输入：arr = ["un","iq","ue"]
输出：4
解释：所有可能的串联组合是 "","un","iq","ue","uniq" 和 "ique"，最大长度为 4。
```

示例 2：

```
输入：arr = ["cha","r","act","ers"]
输出：6
解释：可能的解答有 "chaers" 和 "acters"。
```

解题思路：

回溯 + 位运算

将字符串表示位二进制形式（mask >> ch 1 << ch）

题解代码：

```java
class Solution {
    int ans = 0;

    public int maxLength(List<String> arr) {
        List<Integer> masks = new ArrayList<>();
        for(String s : arr){
            int mask = 0;
            for(int i = 0; i < s.length(); ++i){
                int ch = s.charAt(i) - 'a';
                if(((mask >> ch) & 1) != 0){ // 若 mask 已有 ch，则说明 s 含有重复字母，无法构成可行解
                    mask = 0;
                    break;
                }
                mask |= 1 << ch; // 将 ch 加入 mask 中
            }
            if(mask > 0){
                masks.add(mask);
            }
        }
        backtrack(masks,0,0);
        return ans;
    }
    public void backtrack(List<Integer> masks, int pos, int mask){
        if(pos == masks.size()){
            ans = Math.max(ans,Integer.bitCount(mask));
            return;
        }
        if((mask & masks.get(pos)) == 0){ // mask 和 masks[pos] 无公共元素
            backtrack(masks,pos+1,mask | masks.get(pos));
        }
        backtrack(masks,pos+1,mask);  //不选择当前字符
```

```
    }
}
```

剑指 Offer 38. 字符串的排列

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

```
输入：s = "abc"
输出：["abc","acb","bac","bca","cab","cba"]
```

解题思路:

回溯法，需要去重，去重代码：

```
if(vis[j] || (j>0 && !vis[j-1] && arr[j-1] == arr[j])){
    continue;
}
```

题解代码:

```
class Solution {
    List<String> res;
    boolean[] vis;
    public String[] permutation(String s) {
        int n = s.length();
        res = new ArrayList<String>();
        vis = new boolean[n];
        char[] arr = s.toCharArray();
        Arrays.sort(arr);
        StringBuffer perm = new StringBuffer();
        backtrack(arr, n, perm);
        int size = res.size();
        String[] recArr = new String[size];
        for (int i = 0; i < size; i++) {
            recArr[i] = res.get(i);
        }
        return recArr;
    }
    public void backtrack(char []arr, int n, StringBuffer perm){
        if(perm.length() == n){
            res.add(perm.toString());
            return;
        }
        for(int j = 0; j < n; ++j){
            if(vis[j] || (j>0 && !vis[j-1] && arr[j-1] == arr[j])){
                continue;
            }
            vis[j] = true;
            perm.append(arr[j]);
            backtrack(arr,n,perm);
            perm.deleteCharAt(perm.length() - 1);
            vis[j] = false;
        }
    }
}
```

贪心

406、根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面 正好 有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

示例 1：

```
输入：people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
输出：[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
解释：
编号为 0 的人身高为 5 ，没有身高更高或者相同的人排在他前面。
编号为 1 的人身高为 7 ，没有身高更高或者相同的人排在他前面。
编号为 2 的人身高为 5 ，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。
编号为 3 的人身高为 6 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
编号为 4 的人身高为 4 ，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。
编号为 5 的人身高为 7 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
因此 [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]] 是重新构造后的队列。
```

解题思路：

考虑是依据哪个维度进行排序，先以身高h进行排序，若身高相同k值小的排在前面；接着按照排序的顺序依次插入，插入的位置依据维度k。

题解代码：

```java
class Solution {
    public int[][] reconstructQueue(int[][] people) {
        Arrays.sort(people, new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
                if (o1[0] != o2[0]) {
                    return Integer.compare(o2[0],o1[0]);
                } else {
                    return Integer.compare(o1[1],o2[1]);
                }
            }
        });
        LinkedList<int[]> que = new LinkedList<>();

        for (int[] p : people) {
            que.add(p[1],p);
        }

        return que.toArray(new int[people.length][]);
    }
}
```

452、用最少数量的剪引爆气球

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以纵坐标并不重要，因此只要知道开始和结束的横坐标就足够了。开始坐标总是小于结束坐标。

一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

给你一个数组 points ， 其中 points [i] = [xstart,xend] ， 返回引爆所有气球所必须射出的最小弓箭数。

示例 1：

```
输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：对于该样例，x = 6 可以射爆 [2,8],[1,6] 两个气球，以及 x = 11 射爆另外两个气球
```

示例 2：

```
输入：points = [[1,2],[3,4],[5,6],[7,8]]
输出：4
```

解题思路：

按照数组的起始位置进行排序，然后从前向后遍历，如果气球重叠了更新重叠气球最小右边界

$points[i][1] = min(points[i-1][1], points[i][1])$

否则弓箭数量加一

题解代码：

```java
class Solution {
    public int findMinArrowShots(int[][] points) {
        if(points.length == 0) return 0;
        Arrays.sort(points,(o1,o2) -> Integer.compare(o1[0],o2[0]));

        int count = 1;
        for(int i = 1; i < points.length; i++){
            if(points[i][0] > points[i-1][1]){
                count++;
            }else{
                points[i][1] = Math.min(points[i][1],points[i-1][1]);
            }
        }
        return count;
    }
}
```

1946、子字符串突变后可能得到的最大整数

给你一个字符串 num ， 该字符串表示一个大整数。另给你一个长度为 10 且 下标从 0 开始 的整数数组 change ，该数组将 0-9 中的每个数字映射到另一个数字。更规范的说法是，数字 d 映射为数字 change[d] 。

你可以选择 突变 num 的任一子字符串。突变 子字符串意味着将每位数字 num[i] 替换为该数字在 change 中的映射（也就是说，将 num[i] 替换为 change[num[i]]）。

请你找出在对 num 的任一子字符串执行突变操作（也可以不执行）后，可能得到的 最大整数 ，并用字符串表示返回。

子字符串 是字符串中的一个连续序列。

示例 1：

```
输入：num = "132", change = [9,8,5,0,3,6,4,2,6,8]
输出："832"
解释：替换子字符串 "1"：

1 映射为 change[1] = 8 。
因此 "132" 变为 "832" 。
"832" 是可以构造的最大整数，所以返回它的字符串表示。
```

解题思路：

贪心，找替换的左右边界，注意是子字符串

题解代码：

```python
class Solution:
    def maximumNumber(self, num: str, change: List[int]) -> str:
        n = len(num)
        num = list(num)
        for i in range(n):
            # 寻找第一个突变后数值更大的位作为左边界
            if change[int(num[i])] > int(num[i]):
                # 尝试更新右边界
                while i < n and change[int(num[i])] >= int(num[i]):
                    num[i] = str(change[int(num[i])])
                    i += 1
                break
        return ''.join(num)
```

BFS

752、打开转盘锁

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' 。每个拨轮可以自由旋转：例如把 '9' 变为 '0'，'0' 变为 '9' 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000' ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

示例 1:

```
输入：deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出：6
解释：
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
因为当拨动到 "0102" 时这个锁就会被锁定。
```

解题思路：

广度优先搜索，使用队列

题解代码：

```java
class Solution {
    public int openLock(String[] deadends, String target) {
        if("0000".equals(target)){
            return 0;
```

```java
        }
        Set<String> dead = new HashSet<>();
        for(String deadend : deadends){
            dead.add(deadend);
        }
        if(dead.contains("0000")){
            return -1;
        }

        int step = 0;
        Queue<String> queue = new LinkedList<>();
        queue.offer("0000");
        Set<String> seen = new HashSet<>();
        seen.add("0000");

        while(!queue.isEmpty()){
            ++step;
            int size = queue.size();
            for(int i = 0; i < size; ++i){
                String status = queue.poll();
                for(String newStatus : get(status)){
                    if(!seen.contains(newStatus) && !dead.contains(newStatus)){
                        if(newStatus.equals(target)){
                            return step;
                        }
                        queue.offer(newStatus);
                        seen.add(newStatus);
                    }
                }
            }
        }
        return -1;
    }

    public char numPrev(char x){
        return x == '0' ? '9' : (char)(x - 1);
    }

    public char numSucc(char x){
        return x == '9' ? '0' : (char)(x + 1);
    }

    public List<String> get(String status){
        List<String> ret = new ArrayList<>();
        char[] array = status.toCharArray();
        for(int i = 0; i < 4; ++i){
            char num = array[i];
            array[i] = numPrev(num);
            ret.add(new String(array));
            array[i] = numSucc(num);
            ret.add(new String(array));
            array[i] = num;
        }
        return ret;
    }
}
```

773、滑动谜题

在一个 2 x 3 的板上（board）有 5 块砖瓦，用数字 1~5 来表示, 以及一块空缺用 0 来表示.

一次移动定义为选择 0 与一个相邻的数字（上下左右）进行交换.

最终当板 board 的结果是 [[1,2,3],[4,5,0]] 谜板被解开。

给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1 。

示例：

```
输入：board = [[1,2,3],[4,0,5]]
输出：1
解释：交换 0 和 5 ，1 步完成
```

```
输入：board = [[1,2,3],[5,4,0]]
输出：-1
解释：没有办法完成谜板
```

解题思路：

广度优先搜索，使用队列

题解代码：

```java
class Solution {
    int [][] neighbors = {{1,3},{0,2,4},{1,5},{0,4},{1,3,5},{2,4}};
    public int slidingPuzzle(int[][] board) {
        StringBuffer sb = new StringBuffer();
        for(int i = 0; i < 2; ++i){
            for(int j= 0; j < 3; ++j){
                sb.append(board[i][j]);
            }
        }
        String initial = sb.toString();
        if("123450".equals(initial)){
            return 0;
        }
        int step = 0;
        Queue<String> queue = new LinkedList<>();
        queue.offer(initial);
        Set<String> seen = new HashSet<>();
        seen.add(initial);
        while(!queue.isEmpty()){
            ++step;
            int size = queue.size();
            for(int i = 0; i < size; ++i){
                String status = queue.poll();
                for(String nextStatus:get(status)){
                    if(!seen.contains(nextStatus)){
                        if("123450".equals(nextStatus)){
                            return step;
                        }
                        queue.offer(nextStatus);
                        seen.add(nextStatus);
                    }
                }
            }
        }
        return -1;
```

```
    }

    public List<String> get(String status){
        List<String> ret = new ArrayList<>();
        char[] array = status.toCharArray();
        int x = status.indexOf('0');
        for(int y : neighbors[x]){
            swap(array,x,y);
            ret.add(new String(array));
            swap(array,x,y);
        }
        return ret;
    }
    public void swap(char[] array, int x, int y){
        char temp = array[x];
        array[x] = array[y];
        array[y] = temp;
    }
}
```

815、公交路线

给你一个数组 routes ， 表示一系列公交线路，其中每个 routes[i] 表示一条公交线路，第 i 辆公交车将会在上面循环行驶。

例如，路线 routes[0] = [1, 5, 7] 表示第 0 辆公交车会一直按序列 1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ... 这样的车站路线行驶。
现在从 source 车站出发（初始时不在公交车上），要前往 target 车站。 期间仅可乘坐公交车。

求出 最少乘坐的公交车数量 。如果不可能到达终点车站，返回 -1 。

示例 1：

```
输入：routes = [[1,2,7],[3,6,7]], source = 1, target = 6
输出：2
解释：最优策略是先乘坐第一辆公交车到达车站 7 ，然后换乘第二辆公交车到车站 6 。
```

示例 2：

```
输入：routes = [[7,12],[4,5,15],[6],[15,19],[9,12,13]], source = 15, target = 12
输出：-1
```

解题思路：

BFS 建立不同路线中相同车站的映射关系

题解代码：

```
class Solution {
    public int numBusesToDestination(int[][] routes, int source, int target) {
        if(source == target){
            return 0;
        }

        int n = routes.length;
        boolean[][] edge = new boolean[n][n];
        Map<Integer,List<Integer>> res = new HashMap<Integer,List<Integer>>();
        for(int i = 0; i < n; ++i){
            for(int site: routes[i]){
                List<Integer> list = res.getOrDefault(site,new ArrayList<>());
```

```
                for(int j : list){
                    edge[i][j] = edge[j][i] = true;
                }
                list.add(i);
                res.put(site,list);
            }
        }

        int[] dis = new int[n];
        Arrays.fill(dis,-1);
        Queue<Integer> que = new LinkedList<>();
        for(int bus: res.getOrDefault(source,new ArrayList<>())){
            dis[bus] = 1;
            que.offer(bus);
        }
        while(!que.isEmpty()){
            int x = que.poll();
            for(int y = 0; y < n; ++y){
                if(edge[x][y] && dis[y] == -1){
                    dis[y] = dis[x] + 1;
                    que.offer(y);
                }
            }
        }
        int ret = Integer.MAX_VALUE;
        for(int bus: res.getOrDefault(target,new ArrayList<>())){
            if(dis[bus] != -1){
                ret = Math.min(ret,dis[bus]);
            }
        }
        return ret == Integer.MAX_VALUE ? -1 : ret;
    }
}
```

542、01矩阵

给定一个由 0 和 1 组成的矩阵 mat ， 请输出一个大小相同的矩阵，其中每一个格子是 mat 中对应位置元素到最近的 0 的距离。

两个相邻元素间的距离为 1 。

示例1：

```
输入：mat = [[0,0,0],[0,1,0],[0,0,0]]
输出：[[0,0,0],[0,1,0],[0,0,0]]
```

解题思路：

多源广度搜索

动态规划

题解代码：

```
class Solution {
    static int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    public int[][] updateMatrix(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        int[][] dist = new int[m][n];
        boolean[][] seen = new boolean[m][n];
```

```
        Queue<int[]> queue = new LinkedList<int[]>();
        // 将所有的 0 添加进初始队列中
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    queue.offer(new int[]{i, j});
                    seen[i][j] = true;
                }
            }
        }

        // 广度优先搜索
        while (!queue.isEmpty()) {
            int[] cell = queue.poll();
            int i = cell[0], j = cell[1];
            for (int d = 0; d < 4; ++d) {
                int ni = i + dirs[d][0];
                int nj = j + dirs[d][1];
                if (ni >= 0 && ni < m && nj >= 0 && nj < n && !seen[ni][nj]) {
                    dist[ni][nj] = dist[i][j] + 1;
                    queue.offer(new int[]{ni, nj});
                    seen[ni][nj] = true;
                }
            }
        }

        return dist;
    }
}
```

```
class Solution:
    def updateMatrix(self, matrix: List[List[int]]) -> List[List[int]]:
        m, n = len(matrix), len(matrix[0])
        # 初始化动态规划的数组，所有的距离值都设置为一个很大的数
        dist = [[10**9] * n for _ in range(m)]
        # 如果 (i, j) 的元素为 0，那么距离为 0
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == 0:
                    dist[i][j] = 0
        # 只有 水平向左移动 和 竖直向上移动，注意动态规划的计算顺序
        for i in range(m):
            for j in range(n):
                if i - 1 >= 0:
                    dist[i][j] = min(dist[i][j], dist[i - 1][j] + 1)
                if j - 1 >= 0:
                    dist[i][j] = min(dist[i][j], dist[i][j - 1] + 1)
        # 只有 水平向左移动 和 竖直向下移动，注意动态规划的计算顺序
        for i in range(m - 1, -1, -1):
            for j in range(n):
                if i + 1 < m:
                    dist[i][j] = min(dist[i][j], dist[i + 1][j] + 1)
                if j - 1 >= 0:
                    dist[i][j] = min(dist[i][j], dist[i][j - 1] + 1)
        # 只有 水平向右移动 和 竖直向上移动，注意动态规划的计算顺序
        for i in range(m):
            for j in range(n - 1, -1, -1):
                if i - 1 >= 0:
                    dist[i][j] = min(dist[i][j], dist[i - 1][j] + 1)
                if j + 1 < n:
```

```
                    dist[i][j] = min(dist[i][j], dist[i][j + 1] + 1)
        # 只有 水平向右移动 和 竖直向下移动，注意动态规划的计算顺序
        for i in range(m - 1, -1, -1):
            for j in range(n - 1, -1, -1):
                if i + 1 < m:
                    dist[i][j] = min(dist[i][j], dist[i + 1][j] + 1)
                if j + 1 < n:
                    dist[i][j] = min(dist[i][j], dist[i][j + 1] + 1)
        return dist
```

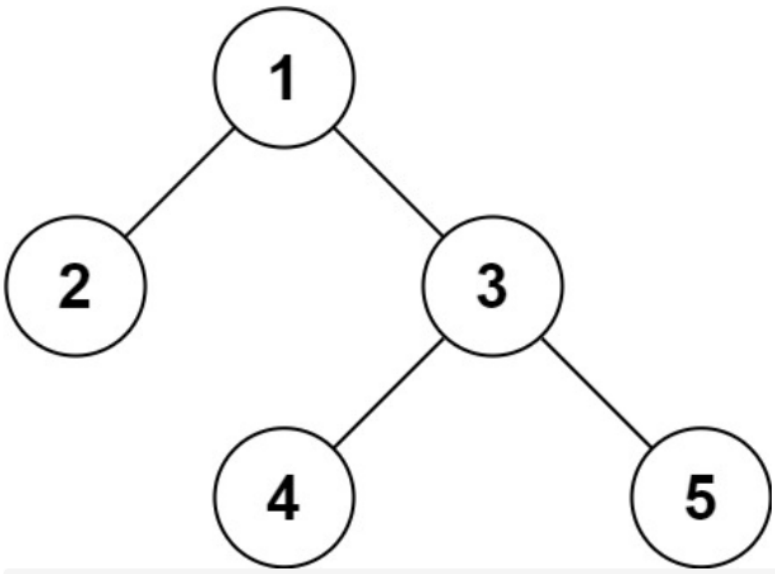
DFS

剑指Offer 37. 序列化二叉树

请实现两个函数，分别用来序列化和反序列化二叉树。

你需要设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例：



输入： root = [1,2,3,null,null,4,5]
输出： [1,2,3,null,null,4,5]

解题思路：

先序、中序、后序遍历都可以

这里先序遍历这颗二叉树，遇到空子树的时候序列化成 None，否则继续递归序列化。那么我们如何反序列化呢？首先我们需要根据，把原先的序列分割开来得到先序遍历的元素列表，然后从左向右遍历这个序列：

- 如果当前的元素为 None，则当前为空树
- 否则先解析这棵树的左子树，再解析它的右子树

题解代码：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
```

```java
 */
public class Codec {
    public String serialize(TreeNode root) {
        return rserialize(root, "");
    }

    public TreeNode deserialize(String data) {
        String[] dataArray = data.split(",");
        List<String> dataList = new LinkedList<String>(Arrays.asList(dataArray));
        return rdeserialize(dataList);
    }

    public String rserialize(TreeNode root, String str) {
        if (root == null) {
            str += "None,";
        } else {
            str += str.valueOf(root.val) + ",";
            str = rserialize(root.left, str);
            str = rserialize(root.right, str);
        }
        return str;
    }

    public TreeNode rdeserialize(List<String> dataList) {
        if (dataList.get(0).equals("None")) {
            dataList.remove(0);
            return null;
        }

        TreeNode root = new TreeNode(Integer.valueOf(dataList.get(0)));
        dataList.remove(0);
        root.left = rdeserialize(dataList);
        root.right = rdeserialize(dataList);

        return root;
    }
}
```

LCP 07、传递信息

小朋友 A 在和 ta 的小伙伴们玩传信息游戏，游戏规则如下：

有 n 名玩家，所有玩家编号分别为 0 ~ n-1，其中小朋友 A 的编号为 0
每个玩家都有固定的若干个可传信息的其他玩家（也可能没有）。传信息的关系是单向的（比如 A 可以向 B 传信息，但 B 不能向 A 传信息）。
每轮信息必须需要传递给另一个人，且信息可重复经过同一个人
给定总玩家数 n，以及按 [玩家编号,对应可传递玩家编号] 关系组成的二维数组 relation。返回信息从小 A (编号 0) 经过 k 轮传递到编号为 n-1 的小伙伴处的方案数；若不能到达，返回 0。

示例 1：

```
输入：n = 5, relation = [[0,2],[2,1],[3,4],[2,3],[1,4],[2,0],[0,4]], k = 3
输出：3
解释：信息从小 A 编号 0 处开始，经 3 轮传递，到达编号 4。共有 3 种方案，分别是 0->2->0->4， 0->2->1->4， 0->2->3->4。
```

示例 2：

```
输入：n = 3, relation = [[0,2],[2,1]], k = 2
输出：0
解释：信息不能从小 A 处经过 2 轮传递到编号 2
```

解题思路：

DFS或者BFS，将输入转换为一对多的关系形式

题解代码：

```java
class Solution {
    int ways, n, k;
    List<List<Integer>> edges;

    public int numWays(int n, int[][] relation, int k) {
        ways = 0;
        this.n = n;
        this.k = k;
        edges = new ArrayList<List<Integer>>();
        for (int i = 0; i < n; i++) {
            edges.add(new ArrayList<Integer>());
        }
        for (int[] edge : relation) {
            int src = edge[0], dst = edge[1];
            edges.get(src).add(dst);
        }
        dfs(0, 0);
        return ways;
    }

    public void dfs(int index, int steps) {
        if (steps == k) {
            if (index == n - 1) {
                ways++;
            }
            return;
        }
        List<Integer> list = edges.get(index);
        for (int nextIndex : list) {
            dfs(nextIndex, steps + 1);
        }
    }
}
```

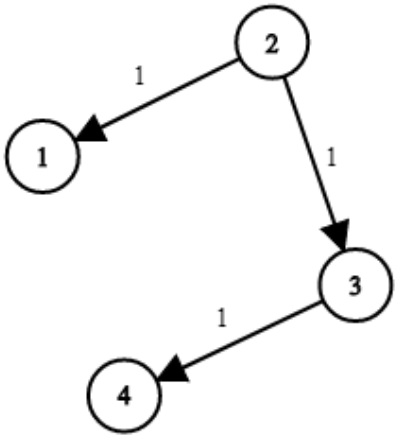
743、网络延迟时间

有 n 个网络节点，标记为 1 到 n。

给你一个列表 times，表示信号经过 有向 边的传递时间。 times[i] = (ui, vi, wi)，其中 ui 是源节点，vi 是目标节点， wi 是一个信号从源节点传递到目标节点的时间。

现在，从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 -1 。

示例 1：




```
输入: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
输出: 2
```

解题思路:

BFS DFS Dijkstra

题解代码:

```java
class Solution {
    public int networkDelayTime(int[][] times, int N, int K) {
        Map<Integer, List<int[]>> map = new HashMap<>();
        // 初始化邻接表
        for (int[] t : times) {
            map.computeIfAbsent(t[0], k -> new ArrayList<>()).add(new int[]{t[1], t[2]});
        }

        // 初始化dis数组和vis数组
        int[] dis = new int[N + 1];
        Arrays.fill(dis, 0x3f3f3f3f);
        boolean[] vis = new boolean[N + 1];

        // 起点的dis为0，但是别忘记0也要搞一下，因为它是不参与的，我计算结果的时候直接用了stream，所以这个0也就要初始化下了
        dis[K] = 0;
        dis[0] = 0;

        // new一个小堆出来，按照dis升序排，一定要让它从小到大排，省去了松弛工作
        PriorityQueue<Integer> queue = new PriorityQueue<>((o1, o2) -> dis[o1] - dis[o2]);
        // 把起点放进去
        queue.offer(K);

        while (!queue.isEmpty()) {
            // 当队列不空，拿出一个源出来
            Integer poll = queue.poll();
                if(vis[poll]) continue;
            // 把它标记为访问过
            vis[poll] = true;
            // 遍历它的邻居们，当然可能没邻居，这里用getOrDefault处理就很方便
            List<int[]> list = map.getOrDefault(poll, Collections.emptyList());
            for (int[] arr : list) {
                int next = arr[0];
                // 如果这个邻居访问过了，继续
                if (vis[next]) continue;
                // 更新到这个邻居的最短距离，看看是不是当前poll出来的节点到它更近一点
```

```
                dis[next] = Math.min(dis[next], dis[poll] + arr[1]);
                queue.offer(next);
            }
        }
        // 拿到数组中的最大值比较下，返回结果
        int res = Arrays.stream(dis).max().getAsInt();
        return res == 0x3f3f3f3f ? -1 : res;
    }
}
```

802、找到最终的安全状态

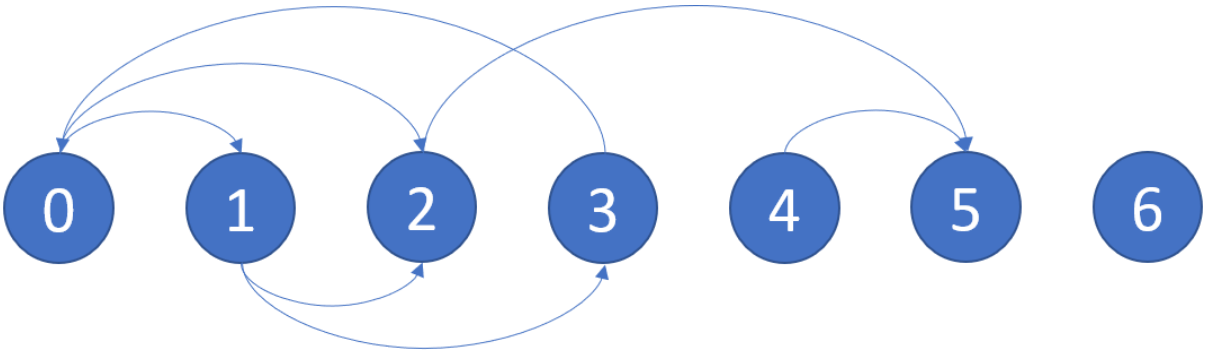
在有向图中，以某个节点为起始节点，从该点出发，每一步沿着图中的一条有向边行走。如果到达的节点是终点（即它没有连出的有向边），则停止。

对于一个起始节点，如果从该节点出发，无论每一步选择沿哪条有向边行走，最后必然在有限步内到达终点，则将该起始节点称作是 安全 的。

返回一个由图中所有安全的起始节点组成的数组作为答案。答案数组中的元素应当按 升序 排列。

该有向图有 n 个节点，按 0 到 n - 1 编号，其中 n 是 graph 的节点数。图以下述形式给出：graph[i] 是编号 j 节点的一个列表，满足 (i, j) 是图的一条有向边。

示例 1：




```
输入：graph = [[1,2],[2,3],[5],[0],[5],[],[]]
输出：[2,4,5,6]
解释：示意图如上。
```

解题思路：

深度优先搜索 **三色法**

我们可以使用深度优先搜索来找环，并在深度优先搜索时，用三种颜色对节点进行标记，标记的规则如下：

白色（用 0 表示）：该节点尚未被访问；
灰色（用 1 表示）：该节点位于递归栈中，或者在某个环上；
黑色（用 2 表示）：该节点搜索完毕，是一个安全节点。

题解代码：

```
class Solution {
    public List<Integer> eventualSafeNodes(int[][] graph) {
        int n = graph.length;
        int[] color = new int[n];
        List<Integer> ans = new ArrayList<Integer>();
        for (int i = 0; i < n; ++i) {
            if (safe(graph, color, i)) {
                ans.add(i);
            }
        }
```

```
        return ans;
    }

    public boolean safe(int[][] graph, int[] color, int x) {
        if (color[x] > 0) {
            return color[x] == 2;
        }
        color[x] = 1;
        for (int y : graph[x]) {
            if (!safe(graph, color, y)) {
                return false;
            }
        }
        color[x] = 2;
        return true;
    }
}
```

动态规划

剑指offer 49、丑数

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

示例:

```
输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
```

题解代码:

```
class Solution {
    public int nthUglyNumber(int n) {
        int a = 0, b = 0, c = 0;
        int[] dp = new int[n];
        dp[0] = 1;
        for(int i = 1; i < n; i++) {
            int n2 = dp[a] * 2, n3 = dp[b] * 3, n5 = dp[c] * 5;
            dp[i] = Math.min(Math.min(n2, n3), n5);
            if(dp[i] == n2) a++;
            if(dp[i] == n3) b++;
            if(dp[i] == n5) c++;
        }
        return dp[n - 1];
    }
}

```

1049、最后一块石头的重量II

有一堆石头，用整数数组 stones 表示。其中 stones[i] 表示第 i 块石头的重量。

每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：

如果 x == y，那么两块石头都会被完全粉碎；
如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
最后，最多只会剩下一块 石头。返回此石头 最小的可能重量 。如果没有石头剩下，就返回 0。

示例 1：

```
输入：stones = [2,7,4,1,8,1]
输出：1
解释：
组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，
组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。
```

示例 2：

```
输入：stones = [31,26,33,21,40]
输出：5
```

示例 3：

```
输入：stones = [1,2]
输出：1
```

解题思路：无论以何种顺序粉碎石头，最后一块石头的重量总是可以表示成

$$\sum_{i=1}^{n-1} k_i \times stones_i, k_i \in -1, 1$$

记石头的总重量为sum, k_i = -1的石头的重量之和为neg, 则其余的k_i = 1的石头的重量之和为sum - neg。则有：

$$\sum_{i=0}^{n-1} k_i \times stones_i = (sum - neg) - neg = sum - 2 \cdot neg$$

要使最后一块石头的重量尽可能地小，neg需要在不超过[*sum*/2] 的前提下尽可能地大。

定义二维布尔数组dp，其中$dp[i+1][j]$ 表示前i个石头能否凑出重量j。特别地，$dp[0][]$ 为不选任何石头的状态，因此除了 $dp[0][0]$为真，其余 $dp[0][j]$全为假。

对于第 i 个石头，考虑凑出重量 j：

- 若$j < stones[i]$，则不能选第 i 个石头，此时有$dp[i+1][j] = dp[i][j]$；
- 若$j \geq stones[i]$，存在选或不选两种决策，不选时有 $dp[i+1][j] = dp[i][j]$，选时需要考虑能否凑出重量 $j - stones$[i]，即 $dp[i+1][j] = dp[i][j - stones[i]]$。若二者均为假则 $dp[i+1][j]$ 为假，否则 $dp[i+1][j]$ 为真。

因此状态转移方程如下：

$$dp[i+1][j] = \begin{cases} dp[i][j], & j < stones[i] \\ dp[i][j] \vee dp[i][j - stones[i]], & j \geq stones[i] \end{cases}$$

其中 $\vee$表示逻辑或运算。求出$dp[n][]$后，所有为真的 $dp[n][j]$ 中，最大的 j 即为 neg 能取到的最大值。代入 $sum - 2 \cdot neg$ 中即得到最后一块石头的最小重量。

题解代码：

```java
class Solution {
    public int lastStoneWeightII(int[] stones) {
        int sum = 0;
        for (int weight : stones) {
            sum += weight;
        }
        int n = stones.length, m = sum / 2;
        boolean[][] dp = new boolean[n + 1][m + 1];
```

```
        dp[0][0] = true;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j <= m; ++j) {
                if (j < stones[i]) {
                    dp[i + 1][j] = dp[i][j];
                } else {
                    dp[i + 1][j] = dp[i][j] || dp[i][j - stones[i]];
                }
            }
        }
        for (int j = m;; --j) {
            if (dp[n][j]) {
                return sum - 2 * j;
            }
        }
    }
}
```

322、零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

示例 1：

```
输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1
```

示例 2：

```
输入：coins = [2], amount = 3
输出：-1
```

解题思路：

- 回溯
- 动态规划

  $dp[i]$ 表示组成金额i所需最少的硬币数量。

  $dp[0]=0,\ dp[i]=min(dp[i],dp[i-coin[j]]+1)$

题解代码：

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [float('inf')] * (amount + 1)
        dp[0] = 0
        for coin in coins:
            for x in range(coin,amount+1):
                dp[x] = min(dp[x],dp[x-coin] + 1)
        return dp[amount] if dp[amount] != float('inf') else -1
```

518、零钱兑换II

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

解题思路:

通过动态规划的方法计算可能的组合数。用 $dp[x]$ 表示金额之和等于 xx 的硬币组合数，目标是求$dp[amount]$。

- 初始化$dp[0]=1$;
- 遍历 $coins$，对于其中的每个元素 $coin$，进行如下操作:
  - 遍历 i 从 $coin$ 到 $amount$，将 $dp[i-coin]$的值加到 $dp[i]$。
- 最终得到 $dp[amount]$ 的值即为答案。

题解代码:

```java
class Solution {
    public int change(int amount, int[] coins) {
        int[] dp = new int[amount + 1];
        dp[0] = 1;
        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                dp[i] += dp[i - coin];
            }
        }
        return dp[amount];
    }
}
```

377、组合总和IV

给你一个由 不同 整数组成的数组 nums ，和一个目标整数 target 。请你从 nums 中找出并返回总和为 target 的元素组合的个数。

题目数据保证答案符合 32 位整数范围。

示例 1:

```
输入：nums = [1,2,3], target = 4
输出：7
解释：
所有可能的组合为：
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
请注意，顺序不同的序列被视作不同的组合。
```

解题思路：

完全背包，和上题类似，但是本题求的是排列（组合不强调顺序）

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

题解代码：

```python
class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        dp = [0]*(target + 1)
        dp[0] = 1
        for i in range(1,target + 1):
            for num in nums:
                if i >= num:
                    dp[i] += dp[i-num]
        return dp[-1];
```

279、完全平方数

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

给你一个整数 n ，返回和为 n 的完全平方数的 最少数量 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

```
输入：n = 12
输出：3
解释：12 = 4 + 4 + 4
```

示例 2：

```
输入：n = 13
输出：2
解释：13 = 4 + 9
```

题解代码：

```java
class Solution {
    public int numSquares(int n) {
        int[] f = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            int minn = Integer.MAX_VALUE;
            for (int j = 1; j * j <= i; j++) {
                minn = Math.min(minn, f[i - j * j]);
            }
            f[i] = minn + 1;
        }
        return f[n];
    }
}
```

01背包

$dp[i][j]$表示从下标为[0~i]的物品里任意取，放进容量为j的背包，价值总和最大是多少。

递推公式：$dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]] + value[i])$;

```java
public static void main(String[] args) {
    int[] weight = {1, 3, 4};
    int[] value = {15, 20, 30};
    int bagSize = 4;
    testWeightBagProblem(weight, value, bagSize);
}

public static void testWeightBagProblem(int[] weight, int[] value, int bagSize){
    int wLen = weight.length, value0 = 0;
    //定义dp数组：dp[i][j]表示背包容量为j时，前i个物品能获得的最大价值
    int[][] dp = new int[wLen + 1][bagSize + 1];
    //初始化：背包容量为0时，能获得的价值都为0
    for (int i = 0; i <= wLen; i++){
        dp[i][0] = value0;
    }
    //遍历顺序：先遍历物品，再遍历背包容量
    for (int i = 1; i <= wLen; i++){
        for (int j = 1; j <= bagSize; j++){
            if (j < weight[i - 1]){
                dp[i][j] = dp[i - 1][j];
            }else{
                dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weight[i - 1]] +
value[i - 1]);
            }
        }
    }
    //打印dp数组
    for (int i = 0; i <= wLen; i++){
        for (int j = 0; j <= bagSize; j++){
            System.out.print(dp[i][j] + " ");
        }
        System.out.print("\n");
    }
}
```

343、整数拆分

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

示例 1:

```
输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。
```

示例 2:

```
输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。
```

解题思路:

$dp[i]$表示拆分数字 i 得到的最大乘积

$dp[i] = max\{dp[i], j*(i-j), j*dp[i-j]\}$

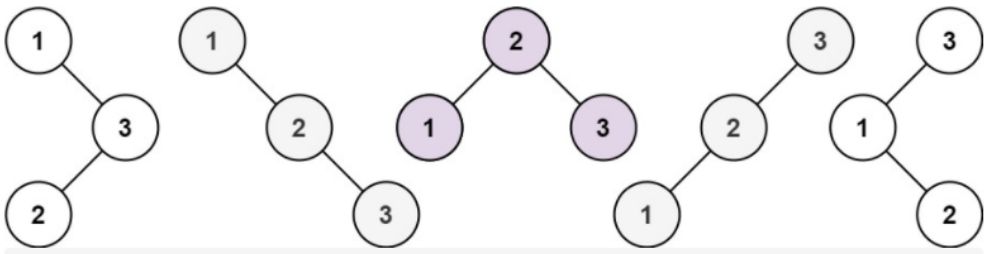
题解代码:

```python
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [0] * (n + 1)
        for i in range(2, n + 1):
            for j in range(i):
                dp[i] = max(dp[i], j * (i - j), j * dp[i - j])
        return dp[n]
```

96、不同的二叉搜索树

给你一个整数 `n` ，求恰由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同的 **二叉搜索树** 有多少种？返回满足题意的二叉搜索树的种数。

示例 1:



输入： n = 3
输出： 5

示例 2:

输入： n = 1
输出： 1

解题思路:

$dp[\ ]$表示1到i为节点组成的二叉搜索树的个数，例如dp[3]，就是 元素1为头结点搜索树的数量 + 元素2为头结点搜索树的数量 + 元素3为头结点搜索树的数量

$dp[i] += dp[以j为头结点左子树节点数量] * dp[以j为头结点右子树节点数量]$

所以递推公式：$dp[i] += dp[j-1] * dp[i-j]$ ，j-1 为j为头结点左子树节点数量，i-j 为以j为头结点右子树节点数量

题解代码：

```
class Solution {
    public int numTrees(int n) {
        int []dp = new int[n+1];
        dp[0] = 1;
        for(int i= 1; i < n+1; ++i){
            for(int j = 1; j <= i; ++j){
                dp[i] += dp[j-1] * dp[i-j];
            }
        }
        return dp[n];
    }
}
```

416、分割等和子集

类似题：1049. 最后一块石头的重量 II

给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
```

示例 2：

```
输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集
```

解题思路：

DFS 或者 动态规划

题解代码：

动态规划

```
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;
        if (n < 2) {
            return false;
        }
        int sum = 0, maxNum = 0;
        for (int num : nums) {
            sum += num;
            maxNum = Math.max(maxNum, num);
        }
        if (sum % 2 != 0) {
            return false;
        }
```

```
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        boolean[] dp = new boolean[target + 1];
        dp[0] = true;
        for (int i = 0; i < n; i++) {
            int num = nums[i];
            for (int j = target; j >= num; --j) {
                dp[j] |= dp[j - num];
            }
        }
        return dp[target];
    }
}
```

DFS

```
class Solution(object):
    def canPartition(self, nums):
        if not nums: return True
        total = sum(nums)
        if total & 1:  # 和为奇数
            return False
        total = total >> 1  # 除2
        nums.sort(reverse=True)  # 逆排序
        if total < nums[0]:  # 当数组最大值超过总和的一半
            return False
        return self.dfs(nums, total)

    def dfs(self, nums, total):
        if total == 0:
            return True
        if total < 0:
            return False
        for i in range(len(nums)):
            if self.dfs(nums[i+1:], total - nums[i]):  # 除去i及其之前，保证每个数只用一次
                return True
        return False
```

474、一和零

给你一个二进制字符串数组 strs 和两个整数 m 和 n 。

请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。

如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

示例 1：

```
输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
输出：4
解释：最多有 5 个 0 和 3 个 1 的最大子集是 {"10","0001","1","0"} ，因此答案是 4 。
其他满足题意但较小的子集包括 {"0001","1"} 和 {"10","1","0"} 。{"111001"} 不满足题意，因为它含 4 个 1 ，大于 n 的值 3 。
```

示例 2：

```
输入：strs = ["10", "0", "1"], m = 1, n = 1
输出：2
解释：最大的子集是 {"0", "1"} ，所以答案是 2 。
```

解题思路：

本题其实是01背包问题！

这不过这个背包有两个维度，一个是m 一个是n，而不同长度的字符串就是不同大小的待装物品。

开始动规五部曲：

1. 确定dp数组（dp table）以及下标的含义

三维定义：定义三维数组 dp，其中 dp[i][j][k] 表示在前 i 个字符串中，使用 j 个 0 和 k 个 1 的情况下最多可以得到的字符串数量。假设数组 str 的长度为 l，则最终答案为 dp[l][m][n]。

二维定义：**dp[i][j]：最多有i个0和j个1的strs的最大子集的大小为dp[i][j]。**

2. 确定递推公式

dp[i][j] 可以由前一个strs里的字符串推导出来，strs里的字符串有zeroNum个0，oneNum个1。

dp[i][j] 就可以是 dp[i - zeroNum][j - oneNum] + 1。

然后我们在遍历的过程中，取dp[i][j]的最大值。

所以递推公式：dp[i][j] = max(dp[i][j], dp[i - zeroNum][j - oneNum] + 1);

此时大家可以回想一下01背包的递推公式：dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

对比一下就会发现，字符串的zeroNum和oneNum相当于物品的重量（weight[i]），字符串本身的个数相当于物品的价值（value[i]）。

这就是一个典型的01背包！ 只不过物品的重量有了两个维度而已。

3. dp数组如何初始化

01背包的dp数组初始化为0就可以。因为物品价值不会是负数，初始为0，保证递推的时候dp[i][j]不会被初始值覆盖。

4. 确定遍历顺序

01背包一定是外层for循环遍历物品，内层for循环遍历背包容量且从后向前遍历！

那么本题也是，物品就是strs里的字符串，背包容量就是题目描述中的m和n。

代码如下：

```
for (String str : strs) { // 遍历物品
    int oneNum = 0, zeroNum = 0;
    for (char c : str) {
        if (c == '0') zeroNum++;
        else oneNum++;
    }
    for (int i = m; i >= zeroNum; i--) { // 遍历背包容量且从后向前遍历！
        for (int j = n; j >= oneNum; j--) {
            dp[i][j] = max(dp[i][j], dp[i - zeroNum][j - oneNum] + 1);
        }
    }
}
```

题解代码：

```
class Solution {
```

```
    public int findMaxForm(String[] strs, int m, int n) {
        //dp[i][j]表示i个0和j个1时的最大子集
        int[][] dp = new int[m + 1][n + 1];
        int oneNum, zeroNum;
        for (String str : strs) {
            oneNum = 0;
            zeroNum = 0;
            for (char ch : str.toCharArray()) {
                if (ch == '0') {
                    zeroNum++;
                } else {
                    oneNum++;
                }
            }
            //倒序遍历
            for (int i = m; i >= zeroNum; i--) {
                for (int j = n; j >= oneNum; j--) {
                    dp[i][j] = Math.max(dp[i][j], dp[i - zeroNum][j - oneNum] +
1);
                }
            }
        }
        return dp[m][n];
    }
}
```

139、单词拆分

给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

拆分时可以重复使用字典中的单词。
你可以假设字典中没有重复的单词。
示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

解题思路：

- 回溯

使用记忆化函数，保存出现过的 backtrack(s)，避免重复计算。

定义回溯函数 backtrack(s)

若 s 长度为 0，则返回 True，表示已经使用 wordDict 中的单词分割完。

初试化当前字符串是否可以被分割 res=False

遍历结束索引 i，遍历区间 [1,n+1)：

若 s[0,⋯,i−1] 在 wordDict 中：res=backtrack(s[i,⋯,n−1]) or res。解释：保存遍历结束索引中，可以使字符串切割完成的情况。

返回 res

返回 backtrack(s)

- 动态规划

单词就是物品，字符串s就是背包，单词能否组成字符串s，就是问物品能不能把背包装满。

拆分时可以重复使用字典中的单词，说明就是一个完全背包！

动规五部曲分析如下：

1. 确定dp数组以及下标的含义

dp[i]：字符串长度为i的话，dp[i]为true，表示可以拆分为一个或多个在字典中出现的单词。

2. 确定递推公式

如果确定dp[j] 是true，且 [j, i] 这个区间的子串出现在字典里，那么dp[i]一定是true。（j < i ）。

所以递推公式是 if([j, i] 这个区间的子串出现在字典里 && dp[j]是true) 那么 dp[i] = true。

3. dp数组如何初始化

从递归公式中可以看出，dp[i] 的状态依靠 dp[j]是否为true，那么dp[0]就是递归的根基，dp[0]一定要为true，否则递归下去后面都都是false了。

那么dp[0]有没有意义呢?

dp[0]表示如果字符串为空的话，说明出现在字典里。

但题目中说了“给定一个非空字符串 s” 所以测试数据中不会出现i为0的情况，那么dp[0]初始为true完全就是为了推导公式。

下标非0的dp[i]初始化为false，只要没有被覆盖说明都是不可拆分为一个或多个在字典中出现的单词。

题解代码：

```java
//动态规划
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        boolean[] valid = new boolean[s.length() + 1];
        valid[0] = true;
        for(int i = 1; i <= s.length(); ++i){
            for(int j = 0; j < i; ++j){
                if(wordDict.contains(s.substring(j,i)) && valid[j]){
                    valid[i] = true;
                }
            }
        }
        return valid[s.length()];
    }
}
```

```java
//记忆化回溯
class Solution {
    Set<String> memory = new HashSet<>();
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> set = new HashSet<>();
        for(String str : wordDict){
            set.add(str);
        }
        return DFS(s,set);
    }
    public boolean DFS(String s,Set<String> set){
        if(s.length()==0) return true;
        if(memory.contains(s)) return false;//如果记忆中存在此字符串，返回false，结束递归。
        StringBuilder strb = new StringBuilder();
        for(int i=0;i<s.length();i++){
```

```
            strb.append(s.charAt(i));
            if(set.contains(strb.toString()) &&
!memory.contains(s.substring(i+1))){
                if(DFS(s.substring(i+1),set)){
                    return true;
                }else{
                    memory.add(s.substring(i+1));//对子串失败的情况进行记忆
                }
            }
        }
        memory.add(s);//对s失败的情况进行记忆
        return false;
    }
}
```

5815、扣分后的最大得分

给你一个 m x n 的整数矩阵 points （下标从 0 开始）。一开始你的得分为 0，你想最大化从矩阵中得到的分数。

你的得分方式为：每一行 中选取一个格子，选中坐标为 (r, c) 的格子会给你的总得分 增加 points[r][c] 。

然而，相邻行之间被选中的格子如果隔得太远，你会失去一些得分。对于相邻行 r 和 r + 1 （其中 0 <= r < m - 1），选中坐标为 (r, c1) 和 (r + 1, c2) 的格子，你的总得分 减少 abs(c1 - c2) 。

请你返回你能得到的 最大 得分。

abs(x) 定义为：

如果 x >= 0 ，那么值为 x 。
如果 x < 0 ，那么值为 -x 。

实例1：

```
输入：points = [[1,2,3],[1,5,1],[3,1,1]]
输出：9
解释：
蓝色格子是最优方案选中的格子，坐标分别为 (0, 2)，(1, 1) 和 (2, 0) 。
你的总得分增加 3 + 5 + 3 = 11 。
但是你的总得分需要扣除 abs(2 - 1) + abs(1 - 0) = 2 。
你的最终得分为 11 - 2 = 9 。
```

解题思路：

暴力：最普通的方法我们可以想到，$dp[i][j]$表示第i行第j列的元素所能得到的最大分数。

那么对于第i行，我们遍历每个j，针对i-1行的每一个元素计算他们的新得分，就能得到当前元素对应的最高得分：

$$dp[i][j] = max(dp[i-1][k] + abs(k-j))$$

时间复杂度为$O(mn^2)$。

优化：对于当前行的j元素，我们从左到右计算在它左方和上方的最大值 $lmax$。每次右移，$lmax-1$，于是j元素对应的左边的最大值为 $max(lmax-1, dp[j])$。同理，右边的最大值为 $max(rmax-1, dp[j])$。

取 $lmax, rmax, dp[i-1][j]$的最大值作为当前j元素之前的最大得分，那么再加上当前得分就是$dp[i][j]$对应的最大的分：

$$dp[i][j] = max(lmax, rmax, dp[i-1][j]) + points[i][j]$$

这样把查找的时间从$O(n^2)$降到了$O(n)$, 总的时间复杂度为$O(mn)$。

题解代码：

```java
class Solution {
    public long maxPoints(int[][] points) {
        int m = points.length;
        int n = points[0].length;
        long[] dp = new long[n];
        for (int i = 0; i < m; i++) {
            long[] cur = new long[n + 1];
            long lmax = 0;
            for (int j = 0; j < n; j++) {
                lmax = Math.max(lmax - 1, dp[j]);
                cur[j] = lmax;
            }
            long rmax = 0;
            for (int j = n - 1; j >= 0; j--) {
                rmax = Math.max(rmax - 1, dp[j]);
                cur[j] = Math.max(cur[j], rmax);
            }
            for (int j = 0; j < n; j++) {
                dp[j] = cur[j] + points[i][j];
            }
        }
        long ans = 0;
        for (int j = 0; j < n; j++) {
            ans = Math.max(ans, dp[j]);
        }
        return ans;
    }
}
```

1035、不相交的线

在两条独立的水平线上按给定的顺序写下 nums1 和 nums2 中的整数。

现在，可以绘制一些连接两个数字 nums1[i] 和 nums2[j] 的直线，这些直线需要同时满足满足：

nums1[i] == nums2[j]
且绘制的直线不与任何其他连线（非水平线）相交。
请注意，连线即使在端点也不能相交：每个数字只能属于一条连线。

以这种方法绘制线条，并返回可以绘制的最大连线数。

示例 1：


```
输入：nums1 = [1,4,2], nums2 = [1,2,4]
输出：2
解释：可以画出两条不交叉的线，如上图所示。
但无法画出第三条不相交的直线，因为从 nums1[1]=4 到 nums2[2]=4 的直线将与从 nums1[2]=2 到 nums2[1]=2 的直线相交。
```

解题思路：

思路和最长公共子序列一样，动态规划。

状态转移：

```
if (text1[i - 1] == text2[j - 1]) {
    dp[i][j] = dp[i - 1][j - 1] + 1;
} else {
    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
}
```

题解代码：

```
class Solution {
    public int maxUncrossedLines(int[] nums1, int[] nums2) {
        int m = nums1.length,n = nums2.length;
        int [][]dp = new int [m+1][n+1];
        for(int i = 1; i <= m; ++i){
            int num1 = nums1[i - 1];
            for(int j = 1; j <= n; ++j){
                int num2 = nums2[j - 1];
                if(num1 == num2){
                    dp[i][j] = dp[i-1][j-1] + 1;
                }else{
                    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
                }
            }
        }
        return dp[m][n];
    }
}
```

115、不同的子序列

给定一个字符串 s 和一个字符串 t ，计算在 s 的子序列中 t 出现的个数。

字符串的一个 子序列 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，"ACE" 是 "ABCDE" 的一个子序列，而 "AEC" 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

```
输入：s = "rabbbit", t = "rabbit"
输出：3
解释：
如下图所示，有 3 种可以从 s 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)
rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
```

解题思路：

动态规划

dp[i][j]：以i-1为结尾的s子序列中出现以j-1为结尾的t的个数为dp[i][j]。

这一类问题，基本是要分析两种情况

- s[i - 1] 与 t[j - 1]相等
- s[i - 1] 与 t[j - 1] 不相等

当s[i - 1] 与 t[j - 1]相等时，dp[i][j]可以有两部分组成。

一部分是用s[i - 1]来匹配，那么个数为dp[i - 1][j - 1]。

一部分是不用s[i - 1]来匹配，个数为dp[i - 1][j]。

这里可能有同学不明白了，为什么还要考虑 不用s[i - 1]来匹配，都相同了指定要匹配啊。

例如： s：bagg 和 t：bag ，s[3] 和 t[2]是相同的，但是字符串s也可以不用s[3]来匹配，即用s[0]s[1]s[2]组成的bag。

当然也可以用s[3]来匹配，即：s[0]s[1]s[3]组成的bag。

所以当s[i - 1] 与 t[j - 1]相等时，dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];

当s[i - 1] 与 t[j - 1]不相等时，dp[i][j]只有一部分组成，不用s[i - 1]来匹配，即：dp[i - 1][j]

所以递推公式为：dp[i][j] = dp[i - 1][j];

题解代码：

```
class Solution {
    public int numDistinct(String s, String t) {
        int [][]dp = new int[s.length() + 1][t.length() + 1];
        for(int i = 0; i < s.length() + 1; ++i){
            dp[i][0] = 1;
        }

        for(int i = 1; i < s.length() + 1; ++i){
            for(int j = 1; j < t.length() + 1; ++j){
                if(s.charAt(i-1) == t.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[s.length()][t.length()];
    }
}
```

72、编辑距离

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

示例 1：

```
输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

解题思路：

动态规划

数组定义：dp[i][j] 表示以下标i-1为结尾的字符串word1，和以下标j-1为结尾的字符串word2，最近编辑距离为dp[i][j]。

递推公式：

```
if (word1[i - 1] == word2[j - 1])
    不操作
if (word1[i - 1] != word2[j - 1])
    增
    删
    换

if (word1[i - 1] == word2[j - 1]) {
    dp[i][j] = dp[i - 1][j - 1];
}
else {
    dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
}
```

遍历顺序：从左到右，从上到下

题解代码：

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length(), n = word2.length();
        int [][]dp = new int[m+1][n+1];
        for(int i = 0; i < m+1; ++i) dp[i][0] = i;
        for(int j = 0; j < n+1; ++j) dp[0][j] = j;

        for(int i = 1; i <= m; ++i){
            for(int j = 1; j <= n;++j){
                if(word1.charAt(i-1) == word2.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1];
                }else{
                    dp[i][j] = Math.min(Math.min(dp[i-1][j-1],dp[i-1][j]),dp[i][j-
1]) + 1;
                }
            }
        }
        return dp[m][n];
    }
}
```

516、最长回文子序列

给你一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

示例 1：

```
输入：s = "bbbab"
输出：4
解释：一个可能的最长回文子序列为 "bbbb" 。
```

示例 2：

```
输入：s = "cbbd"
输出：2
解释：一个可能的最长回文子序列为 "bb" 。
```

解题思路：

动态规划

数组定义：dp[i][j]：字符串s在[i, j]范围内最长的回文子序列的长度为dp[i][j]。

递推公式：

```
if (s[i] == s[j]) {
    dp[i][j] = dp[i + 1][j - 1] + 2;
} else {
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
}
```

遍历顺序：从下到上

题解代码：

```
public class Solution {
    public int longestPalindromeSubseq(String s) {
        int len = s.length();
        int[][] dp = new int[len + 1][len + 1];
        for (int i = len - 1; i >= 0; i--) { // 从后往前遍历 保证情况不漏
            dp[i][i] = 1; // 初始化
            for (int j = i + 1; j < len; j++) {
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                } else {
                    dp[i][j] = Math.max(dp[i + 1][j], Math.max(dp[i][j], dp[i][j - 1]));
                }
            }
        }
        return dp[0][len - 1];
    }
}
```

300、最长递增子序列

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

示例 1：

```
输入：nums = [10,9,2,5,3,7,101,18]
输出：4
解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
```

示例 2：

```
输入：nums = [0,1,0,3,2,3]
输出：4
```

解题思路：

- 动态规划

  定义 dp[i] 为前 i 个元素，以第 i 个数字结尾的最长上升子序列。

  $d[i] = max(dp[j]) + 1$，其 中$0 <= j < i$且$num[j] < num[i]$

- 贪心和二分查找

  维护一个单调数组 d[i]，表示长度为 i 的最长上升子序列的末尾元素的最小值，用 len 记录目前最长上升子序列的长度，起始时 len 为 1，d[1]=nums[0]。

  我们依次遍历数组 nums 中的每个元素，并更新数组 d 和 len 的值。如果 nums[i]>d[len] 则更新 len=len+1，否则在 d[1...len]中找满足 d[i−1]<nums[j]<d[i] 的下标 ii，并更新 d[i]=nums[j]。

  根据 d 数组的单调性，我们可以使用二分查找寻找下标 i，优化时间复杂度。

题解代码：

```java
//动态规划
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int maxans = 1;
        for (int i = 1; i < nums.length; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxans = Math.max(maxans, dp[i]);
        }
        return maxans;
    }
}

//贪心 + 二分查找
class Solution {
    public int lengthOfLIS(int[] nums) {
        int len = 1, n = nums.length;
        if (n == 0) {
            return 0;
        }
        int[] d = new int[n + 1];
        d[len] = nums[0];
        for (int i = 1; i < n; ++i) {
            if (nums[i] > d[len]) {
                d[++len] = nums[i];
            } else {
                int l = 1, r = len, pos = 0; // 如果找不到说明所有的数都比 nums[i] 大，此时要更新 d[1]，所以这里将 pos 设为 0
                while (l <= r) {
                    int mid = (l + r) >> 1;
                    if (d[mid] < nums[i]) {
                        pos = mid;
                        l = mid + 1;
                    } else {
                        r = mid - 1;
```

```
                    }
                }
                d[pos + 1] = nums[i];
            }
        }
        return len;
    }
}
```

单调栈

739、每日温度

请根据每日 气温 列表 temperatures ，请计算在每一天需要等几天才会有更高的温度。如果气温在这之后都不会升高，请在该位置用 0 来代替。

示例 1:

```
输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]
```

示例 2:

```
输入: temperatures = [30,40,50,60]
输出: [1,1,1,0]
```

解题思路:

单调栈，确定是从大到小还是从小到大

- 当前遍历的元素T[i]小于栈顶元素T[st.top()]的情况
- 当前遍历的元素T[i]等于栈顶元素T[st.top()]的情况
- 当前遍历的元素T[i]大于栈顶元素T[st.top()]的情况

题解代码:

```
class Solution {
    public int[] dailyTemperatures(int[] temperatures) {
        Stack<Integer> stack = new Stack<>();
        int []res= new int[temperatures.length];
        for(int i = 0; i < temperatures.length; ++i){
            while(!stack.isEmpty() && temperatures[i] >
temperatures[stack.peek()]){
                int pre = stack.pop();
                res[pre] = i - pre;
            }
            stack.push(i);
        }
        return res;
    }
}
```

496、下一个更大元素I

给你两个 没有重复元素 的数组 nums1 和 nums2 ，其中nums1 是 nums2 的子集。

请你找出 nums1 中每个元素在 nums2 中的下一个比其大的值。

nums1 中数字 x 的下一个更大元素是指 x 在 nums2 中对应位置的右边的第一个比 x 大的元素。如果不存在，对应位置输出 -1 。

示例 1:

```
输入: nums1 = [4,1,2], nums2 = [1,3,4,2].
输出: [-1,3,-1]
解释:
    对于 num1 中的数字 4 ，你无法在第二个数组中找到下一个更大的数字，因此输出 -1 。
    对于 num1 中的数字 1 ，第二个数组中数字1右边的下一个较大数字是 3 。
    对于 num1 中的数字 2 ，第二个数组中没有下一个更大的数字，因此输出 -1 。
```

解题思路:

单调栈和hash表

题解代码:

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Stack<Integer> stack = new Stack<>();
        Map<Integer,Integer> map = new HashMap<>();
        for(int i = 0; i < nums2.length; ++i){
            while(!stack.isEmpty() && nums2[i] > stack.peek()){
                map.put(stack.pop(),nums2[i]);
            }
            stack.push(nums2[i]);
        }
        int []res = new int[nums1.length];
        for(int i = 0; i < nums1.length; ++i){
            res[i] = map.getOrDefault(nums1[i],-1);
        }
        return res;
    }
}
```

异或操作

260、只出现一次的数字III

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。你可以按 任意顺序 返回答案。

进阶：你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现?

示例 1:

```
输入: nums = [1,2,1,3,2,5]
输出: [3,5]
解释: [5, 3] 也是有效的答案。
```

示例 2:

```
输入: nums = [-1,0]
输出: [-1,0]
```

解题思路:

先对所有数字进行一次异或，得到两个出现一次的数字的异或值。

在异或结果中找到任意为 1 的位。

根据这一位对所有的数字进行分组。

在每个组内进行异或操作，得到两个数字。

题解代码：

```java
class Solution {
    public int[] singleNumber(int[] nums) {
        int ret = 0;
        for (int n : nums) {
            ret ^= n;
        }
        int div = 1;
        while ((div & ret) == 0) {
            div <<= 1;
        }
        int a = 0, b = 0;
        for (int n : nums) {
            if ((div & n) != 0) {
                a ^= n;
            } else {
                b ^= n;
            }
        }
        return new int[]{a, b};
    }
}
```

268、丢失的数字

给定一个包含 [0, n] 中 n 个数的数组 nums ， 找出 [0, n] 这个范围内没有出现在数组中的那个数。

进阶：

你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题?

示例 1：

```
输入：nums = [3,0,1]
输出：2
解释：n = 3，因为有 3 个数字，所以所有的数字都在范围 [0,3] 内。2 是丢失的数字，因为它没有出现在 nums 中。
```

解题思路：

索引和值进行异或操作

题解代码：

```java
class Solution {
    public int missingNumber(int[] nums) {
        int missing = nums.length;
        for (int i = 0; i < nums.length; i++) {
            missing ^= i ^ nums[i];
        }
        return missing;
    }
}
```

KMP

最长公共前后缀求解代码

```
void getNext(int []next, String s){
    int j = -1;
    next[0] = j;
    for(int i = 1; i < s.length(); ++i){
        while(j >= 0 && s.charAt(i) != s.charAt(j+1)){
            j = next[j];
        }
        if(s.charAt(i) == s.charAt(j+1)){
            j++;
        }
        next[i] = j;
    }
}
```

459、重复的子字符串

给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。

```
示例 1:

输入: "abab"

输出: True

解释: 可由子字符串 "ab" 重复两次构成。
```

解题思路:

- 暴力 枚举前缀
- KMP的next数组 **数组长度减去最长相同前后缀的长度相当于是第一个周期的长度，也就是一个周期的长度，如果这个周期可以被整除，就说明整个数组就是这个周期的循环。**

题解代码:

```
class Solution {
    public boolean repeatedSubstringPattern(String s) {
        if (s.equals("")) return false;

        int len = s.length();
        // 原串加个空格(哨兵)，使下标从1开始，这样j从0开始，也不用初始化了
        s = " " + s;
        char[] chars = s.toCharArray();
        int[] next = new int[len + 1];

        // 构造 next 数组过程，j从0开始(空格)，i从2开始
        for (int i = 2, j = 0; i <= len; i++) {
            // 匹配不成功，j回到前一位置 next 数组所对应的值
            while (j > 0 && chars[i] != chars[j + 1]) j = next[j];
            // 匹配成功，j往后移
            if (chars[i] == chars[j + 1]) j++;
            // 更新 next 数组的值
            next[i] = j;
        }

        // 最后判断是否是重复的子字符串，这里 next[len] 即代表next数组末尾的值
```

```
        if (next[len] > 0 && len % (len - next[len]) == 0) {
            return true;
        }
        return false;
    }
}
```