LeetCode Medium & Hard Level

1. Two Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have *exactly* one solution, and you may not use the *same* element twice.

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

这道 Two Sum 的题目作为 LeetCode 的开篇之题，乃是经典中的经典，正所谓‘**平生不识 TwoSum，刷尽 LeetCode 也枉然**’，就像英语单词书的第一个单词总是 Abandon 一样，很多没有毅力坚持的人就只能记住这一个单词，所以通常情况下单词书就前几页有翻动的痕迹，后面都是崭新如初，道理不需多讲，鸡汤不必多灌，明白的人自然明白。

这道题给了我们一个数组，还有一个目标数target，让找到两个数字，使其和为 target，乍一看就感觉可以用暴力搜索，但是猜到 OJ 肯定不会允许用暴力搜索这么简单的方法，于是去试了一下，果然是 Time Limit Exceeded，这个算法的时间复杂度是 O(n^2)。那么只能想个 O(n) 的算法来实现，由于暴力搜索的方法是遍历所有的两个数字的组合，然后算其和，这样虽然节省了空间，但是时间复杂度高。一般来说，为了提高时间的复杂度，需要用空间来换，这算是一个 trade off 吧，但这里只想用线性的时间复杂度来解决问题，就是说只能遍历一个数字，那么另一个数字呢，可以事先将其存储起来，使用一个 HashMap，来建立数字和其坐标位置之间的映射，由于 HashMap 是常数级的查找效率，这样在遍历数组的时候，用 target 减去遍历到的数字，就是另一个需要的数字了，直接在 HashMap 中查找其是否存在即可，注意要判断查找到的数字不是第一个数字，比如 target 是4，遍历到了一个2，那么另外一个2不能是之前那个2，整个实现步骤为：先遍历一遍数组，建立 HashMap 映射，然后再遍历一遍，开始查找，找到则记录 index。代码如下：

C++ 解法一：

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> m;
        vector<int> res;
        for (int i = 0; i < nums.size(); ++i) {
            m[nums[i]] = i;
        }
        for (int i = 0; i < nums.size(); ++i) {
            int t = target - nums[i];
            if (m.count(t) && m[t] != i) {
```

```
                res.push_back(i);
                res.push_back(m[t]);
                break;
            }
        }
        return res;
    }
};
```

Java解法一：

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> m = new HashMap<Integer, Integer>();
        int[] res = new int[2];
        for (int i = 0; i < nums.length; ++i) {
            m.put(nums[i], i);
        }
        for (int i = 0; i < nums.length; ++i) {
            int t = target - nums[i];
            if (m.containsKey(t) && m.get(t) != i) {
                res[0] = i;
                res[1] = m.get(t);
                break;
            }
        }
        return res;
    }
}
```

或者可以写的更加简洁一些，把两个 for 循环合并成一个：

C++解法二：

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> m;
        for (int i = 0; i < nums.size(); ++i) {
            if (m.count(target - nums[i])) {
                return {i, m[target - nums[i]]};
            }
            m[nums[i]] = i;
        }
        return {};
    }
};
```

Java解法二：

```java
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> m = new HashMap<Integer, Integer>();
        int[] res = new int[2];
        for (int i = 0; i < nums.length; ++i) {
            if (m.containsKey(target - nums[i])) {
                res[0] = i;
                res[1] = m.get(target - nums[i]);
                break;
            }
            m.put(nums[i], i);
        }
        return res;
    }
}
```

2. Add Two Numbers

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example:

```
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807.
```

这道并不是什么难题，算法很简单，链表的数据类型也不难，就是建立一个新链表，然后把输入的两个链表从头往后撸，每两个相加，添加一个新节点到新链表后面。为了避免两个输入链表同时为空，我们建立一个 dummy 结点，将两个结点相加生成的新结点按顺序加到 dummy 结点之后，由于 dummy 结点本身不能变，所以用一个指针 cur 来指向新链表的最后一个结点。好，可以开始让两个链表相加了，这道题好就好在最低位在链表的开头，所以可以在遍历链表的同时按从低到高的顺序直接相加。while 循环的条件两个链表中只要有一个不为空行，由于链表可能为空，所以在取当前结点值的时候，先判断一下，若为空则取0，否则取结点值。然后把两个结点值相加，同时还要加上进位 carry。然后更新 carry，直接 sum/10 即可，然后以 sum%10 为值建立一个新结点，连到 cur 后面，然后 cur 移动到下一个结点。之后再更新两个结点，若存在，则指向下一个位置。while 循环退出之后，最高位的进位问题要最后特殊处理一下，若 carry 为1，则再建一个值为1的结点，代码如下：

C++ 解法：

```cpp
class Solution {
public:
```

```
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1), *cur = dummy;
        int carry = 0;
        while (l1 || l2) {
            int val1 = l1 ? l1->val : 0;
            int val2 = l2 ? l2->val : 0;
            int sum = val1 + val2 + carry;
            carry = sum / 10;
            cur->next = new ListNode(sum % 10);
            cur = cur->next;
            if (l1) l1 = l1->next;
            if (l2) l2 = l2->next;
        }
        if (carry) cur->next = new ListNode(1);
        return dummy->next;
    }
};
```

Java 解法

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode cur = dummy;
        int carry = 0;
        while (l1 != null || l2 != null) {
            int d1 = l1 == null ? 0 : l1.val;
            int d2 = l2 == null ? 0 : l2.val;
            int sum = d1 + d2 + carry;
            carry = sum >= 10 ? 1 : 0;
            cur.next = new ListNode(sum % 10);
            cur = cur.next;
            if (l1 != null) l1 = l1.next;
            if (l2 != null) l2 = l2.next;
        }
        if (carry == 1) cur.next = new ListNode(1);
        return dummy.next;
    }
}
```

4. Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters.

Example 1:

```
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Note that the answer must be a substring, "pwke" is a _subsequence_ and not
a substring.
```

这道求最长无重复子串的题和之前那道 Isomorphic Strings 很类似，属于 LeetCode 早期经典题目，博主认为是可以跟 Two Sum 媲美的一道题。给了我们一个字符串，让求最长的无重复字符的子串，注意这里是子串，不是子序列，所以必须是连续的。先不考虑代码怎么实现，如果给一个例子中的例子 "abcabcbb"，让你手动找无重复字符的子串，该怎么找。博主会一个字符一个字符的遍历，比如 a，b，c，然后又出现了一个a，那么此时就应该去掉第一次出现的a，然后继续往后，又出现了一个b，则应该去掉一次出现的b，以此类推，最终发现最长的长度为3。所以说，需要记录之前出现过的字符，记录的方式有很多，最常见的是统计字符出现的个数，但是这道题字符出现的位置很重要，所以可以使用 HashMap 来建立字符和其出现位置之间的映射。进一步考虑，由于字符会重复出现，到底是保存所有出现的位置呢，还是只记录一个位置？我们之前手动推导的方法实际上是维护了一个滑动窗口，窗口内的都是没有重复的字符，需要尽可能的扩大窗口的大小。由于窗口在不停向右滑动，所以只关心每个字符最后出现的位置，并建立映射。窗口的右边界就是当前遍历到的字符的位置，为了求出窗口的大小，需要一个变量 left 来指向滑动窗口的左边界，这样，如果当前遍历到的字符从未出现过，那么直接扩大右边界，如果之前出现过，那么就分两种情况，在或不在滑动窗口内，如果不在滑动窗口内，那么就没事，当前字符可以加进来，如果在的话，就需要先在滑动窗口内去掉这个已经出现过的字符了，去掉的方法并不需要将左边界 left 一位一位向右遍历查找，由于 HashMap 已经保存了该重复字符最后出现的位置，所以直接移动 left 指针就可以了。维护一个结果 res，每次用出现过的窗口大小来更新结果 res，就可以得到最终结果啦。

这里可以建立一个 HashMap，建立每个字符和其最后出现位置之间的映射，然后需要定义两个变量 res 和 left，其中 res 用来记录最长无重复子串的长度，left 指向该无重复子串左边的起始位置的前一个，由于是前一个，所以初始化就是 -1，然后遍历整个字符串，对于每一个遍历到的字符，如果该字符已经在 HashMap 中存在了，并且如果其映射值大于 left 的话，那么更新 left 为当前映射值。然后映射值更新为当前坐标i，这样保证了 left 始终为当前边界的前一个位置，然后计算窗口长度的时候，直接用 i-left 即可，用来更新结果 res。

这里解释下程序中那个 if 条件语句中的两个条件 m.count(s[i]) && m[s[i]] > left，因为一旦当前字符 s[i] 在 HashMap 已经存在映射，说明当前的字符已经出现过了，而若 m[s[i]] > left 成立，说明之前出现过的字符在窗口内，那么如果要加上当前这个重复的字符，就要移除之前的那个，所以让 left 赋值为 m[s[i]]，由于 left 是窗口左边界的前一个位置（这也是 left 初始化为 -1 的原因，因为窗口左边界是从0

开始遍历的），所以相当于已经移除出滑动窗口了。举一个最简单的例子 "aa"，当 i=0 时，建立了 a->0 的映射，并且此时结果 res 更新为1，那么当 i=1 的时候，发现a在 HashMap 中，并且映射值0大于 left 的 -1，所以此时 left 更新为0，映射对更新为 a->1，那么此时 i-left 还为1，不用更新结果 res，那么最终结果 res 还为1，正确，代码如下：

C++ 解法一：

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int res = 0, left = -1, n = s.size();
        unordered_map<int, int> m;
        for (int i = 0; i < n; ++i) {
            if (m.count(s[i]) && m[s[i]] > left) {
                left = m[s[i]];
            }
            m[s[i]] = i;
            res = max(res, i - left);
        }
        return res;
    }
};
```

下面这种写法是上面解法的精简模式，这里我们可以建立一个 256 位大小的整型数组来代替 HashMap，这样做的原因是 ASCII 表共能表示 256 个字符，但是由于键盘只能表示 128 个字符，所以用 128 也行，然后全部初始化为 -1，这样的好处是不用像之前的 HashMap 一样要查找当前字符是否存在映射对了，对于每一个遍历到的字符，直接用其在数组中的值来更新 left，因为默认是 -1，而 left 初始化也是 -1，所以并不会产生错误，这样就省了 if 判断的步骤，其余思路都一样：

C++ 解法二：

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> m(128, -1);
        int res = 0, left = -1;
        for (int i = 0; i < s.size(); ++i) {
            left = max(left, m[s[i]]);
            m[s[i]] = i;
            res = max(res, i - left);
        }
        return res;
    }
};
```

Java 解法二：

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int[] m = new int[256];
        Arrays.fill(m, -1);
        int res = 0, left = -1;
        for (int i = 0; i < s.length(); ++i) {
            left = Math.max(left, m[s.charAt(i)]);
            m[s.charAt(i)] = i;
            res = Math.max(res, i - left);
        }
        return res;
    }
}
```

下面这种解法使用了 HashSet，核心算法和上面的很类似，把出现过的字符都放入 HashSet 中，遇到 HashSet 中没有的字符就加入 HashSet 中并更新结果 res，如果遇到重复的，则从左边开始删字符，直到删到重复的字符停止：

C++ 解法三：

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int res = 0, left = 0, i = 0, n = s.size();
        unordered_set<char> t;
        while (i < n) {
            if (!t.count(s[i])) {
                t.insert(s[i++]);
                res = max(res, (int)t.size());
            }  else {
                t.erase(s[left++]);
            }
        }
        return res;
    }
};
```

Java解法三：

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int res = 0, left = 0, right = 0;
        HashSet<Character> t = new HashSet<Character>();
        while (right < s.length()) {
            if (!t.contains(s.charAt(right))) {
                t.add(s.charAt(right++));
                res = Math.max(res, t.size());
            } else {
```

```
                t.remove(s.charAt(left++));
            }
        }
        return res;
    }
}
```

4. Median of Two Sorted Arrays

There are two sorted arrays nums1 and nums2 of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

You may assume nums1 and nums2 cannot be both empty.

Example 1:

```
nums1 = [1, 3]
nums2 = [2]

The median is 2.0
```

Example 2:

```
nums1 = [1, 2]
nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5
```

这道题让我们求两个有序数组的中位数，而且限制了时间复杂度为 O(log (m+n))，看到这个时间复杂度，自然而然的想到了应该使用二分查找法来求解。但是这道题被定义为 Hard 也是有其原因的，难就难在要在两个未合并的有序数组之间使用二分法，如果这道题只有一个有序数组，让求中位数的话，估计就是个 Easy 题。对于这道题来说，可以将两个有序数组混合起来成为一个有序数组再做吗，图样图森破，这个时间复杂度限制的就是告诉你金坷垃别想啦。还是要用二分法，而且是在两个数组之间使用，感觉很高端啊。回顾一下中位数的定义，如果某个有序数组长度是奇数，那么其中位数就是最中间那个，如果是偶数，那么就是最中间两个数字的平均值。这里对于两个有序数组也是一样的，假设两个有序数组的长度分别为m和n，由于两个数组长度之和 m+n 的奇偶不确定，因此需要分情况来讨论，对于奇数的情况，直接找到最中间的数即可，偶数的话需要求最中间两个数的平均值。为了简化代码，不分情况讨论，使用一个小 trick，分别找第 (m+n+1) / 2 个，和 (m+n+2) / 2 个，然后求其平均值即可，这对奇偶数均适用。若 m+n 为奇数的话，那么其实 (m+n+1) / 2 和 (m+n+2) / 2 的值相等，相当于两个相同的数字相加再除以2，还是其本身。

好，这里需要定义一个函数来在两个有序数组中找到第K个元素，下面重点来看如何实现找到第K个元素。首先，为了避免拷贝产生新的数组从而增加时间复杂度，使用两个变量i和j分别来标记数组 nums1 和 nums2 的起始位置。然后来处理一些 corner cases，比如当某一个数组的起始位置大于等于其数组长度时，说明其所有数字均已经被淘汰了，相当于一个空数组了，那么实际上就变成了在另一个数组中找数字，直接就可以找出来了。还有就是如果 K=1 的话，只要比较 nums1 和 nums2 的起始位置i和j上

的数字就可以了。难点就在于一般的情况怎么处理？因为需要在两个有序数组中找到第K个元素，为了加快搜索的速度，可以使用二分法，那么对谁二分呢，数组么？其实要对K二分，意思是需要分别在 nums1 和 nums2 中查找第 K/2 个元素，注意这里由于两个数组的长度不定，所以有可能某个数组没有第 K/2 个数字，所以需要先 check 一下，数组中到底存不存在第 K/2 个数字，如果存在就取出来，否则就赋值上一个整型最大值（目的是要在 nums1 或者 nums2 中先淘汰 K/2 个较小的数字，判断的依据就是看 midVal1 和 midVal2 谁更小，但如果某个数组的个数都不到 K/2 个，自然无法淘汰，所以将其对应的 midVal 值设为整型最大值，以保证其不会被淘汰），若某个数组没有第 K/2 个数字，则淘汰另一个数组的前 K/2 个数字即可。举个例子来说吧，比如 nums1 = {3}，nums2 = {2, 4, 5, 6, 7}，K=4，要找两个数组混合中第4个数字，则分别在 nums1 和 nums2 中找第2个数字，而 nums1 中只有一个数字，不存在第二个数字，则 nums2 中的前2个数字可以直接跳过，为啥呢，因为要求的是整个混合数组的第4个数字，不管 nums1 中的那个数字是大是小，第4个数字绝不会出现在 nums2 的前两个数字中，所以可以直接跳过。

有没有可能两个数组都不存在第 K/2 个数字呢，这道题里是不可能的，因为K不是任意给的，而是给的 m+n 的中间值，所以必定至少会有一个数组是存在第 K/2 个数字的。最后就是二分法的核心啦，比较这两个数组的第 K/2 小的数字 midVal1 和 midVal2 的大小，如果第一个数组的第 K/2 个数字小的话，那么说明要找的数字肯定不在 nums1 中的前 K/2 个数字，可以将其淘汰，将 nums1 的起始位置向后移动 K/2 个，并且此时的K也自减去 K/2，调用递归，举个例子来说吧，比如 nums1 = {1, 3}，nums2 = {2, 4, 5}，K=4，要找两个数组混合中第4个数字，那么分别在 nums1 和 nums2 中找第2个数字，nums1 中的第2个数字是3，nums2 中的第2个数字是4，由于3小于4，所以混合数组中第4个数字肯定在 nums2 中，可以将 nums1 的起始位置向后移动 K/2 个。反之，淘汰 nums2 中的前 K/2 个数字，并将 nums2 的起始位置向后移动 K/2 个，并且此时的K也自减去 K/2，调用递归即可，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int m = nums1.size(), n = nums2.size(), left = (m + n + 1) / 2, right = (m + n + 2) / 2;
        return (findKth(nums1, 0, nums2, 0, left) + findKth(nums1, 0, nums2, 0, right)) / 2.0;
    }
    int findKth(vector<int>& nums1, int i, vector<int>& nums2, int j, int k) {
        if (i >= nums1.size()) return nums2[j + k - 1];
        if (j >= nums2.size()) return nums1[i + k - 1];
        if (k == 1) return min(nums1[i], nums2[j]);
        int midVal1 = (i + k / 2 - 1 < nums1.size()) ? nums1[i + k / 2 - 1] : INT_MAX;
        int midVal2 = (j + k / 2 - 1 < nums2.size()) ? nums2[j + k / 2 - 1] : INT_MAX;
        if (midVal1 < midVal2) {
            return findKth(nums1, i + k / 2, nums2, j, k - k / 2);
        } else {
            return findKth(nums1, i, nums2, j + k / 2, k - k / 2);
        }
    }
```

```
};
```

Java解法一：

```java
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length, n = nums2.length, left = (m + n + 1) / 2, right = (m + n + 2) / 2;
        return (findKth(nums1, 0, nums2, 0, left) + findKth(nums1, 0, nums2, 0, right)) / 2.0;
    }
    int findKth(int[] nums1, int i, int[] nums2, int j, int k) {
        if (i >= nums1.length) return nums2[j + k - 1];
        if (j >= nums2.length) return nums1[i + k - 1];
        if (k == 1) return Math.min(nums1[i], nums2[j]);
        int midVal1 = (i + k / 2 - 1 < nums1.length) ? nums1[i + k / 2 - 1] : Integer.MAX_VALUE;
        int midVal2 = (j + k / 2 - 1 < nums2.length) ? nums2[j + k / 2 - 1] : Integer.MAX_VALUE;
        if (midVal1 < midVal2) {
            return findKth(nums1, i + k / 2, nums2, j, k - k / 2);
        } else {
            return findKth(nums1, i, nums2, j + k / 2, k - k / 2);
        }
    }
}
```

上面的解法一直使用的是原数组，同时用了两个变量来分别标记当前的起始位置。我们也可以直接生成新的数组，这样就不要用起始位置变量了，不过拷贝数组的操作可能会增加时间复杂度，也许会超出限制，不过就算当个思路拓展也是极好的。首先要判断数组是否为空，为空的话，直接在另一个数组找第K个即可。还有一种情况是当 K = 1 时，表示要找第一个元素，只要比较两个数组的第一个元素，返回较小的那个即可。这里分别取出两个数组的第 K/2 个数字的位置坐标i和j，为了避免数组没有第 K/2 个数组的情况，每次都和数组长度做比较，取出较小值。这里跟上面的解法有些许不同，上面解法直接取出的是值，而这里取出的是位置坐标，但是思想都是很类似的。不同在于，上面解法中每次固定淘汰 K/2 个数字，而这里由于取出了合法的i和j，所以每次淘汰i或j个。评论区有网友提出，可以让 j = k-i，这样也是对的，可能还更好一些，收敛速度可能会更快一些，参见代码如下：

C++解法二：

```cpp
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int m = nums1.size(), n = nums2.size();
        return (findKth(nums1, nums2, (m + n + 1) / 2) + findKth(nums1, nums2, (m + n + 2) / 2)) / 2.0;
    }
    int findKth(vector<int> nums1, vector<int> nums2, int k) {
```

```
        if (nums1.empty()) return nums2[k - 1];
        if (nums2.empty()) return nums1[k - 1];
        if (k == 1) return min(nums1[0], nums2[0]);
        int i = min((int)nums1.size(), k / 2), j = min((int)nums2.size(), k / 2);
        if (nums1[i - 1] > nums2[j - 1]) {
            return findKth(nums1, vector<int>(nums2.begin() + j, nums2.end()), k - j);
        } else {
            return findKth(vector<int>(nums1.begin() + i, nums1.end()), nums2, k - i);
        }
        return 0;
    }
};
```

Java解法二：

```
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length, n = nums2.length, left = (m + n + 1) / 2, right = (m + n + 2) / 2;
        return (findKth(nums1, nums2, left) + findKth(nums1, nums2, right)) / 2.0;
    }
    int findKth(int[] nums1, int[] nums2, int k) {
        int m = nums1.length, n = nums2.length;
        if (m == 0) return nums2[k - 1];
        if (n == 0) return nums1[k - 1];
        if (k == 1) return Math.min(nums1[0], nums2[0]);
        int i = Math.min(m, k / 2), j = Math.min(n, k / 2);
        if (nums1[i - 1] > nums2[j - 1]) {
            return findKth(nums1, Arrays.copyOfRange(nums2, j, n), k - j);
        } else {
            return findKth(Arrays.copyOfRange(nums1, i, m), nums2, k - i);
        }
    }
}
```

此题还能用迭代形式的二分搜索法来解，是一种相当巧妙的应用，这里就参照 stellari 大神的帖子 来讲解吧。所谓的中位数，换一种角度去看，其实就是把一个有序数组分为长度相等的两段，中位数就是前半段的最大值和后半段的最小值的平均数，也就是离分割点相邻的两个数字的平均值。比如说对于偶数个数组 [1 3 5 7]，那么分割开来就是 [1 3 / 5 7]，其中 '/' 表示分割点，中位数就是3和5的平均值。对于奇数个数组 [1 3 4 5 7]，可以分割为 [1 3 4 / 4 5 7]，可以发现左右两边都有个4，则中位数是两个4的平均数，还是4。这里使用L表示分割点左边的数字，R表示分割点右边的数字，则对于 [1 3 5 7] 来说，L=3，R=5。对于 [1 3 4 5 7] 来说，L=4，R=4。那么对于长度为N的数组来说，可以分别得到L和R的位置，如下所示：

```
N        Index of L        Index of R
1             0                 0
2             0                 1
3             1                 1
4             1                 2
5             2                 2
6             2                 3
7             3                 3
8             3                 4
```

观察上表，可以得到规律，Idx(L)= (N-1)/2，idx(R) = N/2，所以中位数可以用下式表示：

```
(L + R) / 2 = (A[(N - 1) / 2] + A[N / 2]) / 2
```

为了统一数组长度为奇数和偶数的情况，可以使用一个小 tricky，即在每个数字的两边都加上一个特殊字符，比如井号，这个 tricky 其实在马拉车算法中也使用过，可以参见博主之前的帖子 Manacher's Algorithm 马拉车算法。这样做的好处是不管奇数或者偶数，加井号后数组的长度都是奇数，并且切割点的位置也是确定的，比如：

```
[1 3 5 7]    ->    [# 1 # 3 # 5 # 7 #]        N = 4
index                0 1 2 3 4 5 6 7 8        newN = 9

[1 3 4 5 7]    ->    [# 1 # 3 # 4 # 5 # 7 #]        N = 5
index                  0 1 2 3 4 5 6 7 8 9 10        newN = 11
```

这里的N是原数组的长度，newN 是添加井号后新数组的长度，可以发现 newN = 2N+1，而且切割点永远都在新数组中坐标为N的位置，且 idx(L) = (N-1)/2，idx(R) = N/2，这里的N就可以换成分割点的位置，岂不美哉（注意这里的 idx(L) 和 idx(R) 表示的是在未填充#号的坐标位置）！现在假设有两个数组：

```
[1 3 4 5 7]    ->    [# 1 # 3 # 4 # 5 # 7 #]        N1 = 5
index                  0 1 2 3 4 5 6 7 8 9 10        newN1 = 11

[1 2 2 2]    ->    [# 1 # 2 # 2 # 2 #]        N2 = 4
index                0 1 2 3 4 5 6 7 8        newN2 = 9
```

跟只有一个数组的情况类似，这里需要找到一个切割点，使得其分别可以将两个数组分成左右两部分，需要满足的是两个左半边中的任意一个数字都要小于两个右半边数组的数字，注意这里可能有的左半边或右半边会为空，但是两个左半边数字的个数和应该等于两个右半边的个数和。这里还可以观察出一些规律：

1. 总共有 2N1 + 2N2 + 2 个位置，那么除去两个分割点，两个左右半边应该各有 N1 + N2 个数字。
2. 因此，对于一个在 A2 数组中的分割点位置 C2 = K，在 A1 数组中的位置应该为 C1 = N1 + N2 - K，比如假如在 A2 中的分割点位置为 C2 = 2，那么在 A1 中的位置为 C1 = 4 + 5 - C2 = 7。

```
[# 1 # 3 # 4 # (5/5) # 7 #]

[# 1 / 2 # 2 # 2 #]
```

3. 假如两个数组都被分割了，那么就应该会有两个L和R，分别是：

```
L1 = A1[(C1 - 1) / 2]
R1 = A1[C1 / 2]

L2 = A2[(C2 - 1) / 2]
R2 = A2[C2 / 2]
```

对于上面的例子就有：

```
L1 = A1[(7 - 1) / 2] = A1[3] = 5
R1 = A1[7 / 2] = A1[3] = 5

L2 = A2[(2 - 1) / 2] = A2[0] = 1
R2 = A2[2 / 2] = A2[1] = 2
```

现在需要检测这个切割点是否是正确的中位数的切割点，那么根据之前的分析，任意的左半边的数字都需要小于等于右半边的数字，L1 和 L2 是左半边的最大的数字，R1 和 R2 是右半边的最小的数字，所以需要满足下列关系：

```
L1 <= R1 && L1 <= R2 && L2 <= R1 && L2 <= R2
```

由于两个数组都是有序的，所以 L1 <= R1 和 L2 <= R2 都是满足的，那么就只需要满足下列的不等式即可：

```
L1 <= R2 && L2 <= R1
```

这样的话就可以利用二分搜索了，假如 L1 > R2 的话，说明数组 A1 的左半边的数字过大了，需要把切割点 C1 往左移动。假如 L2 > R1，说明数组 A2 的左半边数字过大，需要把分割点 C2 左移。若满足上面的条件，说明当前切割点就是正确的，那么中位数就可以求出来了，即为：

```
(max(L1, L2) + min(R1, R2)) / 2
```

最后还有两点注意事项：

1. 由于 C1 和 C2 是可以互相计算而得，即一个确定了，另一个就可以计算出来了。所以尽量去移动较短的那个数组，这样得到的时间复杂度为 O(lg(min(N1, N2)))。
2. 对于 corner case 的处理，当切割点在 0 或者 2n 的位置时，将L或R的值分别赋值为整型最小值和最大值，这不会改变正确的切割点的位置，会使得代码实现更加方便。

C++ 解法三：

```cpp
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2)
{
        int m = nums1.size(), n = nums2.size();
        if (m < n) return findMedianSortedArrays(nums2, nums1);
        if (n == 0) return ((double)nums1[(m - 1) / 2] + (double)nums1[m /
2]) / 2.0;
        int left = 0, right = n * 2;
        while (left <= right) {
            int mid2 = (left + right) / 2;
            int mid1 = m + n - mid2;
            double L1 = mid1 == 0 ? INT_MIN : nums1[(mid1 - 1) / 2];
            double L2 = mid2 == 0 ? INT_MIN : nums2[(mid2 - 1) / 2];
            double R1 = mid1 == m * 2 ? INT_MAX : nums1[mid1 / 2];
            double R2 = mid2 == n * 2 ? INT_MAX : nums2[mid2 / 2];
            if (L1 > R2) left = mid2 + 1;
            else if (L2 > R1) right = mid2 - 1;
            else return (max(L1, L2) + min(R1, R2)) / 2;
        }
        return -1;
    }
};
```

Java 解法三：

```java
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length, n = nums2.length;
        if (m < n) return findMedianSortedArrays(nums2, nums1);
        if (n == 0) return (nums1[(m - 1) / 2] + nums1[m / 2]) / 2.0;
        int left = 0, right = 2 * n;
        while (left <= right) {
            int mid2 = (left + right) / 2;
            int mid1 = m + n - mid2;
            double L1 = mid1 == 0 ? Double.MIN_VALUE : nums1[(mid1 - 1) /
2];
            double L2 = mid2 == 0 ? Double.MIN_VALUE : nums2[(mid2 - 1) /
2];
            double R1 = mid1 == m * 2 ? Double.MAX_VALUE : nums1[mid1 /
2];
            double R2 = mid2 == n * 2 ? Double.MAX_VALUE : nums2[mid2 /
2];
            if (L1 > R2) left = mid2 + 1;
            else if (L2 > R1) right = mid2 - 1;
            else return (Math.max(L1, L2) + Math.min(R1, R2)) / 2;
        }
        return -1;
```

```
    }
}
```

5. Longest Palindromic Substring

Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.

Example 1:

```
Input: "babad"
Output: "bab"
Note: "aba" is also a valid answer.
```

Example 2:

```
Input: "cbbd"
Output: "bb"
```

这道题让我们求最长回文子串，首先说下什么是回文串，就是正读反读都一样的字符串，比如 "bob", "level", "noon" 等等。那么最长回文子串就是在一个字符串中的那个最长的回文子串。LeetCode 中关于回文串的题共有五道，除了这道，其他的四道为 Palindrome Number，Validate Palindrome，Palindrome Partitioning，Palindrome Partitioning II，我们知道传统的验证回文串的方法就是两个两个的对称验证是否相等，那么对于找回文字串的问题，就要以每一个字符为中心，像两边扩散来寻找回文串，这个算法的时间复杂度是 O(n*n)，可以通过 OJ，就是要注意奇偶情况，由于回文串的长度可奇可偶，比如 "bob" 是奇数形式的回文，"noon" 就是偶数形式的回文，两种形式的回文都要搜索，对于奇数形式的，我们就从遍历到的位置为中心，向两边进行扩散，对于偶数情况，我们就把当前位置和下一个位置当作偶数行回文的最中间两个字符，然后向两边进行搜索，参见代码如下：

解法一：

```
class Solution {
public:
    string longestPalindrome(string s) {
        if (s.size() < 2) return s;
        int n = s.size(), maxLen = 0, start = 0;
        for (int i = 0; i < n - 1; ++i) {
            searchPalindrome(s, i, i, start, maxLen);
            searchPalindrome(s, i, i + 1, start, maxLen);
        }
        return s.substr(start, maxLen);
    }
    void searchPalindrome(string s, int left, int right, int& start, int&
maxLen) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left; ++right;
        }
```

```
        if (maxLen < right - left - 1) {
            start = left + 1;
            maxLen = right - left - 1;
        }
    }
};
```

我们也可以不使用子函数，直接在一个函数中搞定，我们还是要定义两个变量 start 和 maxLen，分别表示最长回文子串的起点跟长度，在遍历s中的字符的时候，我们首先判断剩余的字符数是否小于等于 maxLen 的一半，是的话表明就算从当前到末尾到子串是半个回文串，那么整个回文串长度最多也就是 maxLen，既然 maxLen 无法再变长了，计算这些就没有意义，直接在当前位置 break 掉就行了。否则就要继续判断，我们用两个变量 left 和 right 分别指向当前位置，然后我们先要做的是向右遍历跳过重复项，这个操作很必要，比如对于 noon，i在第一个o的位置，如果我们以o为最中心往两边扩散，是无法得到长度为4的回文串的，只有先跳过重复，此时left指向第一个o，right指向第二个o，然后再向两边扩散。而对于 bob，i在第一个o的位置时，无法向右跳过重复，此时 left 和 right 同时指向o，再向两边扩散也是正确的，所以可以同时处理奇数和偶数的回文串，之后的操作就是更新 maxLen 和 start 了，跟上面的操作一样，参见代码如下：

解法二：

```
class Solution {
public:
    string longestPalindrome(string s) {
        if (s.size() < 2) return s;
        int n = s.size(), maxLen = 0, start = 0;
        for (int i = 0; i < n;) {
            if (n - i <= maxLen / 2) break;
            int left = i, right = i;
            while (right < n - 1 && s[right + 1] == s[right]) ++right;
            i = right + 1;
            while (right < n - 1 && left > 0 && s[right + 1] == s[left - 1]) {
                ++right; --left;
            }
            if (maxLen < right - left + 1) {
                maxLen = right - left + 1;
                start = left;
            }
        }
        return s.substr(start, maxLen);
    }
};
```

此题还可以用动态规划 Dynamic Programming 来解，根 Palindrome Partitioning II 的解法很类似，我们维护一个二维数组 dp，其中 dp[i][j] 表示字符串区间 [i, j] 是否为回文串，当 i = j 时，只有一个字符，肯定是回文串，如果 i = j + 1，说明是相邻字符，此时需要判断 s[i] 是否等于 s[j]，如果i和j不相邻，即 i - j >= 2 时，除了判断 s[i] 和 s[j] 相等之外，dp[i + 1][j - 1] 若为真，就是回文串，通过以上分析，可以写出递推式如下：

dp[i, j] = 1 if i == j

= s[i] == s[j] if j = i + 1

= s[i] == s[j] && dp[i + 1][j - 1] if j > i + 1

这里有个有趣的现象就是如果我把下面的代码中的二维数组由 int 改为 vector 后，就会超时，这说明 int 型的二维数组访问执行速度完爆 std 的 vector 啊，所以以后尽可能的还是用最原始的数据类型吧。

解法三：

```
class Solution {
public:
    string longestPalindrome(string s) {
        if (s.empty()) return "";
        int n = s.size(), dp[n][n] = {0}, left = 0, len = 1;
        for (int i = 0; i < n; ++i) {
            dp[i][i] = 1;
            for (int j = 0; j < i; ++j) {
                dp[j][i] = (s[i] == s[j] && (i - j < 2 || dp[j + 1][i - 1]));
                if (dp[j][i] && len < i - j + 1) {
                    len = i - j + 1;
                    left = j;
                }
            }
        }
        return s.substr(left, len);
    }
};
```

最后要来的就是大名鼎鼎的马拉车算法 Manacher's Algorithm，这个算法的神奇之处在于将时间复杂度提升到了 O(n) 这种逆天的地步，而算法本身也设计的很巧妙，很值得我们掌握，参见我另一篇专门介绍马拉车算法的博客 Manacher's Algorithm 马拉车算法，代码实现如下：

解法四：

```
class Solution {
public:
    string longestPalindrome(string s) {
        string t ="$#";
        for (int i = 0; i < s.size(); ++i) {
            t += s[i];
            t += '#';
        }
        int p[t.size()] = {0}, id = 0, mx = 0, resId = 0, resMx = 0;
        for (int i = 1; i < t.size(); ++i) {
            p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
            while (t[i + p[i]] == t[i - p[i]]) ++p[i];
            if (mx < i + p[i]) {
```

```
                mx = i + p[i];
                id = i;
            }
            if (resMx < p[i]) {
                resMx = p[i];
                resId = i;
            }
        }
        return s.substr((resId - resMx) / 2, resMx - 1);
    }
};
```

10. Regular Expression Matching

Given an input string (`s`) and a pattern (`p`), implement regular expression matching with support for `'.'` and `'*'`.

```
'.' Matches any single character.
'*' Matches zero or more of the preceding element.
```

The matching should cover the entire input string (not partial).

Note:

- `s` could be empty and contains only lowercase letters `a-z`.
- `p` could be empty and contains only lowercase letters `a-z`, and characters like `.` or `*`.

Example 1:

```
Input:
s = "aa"
p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input:
s = "aa"
p = "a*"
Output: true
Explanation: '*' means zero or more of the precedeng element, 'a'.
Therefore, by repeating 'a' once, it becomes "aa".
```

Example 3:

```
Input:
s = "ab"
p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any character (.)".
```

Example 4:

```
Input:
s = "aab"
p = "c*a*b"
Output: true
Explanation: c can be repeated 0 times, a can be repeated 1 time. Therefore
it matches "aab".
```

Example 5:

```
Input:
s = "mississippi"
p = "mis*is*p*."
Output: false
```

这道求正则表达式匹配的题和那道 Wildcard Matching 的题很类似，不同点在于*的意义不同，在之前那道题中，*表示可以代替任意个数的字符，而这道题中的*表示之前那个字符可以有0个，1个或是多个，就是说，字符串 a*b，可以表示b或是 aaab，即a的个数任意，这道题的难度要相对之前那一道大一些，分的情况的要复杂一些，需要用递归 Recursion 来解，大概思路如下：

- 若p为空，若s也为空，返回 true，反之返回 false。

- 若p的长度为1，若s长度也为1，且相同或是p为 '.' 则返回 true，反之返回 false。

- 若p的第二个字符不为*，若此时s为空返回 false，否则判断首字符是否匹配，且从各自的第二个字符开始调用递归函数匹配。

- 若p的第二个字符为*，进行下列循环，条件是若s不为空且首字符匹配（包括 p[0] 为点），调用递归函数匹配s和去掉前两个字符的p（这样做的原因是假设此时的星号的作用是让前面的字符出现0次，验证是否匹配），若匹配返回 true，否则s去掉首字母（因为此时首字母匹配了，我们可以去掉s的首字母，而p由于星号的作用，可以有任意个首字母，所以不需要去掉），继续进行循环。

- 返回调用递归函数匹配s和去掉前两个字符的p的结果（这么做的原因是处理星号无法匹配的内容，比如 s="ab", p="a*b"，*直接进入 while 循环后，我们发现 "ab" 和 "b" 不匹配，所以s变成 "b"，那么此时跳出循环后，就到最后的 return 来比较 "b" 和 "b" 了，返回 true。再举个例子，比如 s="", p="a*"*，由于s为空，不会进入任何的 if 和 while，只能到最后的 return 来比较了，返回 true，正确）。

解法一：

```
class Solution {
public:
    bool isMatch(string s, string p) {

```

```
        if (p.empty()) return s.empty();
        if (p.size() == 1) {
            return (s.size() == 1 && (s[0] == p[0] || p[0] == '.'));
        }
        if (p[1] != '*') {
            if (s.empty()) return false;
            return (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p.substr(1));
        }
        while (!s.empty() && (s[0] == p[0] || p[0] == '.')) {
            if (isMatch(s, p.substr(2))) return true;
            s = s.substr(1);
        }
        return isMatch(s, p.substr(2));
    }
};
```

上面的方法可以写的更加简洁一些，但是整个思路还是一样的，先来判断p是否为空，若为空则根据s的为空的情况返回结果。当p的第二个字符为*号时，由于*号前面的字符的个数可以任意，可以为0，那么我们先用递归来调用为0的情况，就是直接把这两个字符去掉再比较，或者当s不为空，且第一个字符和p的第一个字符相同时，再对去掉首字符的s和p调用递归，注意p不能去掉首字符，因为*号前面的字符可以有无限个；如果第二个字符不为*号，那么就老老实实的比较第一个字符，然后对后面的字符串调用递归，参见代码如下：

解法二：

```
class Solution {
public:
    bool isMatch(string s, string p) {
        if (p.empty()) return s.empty();
        if (p.size() > 1 && p[1] == '*') {
            return isMatch(s, p.substr(2)) || (!s.empty() && (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p));
        } else {
            return !s.empty() && (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p.substr(1));
        }
    }
};
```

我们也可以用 DP 来解，定义一个二维的 DP 数组，其中 dp[i][j] 表示 s[0,i) 和 p[0,j) 是否 match，然后有下面三种情况(下面部分摘自这个帖子)：

1. P[i][j] = P[i - 1][j - 1], if p[j - 1] != '*' && (s[i - 1] == p[j - 1] || p[j - 1] == '.');
2. P[i][j] = P[i][j - 2], if p[j - 1] == '*' and the pattern repeats for 0 times;
3. P[i][j] = P[i - 1][j] && (s[i - 1] == p[j - 2] || p[j - 2] == '.'), if p[j - 1] == '*' and the pattern repeats for at least 1 times.

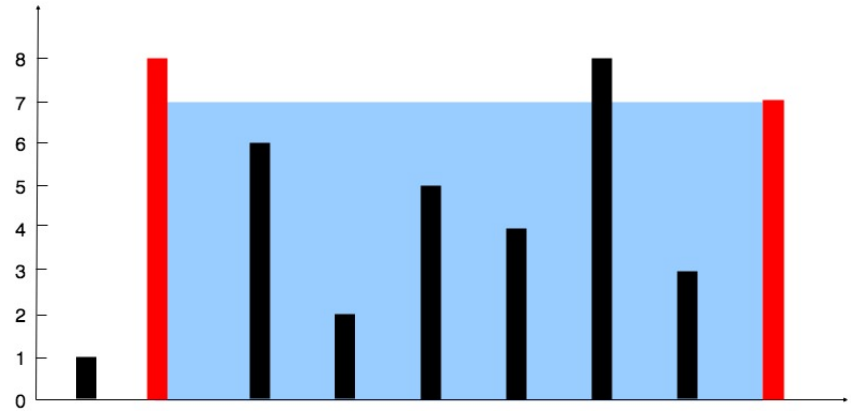
解法三：

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size(), n = p.size();
        vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
        dp[0][0] = true;
        for (int i = 0; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (j > 1 && p[j - 1] == '*') {
                    dp[i][j] = dp[i][j - 2] || (i > 0 && (s[i - 1] == p[j - 2] || p[j - 2] == '.') && dp[i - 1][j]);
                } else {
                    dp[i][j] = i > 0 && dp[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '.');
                }
            }
        }
        return dp[m][n];
    }
};
```

11. Container With Most Water

Given *n* non-negative integers *a1* , *a2* , ..., *an* , where each represents a point at coordinate (*i* , *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i* , *ai*) and (*i* , 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and *n* is at least 2.



The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example:

```
Input: [1,8,6,2,5,4,8,3,7]
Output: 49
```

这道求装最多水的容器的题和那道 Trapping Rain Water 很类似，但又有些不同，那道题让求整个能收集雨水的量，这道只是让求最大的一个的装水量，而且还有一点不同的是，那道题容器边缘不能算在里面，而这道题却可以算，相比较来说还是这道题容易一些，这里需要定义i和j两个指针分别指向数组的左右两端，然后两个指针向中间搜索，每移动一次算一个值和结果比较取较大的，容器装水量的算法是找出左右两个边缘中较小的那个乘以两边缘的距离，代码如下：

C++ 解法一：

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int res = 0, i = 0, j = height.size() - 1;
        while (i < j) {
            res = max(res, min(height[i], height[j]) * (j - i));
            height[i] < height[j] ? ++i : --j;
        }
        return res;
    }
};
```

Java 解法一：

```
public class Solution {
    public int maxArea(int[] height) {
        int res = 0, i = 0, j = height.length - 1;
        while (i < j) {
            res = Math.max(res, Math.min(height[i], height[j]) * (j - i));
            if (height[i] < height[j]) ++i;
            else --j;
        }
        return res;
    }
```

这里需要注意的是，由于 Java 中的三元运算符 A?B:C 必须须要有返回值，所以只能用 if..else.. 来替换，不知道 Java 对于三元运算符这么严格的限制的原因是什么。

下面这种方法是对上面的方法进行了小幅度的优化，对于相同的高度们直接移动i和j就行了，不再进行计算容量了，参见代码如下：

C++ 解法二：

```cpp
class Solution {
public:
    int maxArea(vector<int>& height) {
        int res = 0, i = 0, j = height.size() - 1;
        while (i < j) {
            int h = min(height[i], height[j]);
            res = max(res, h * (j - i));
            while (i < j && h == height[i]) ++i;
            while (i < j && h == height[j]) --j;
        }
        return res;
    }
};
```

Java 解法二：

```java
public class Solution {
    public int maxArea(int[] height) {
        int res = 0, i = 0, j = height.length - 1;
        while (i < j) {
            int h = Math.min(height[i], height[j]);
            res = Math.max(res, h * (j - i));
            while (i < j && h == height[i]) ++i;
            while (i < j && h == height[j]) --j;
        }
        return res;
    }
}
```

12. Integer to Roman

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

```
Symbol       Value
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

For example, two is written as `II` in Roman numeral, just two one's added together. Twelve is written as, `XII`, which is simply `X` + `II`. The number twenty seven is written as `XXVII`, which is `XX` + `V` + `II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral. Input is guaranteed to be within the range from 1 to 3999.

Example 1:

```
Input: 3
Output: "III"
```

Example 2:

```
Input: 4
Output: "IV"
```

Example 3:

```
Input: 9
Output: "IX"
```

Example 4:

```
Input: 58
Output: "LVIII"
Explanation: L = 50, V = 5, III = 3.
```

Example 5:

```
Input: 1994
Output: "MCMXCIV"
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

之前那篇文章写的是罗马数字转化成整数 Roman to Integer， 这次变成了整数转化成罗马数字，基本算法还是一样。由于题目中限定了输入数字的范围 (1 - 3999), 使得题目变得简单了不少。

I - 1

V - 5

X - 10

L - 50

C - 100

D - 500

M - 1000

例如整数 1437 的罗马数字为 MCDXXXVII， 我们不难发现，千位，百位，十位和个位上的数分别用罗马数字表示了。 1000 - M, 400 - CD, 30 - XXX, 7 - VII。所以我们要做的就是用取商法分别提取各个位上的数字，然后分别表示出来：

100 - C

200 - CC

300 - CCC

400 - CD

500 - D

600 - DC

700 - DCC

800 - DCCC

900 - CM

可以分为四类，100 到 300 一类，400 一类，500 到 800 一类，900 最后一类。每一位上的情况都是类似的，代码如下：

解法一：

```
class Solution {
public:
    string intToRoman(int num) {
        string res = "";
        vector<char> roman{'M', 'D', 'C', 'L', 'X', 'V', 'I'};
        vector<int> value{1000, 500, 100, 50, 10, 5, 1};
        for (int n = 0; n < 7; n += 2) {
            int x = num / value[n];
            if (x < 4) {
                for (int i = 1; i <= x; ++i) res += roman[n];
            } else if (x == 4) {
                res = res + roman[n] + roman[n - 1];
            } else if (x > 4 && x < 9) {
                res += roman[n - 1];
                for (int i = 6; i <= x; ++i) res += roman[n];
            } else if (x == 9) {
                res = res + roman[n] + roman[n - 2];
            }
            num %= value[n];
        }
        return res;
```

```
    }
};
```

本题由于限制了输入数字范围这一特殊性，故而还有一种利用贪婪算法的解法，建立一个数表，每次通过查表找出当前最大的数，减去再继续查表，参见代码如下：

解法二：

```
class Solution {
public:
    string intToRoman(int num) {
        string res = "";
        vector<int> val{1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
        vector<string> str{"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
        for (int i = 0; i < val.size(); ++i) {
            while (num >= val[i]) {
                num -= val[i];
                res += str[i];
            }
        }
        return res;
    }
};
```

下面这种方法个人感觉属于比较投机取巧的方法，把所有的情况都列了出来，然后直接按位查表，O(1)的时间复杂度啊，参见代码如下：

解法三：

```
class Solution {
public:
    string intToRoman(int num) {
        string res = "";
        vector<string> v1{"", "M", "MM", "MMM"};
        vector<string> v2{"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
        vector<string> v3{"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
        vector<string> v4{"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
        return v1[num / 1000] + v2[(num % 1000) / 100] + v3[(num % 100) / 10] + v4[num % 10];
    }
};
```

15. 3Sum

Given an array *S* of *n* integers, are there elements *a* , *b* , *c* in *S* such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

- Elements in a triplet (*a* , *b* , *c*) must be in non-descending order. (ie, *a* ≤ *b* ≤ *c*)
- The solution set must not contain duplicate triplets.

```
    For example, given array S = {-1 0 1 2 -1 -4},

    A solution set is:
    (-1, 0, 1)
    (-1, -1, 2)
```

这道题让我们求三数之和，比之前那道 Two Sum 要复杂一些，博主考虑过先 fix 一个数，然后另外两个数使用 Two Sum 那种 HashMap 的解法，但是会有重复结果出现，就算使用 TreeSet 来去除重复也不行，会 TLE，看来此题并不是考 Two Sum 的解法。来分析一下这道题的特点，要找出三个数且和为0，那么除了三个数全是0的情况之外，肯定会有负数和正数，还是要先 fix 一个数，然后去找另外两个数，只要找到两个数且和为第一个 fix 数的相反数就行了，既然另外两个数不能使用 Two Sum 的那种解法来找，如何能更有效的定位呢？我们肯定不希望遍历所有两个数的组合吧，所以如果数组是有序的，那么就可以用双指针以线性时间复杂度来遍历所有满足题意的两个数组合。

对原数组进行排序，然后开始遍历排序后的数组，这里注意不是遍历到最后一个停止，而是到倒数第三个就可以了。这里可以先做个剪枝优化，就是当遍历到正数的时候就 break，为啥呢，因为数组现在是有序的了，如果第一个要 fix 的数就是正数了，则后面的数字就都是正数，就永远不会出现和为0的情况了。然后还要加上重复就跳过的处理，处理方法是从第二个数开始，如果和前面的数字相等，就跳过，因为不想把相同的数字fix两次。对于遍历到的数，用0减去这个 fix 的数得到一个 target，然后只需要再之后找到两个数之和等于 target 即可。用两个指针分别指向 fix 数字之后开始的数组首尾两个数，如果两个数和正好为 target，则将这两个数和 fix 的数一起存入结果中。然后就是跳过重复数字的步骤了，两个指针都需要检测重复数字。如果两数之和小于 target，则将左边那个指针i右移一位，使得指向的数字增大一些。同理，如果两数之和大于 target，则将右边那个指针j左移一位，使得指向的数字减小一些，代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());
        if (nums.empty() || nums.back() < 0 || nums.front() > 0) return
{};
        for (int k = 0; k < (int)nums.size() - 2; ++k) {
            if (nums[k] > 0) break;
            if (k > 0 && nums[k] == nums[k - 1]) continue;
            int target = 0 - nums[k], i = k + 1, j = (int)nums.size() - 1;
```

```
            while (i < j) {
                if (nums[i] + nums[j] == target) {
                    res.push_back({nums[k], nums[i], nums[j]});
                    while (i < j && nums[i] == nums[i + 1]) ++i;
                    while (i < j && nums[j] == nums[j - 1]) --j;
                    ++i; --j;
                } else if (nums[i] + nums[j] < target) ++i;
                else --j;
            }
        }
        return res;
    }
};
```

或者我们也可以利用 TreeSet 的不能包含重复项的特点来防止重复项的产生，那么就不需要检测数字是否被 fix 过两次，不过个人觉得还是前面那种解法更好一些，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        set<vector<int>> res;
        sort(nums.begin(), nums.end());
        if (nums.empty() || nums.back() < 0 || nums.front() > 0) return
{};
        for (int k = 0; k < (int)nums.size() - 2; ++k) {
            if (nums[k] > 0) break;
            int target = 0 - nums[k], i = k + 1, j = (int)nums.size() - 1;
            while (i < j) {
                if (nums[i] + nums[j] == target) {
                    res.insert({nums[k], nums[i], nums[j]});
                    while (i < j && nums[i] == nums[i + 1]) ++i;
                    while (i < j && nums[j] == nums[j - 1]) --j;
                    ++i; --j;
                } else if (nums[i] + nums[j] < target) ++i;
                else --j;
            }
        }
        return vector<vector<int>>(res.begin(), res.end());
    }
};
```

16. 3Sum Closest

Given an array `nums` of *n* integers and an integer `target`, find three integers in `nums` such that the sum is closest to `target`. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example:

```
Given array nums = [-1, 2, 1, -4], and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

这道题让我们求最接近给定值的三数之和，是在之前那道 3Sum 的基础上又增加了些许难度，那么这道题让返回这个最接近于给定值的值，即要保证当前三数和跟给定值之间的差的绝对值最小，所以需要定义一个变量 diff 用来记录差的绝对值，然后还是要先将数组排个序，然后开始遍历数组，思路跟那道三数之和很相似，都是先确定一个数，然后用两个指针 left 和 right 来滑动寻找另外两个数，每确定两个数，求出此三数之和，然后算和给定值的差的绝对值存在 newDiff 中，然后和 diff 比较并更新 diff 和结果 closest 即可，代码如下：

解法一：

```
class Solution {
public:
    int threeSumClosest(vector<int>& nums, int target) {
        int closest = nums[0] + nums[1] + nums[2];
        int diff = abs(closest - target);
        sort(nums.begin(), nums.end());
        for (int i = 0; i < nums.size() - 2; ++i) {
            int left = i + 1, right = nums.size() - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                int newDiff = abs(sum - target);
                if (diff > newDiff) {
                    diff = newDiff;
                    closest = sum;
                }
                if (sum < target) ++left;
                else --right;
            }
        }
        return closest;
    }
};
```

我们还可以稍稍进行一下优化，每次判断一下，当 nums[i]*3 > target 的时候，就可以直接比较 closest 和 nums[i] + nums[i+1] + nums[i+2] 的值，返回较小的那个，因为数组已经排过序了，后面的数字只会越来越大，就不必再往后比较了，参见代码如下：

解法二：

```
class Solution {
public:
    int threeSumClosest(vector<int>& nums, int target) {
        int closest = nums[0] + nums[1] + nums[2];
        int diff = abs(closest - target);
        sort(nums.begin(), nums.end());
        for (int i = 0; i < nums.size() - 2; ++i) {
            if (nums[i] * 3 > target) return min(closest, nums[i] + nums[i + 1] + nums[i + 2]);
            int left = i + 1, right = nums.size() - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                int newDiff = abs(sum - target);
                if (diff > newDiff) {
                    diff = newDiff;
                    closest = sum;
                }
                if (sum < target) ++left;
                else --right;
            }
        }
        return closest;
    }
};
```

17. Letter Combinations of a Phone Number

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example:

```
Input: "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

这道题让我们求电话号码的字母组合，即数字2到9中每个数字可以代表若干个字母，然后给一串数字，求出所有可能的组合，相类似的题目有 Path Sum II，Subsets II，Permutations，Permutations II，Combinations，Combination Sum 和 Combination Sum II 等等。这里可以用递归 Recursion 来解，需要建立一个字典，用来保存每个数字所代表的字符串，然后还需要一个变量 level，记录当前生成的字符串的字符个数，实现套路和上述那些题十分类似。在递归函数中首先判断 level，如果跟 digits 中数字的个数相等了，将当前的组合加入结果 res 中，然后返回。我们通过 digits 中的数字到 dict 中取出字符串，然后遍历这个取出的字符串，将每个字符都加到当前的组合后面，并调用递归函数即可，参见代码如下：

解法一：

```
class Solution {
public:
    vector<string> letterCombinations(string digits) {
        if (digits.empty()) return {};
        vector<string> res;
        vector<string> dict{"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
        letterCombinationsDFS(digits, dict, 0, "", res);
        return res;
    }
    void letterCombinationsDFS(string& digits, vector<string>& dict, int level, string out, vector<string>& res) {
        if (level == digits.size()) {res.push_back(out); return;}
        string str = dict[digits[level] - '0'];
        for (int i = 0; i < str.size(); ++i) {
            letterCombinationsDFS(digits, dict, level + 1, out + str[i], res);
        }
    }
};
```

这道题也可以用迭代 Iterative 来解，在遍历 digits 中所有的数字时，先建立一个临时的字符串数组t，然后跟上面解法的操作一样，通过数字到 dict 中取出字符串 str，然后遍历取出字符串中的所有字符，再遍历当前结果 res 中的每一个字符串，将字符加到后面，并加入到临时字符串数组t中。取出的字符串 str 遍历完成后，将临时字符串数组赋值给结果 res，具体实现参见代码如下：

解法二：

```
class Solution {
public:
    vector<string> letterCombinations(string digits) {
        if (digits.empty()) return {};
        vector<string> res{""};
        vector<string> dict{"", "", "abc", "def", "ghi", "jkl", "mno",
"pqrs", "tuv", "wxyz"};
        for (int i = 0; i < digits.size(); ++i) {
            vector<string> t;
            string str = dict[digits[i] - '0'];
            for (int j = 0; j < str.size(); ++j) {
                for (string s : res) t.push_back(s + str[j]);
            }
            res = t;
        }
        return res;
    }
};
```

18. 4Sum

Given an array *S* of *n* integers, are there elements *a* , *b* , *c* , and *d* in *S* such that *a* + *b* + *c* + *d* = target? Find all unique quadruplets in the array which gives the sum of target.

Note:

- Elements in a quadruplet (*a* , *b* , *c* , *d*) must be in non-descending order. (ie, *a* ≤ *b* ≤ *c* ≤ *d*)
- The solution set must not contain duplicate quadruplets.

```
For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:
(-1,  0, 0, 1)
(-2, -1, 1, 2)
(-2,  0, 0, 2)
```

LeetCode 中关于数字之和还有其他几道，分别是 Two Sum ， 3Sum ， 3Sum Closest，虽然难度在递增，但是整体的套路都是一样的，在这里为了避免重复项，我们使用了 STL 中的 TreeSet，其特点是不能有重复，如果新加入的数在 TreeSet 中原本就存在的话，插入操作就会失败，这样能很好的避免的重复项的存在。此题的 O(n^3) 解法的思路跟 3Sum 基本没啥区别，就是多加了一层 for 循环，其他的都一样，代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int> &nums, int target) {
```

```
        set<vector<int>> res;
        sort(nums.begin(), nums.end());
        for (int i = 0; i < int(nums.size() - 3); ++i) {
            for (int j = i + 1; j < int(nums.size() - 2); ++j) {
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;
                int left = j + 1, right = nums.size() - 1;
                while (left < right) {
                    int sum = nums[i] + nums[j] + nums[left] +
nums[right];
                    if (sum == target) {
                        vector<int> out{nums[i], nums[j], nums[left],
nums[right]};
                        res.insert(out);
                        ++left; --right;
                    } else if (sum < target) ++left;
                    else --right;
                }
            }
        }
        return vector<vector<int>>(res.begin(), res.end());
    }
};
```

但是毕竟用 TreeSet 来进行去重复的处理还是有些取巧，可能在 Java 中就不能这么做，那么还是来看一种比较正统的做法吧，手动进行去重复处理。主要可以进行的有三个地方，首先在两个 for 循环下可以各放一个，因为一旦当前的数字跟上面处理过的数字相同了，那么找下来肯定还是重复的。之后就是当 sum 等于 target 的时候了，在将四个数字加入结果 res 之后，left 和 right 都需要去重复处理，分别像各自的方面遍历即可，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int> &nums, int target) {
        vector<vector<int>> res;
        int n = nums.size();
        sort(nums.begin(), nums.end());
        for (int i = 0; i < n - 3; ++i) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            for (int j = i + 1; j < n - 2; ++j) {
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;
                int left = j + 1, right = n - 1;
                while (left < right) {
                    int sum = nums[i] + nums[j] + nums[left] +
nums[right];
                    if (sum == target) {
                        vector<int> out{nums[i], nums[j], nums[left],
nums[right]};
                        res.push_back(out);
```

```
                        while (left < right && nums[left] == nums[left + 1]) ++left;
                        while (left < right && nums[right] == nums[right - 1]) --right;
                        ++left; --right;
                    } else if (sum < target) ++left;
                    else --right;
                }
            }
        }
        return res;
    }
};
```

22. Generate Parentheses

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

在 LeetCode 中有关括号的题共有七道，除了这一道的另外六道是 Score of Parentheses，Valid Parenthesis String， Remove Invalid Parentheses，Different Ways to Add Parentheses，Valid Parentheses 和 Longest Valid Parentheses。这道题给定一个数字n，让生成共有n个括号的所有正确的形式，对于这种列出所有结果的题首先还是考虑用递归 Recursion 来解，由于字符串只有左括号和右括号两种字符，而且最终结果必定是左括号3个，右括号3个，所以这里定义两个变量 left 和 right 分别表示剩余左右括号的个数，如果在某次递归时，左括号的个数大于右括号的个数，说明此时生成的字符串中右括号的个数大于左括号的个数，即会出现 ')(' 这样的非法串，所以这种情况直接返回，不继续处理。如果 left 和 right 都为0，则说明此时生成的字符串已有3个左括号和3个右括号，且字符串合法，则存入结果中后返回。如果以上两种情况都不满足，若此时 left 大于0，则调用递归函数，注意参数的更新，若 right 大于0，则调用递归函数，同样要更新参数，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> res;
        generateParenthesisDFS(n, n, "", res);
```

```
        return res;
    }
    void generateParenthesisDFS(int left, int right, string out, vector<string> &res) {
        if (left > right) return;
        if (left == 0 && right == 0) res.push_back(out);
        else {
            if (left > 0) generateParenthesisDFS(left - 1, right, out + '(', res);
            if (right > 0) generateParenthesisDFS(left, right - 1, out + ')', res);
        }
    }
};
```

Java 解法一：

```
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> res = new ArrayList<String>();
        helper(n, n, "", res);
        return res;
    }
    void helper(int left, int right, String out, List<String> res) {
        if (left < 0 || right < 0 || left > right) return;
        if (left == 0 && right == 0) {
            res.add(out);
            return;
        }
        helper(left - 1, right, out + "(", res);
        helper(left, right - 1, out + ")", res);
    }
}
```

再来看那一种方法，这种方法是 CareerCup 书上给的方法，感觉也是满巧妙的一种方法，这种方法的思想是找左括号，每找到一个左括号，就在其后面加一个完整的括号，最后再在开头加一个 ()，就形成了所有的情况，需要注意的是，有时候会出现重复的情况，所以用set数据结构，好处是如果遇到重复项，不会加入到结果中，最后我们再把set转为vector即可，参见代码如下：：

n＝1: ()

n=2: (()) ()()

n=3: ((()()) (((()))) ()(()) (())() ()()()

C++ 解法二：

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
```

```
        unordered_set<string> st;
        if (n == 0) st.insert("");
        else {
            vector<string> pre = generateParenthesis(n - 1);
            for (auto a : pre) {
                for (int i = 0; i < a.size(); ++i) {
                    if (a[i] == '(') {
                        a.insert(a.begin() + i + 1, '(');
                        a.insert(a.begin() + i + 2, ')');
                        st.insert(a);
                        a.erase(a.begin() + i + 1, a.begin() + i + 3);
                    }
                }
                st.insert("()" + a);
            }
        }
        return vector<string>(st.begin(), st.end());
    }
};
```

Java 解法二:

```
public class Solution {
    public List<String> generateParenthesis(int n) {
        Set<String> res = new HashSet<String>();
        if (n == 0) {
            res.add("");
        } else {
            List<String> pre = generateParenthesis(n - 1);
            for (String str : pre) {
                for (int i = 0; i < str.length(); ++i) {
                    if (str.charAt(i) == '(') {
                        str = str.substring(0, i + 1) + "()" +
str.substring(i + 1, str.length());
                        res.add(str);
                        str = str.substring(0, i + 1) +  str.substring(i +
3, str.length());
                    }
                }
                res.add("()" + str);
            }
        }
        return new ArrayList(res);
    }
}
```

23. Merge k Sorted Lists

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Example:

```
Input:
[
  1->4->5,
  1->3->4,
  2->6
]
Output: 1->1->2->3->4->4->5->6
```

这道题让我们合并k个有序链表，最终合并出来的结果也必须是有序的，之前做过一道 Merge Two Sorted Lists，是混合插入两个有序链表。这道题增加了难度，变成合并k个有序链表了，但是不管合并几个，基本还是要两两合并。那么首先考虑的方法是能不能利用之前那道题的解法来解答此题。答案是肯定的，但是需要修改，怎么修改呢，最先想到的就是两两合并，就是前两个先合并，合并好了再跟第三个，然后第四个直到第k个。这样的思路是对的，但是效率不高，没法通过 OJ，所以只能换一种思路，这里就需要用到分治法 Divide and Conquer Approach。简单来说就是不停的对半划分，比如k个链表先划分为合并两个 k/2 个链表的任务，再不停的往下划分，直到划分成只有一个或两个链表的任务，开始合并。举个例子来说比如合并6个链表，那么按照分治法，首先分别合并0和3，1和4，2和5。这样下一次只需合并3个链表，再合并1和3，最后和2合并就可以了。代码中的k是通过 (n+1)/2 计算的，这里为啥要加1呢，这是为了当n为奇数的时候，k能始终从后半段开始，比如当 n=5 时，那么此时 k=3，则0和3合并，1和4合并，最中间的2空出来。当n是偶数的时候，加1也不会有影响，比如当 n=4 时，此时 k=2，那么0和2合并，1和3合并，完美解决问题，参见代码如下：

解法一：

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return NULL;
        int n = lists.size();
        while (n > 1) {
            int k = (n + 1) / 2;
            for (int i = 0; i < n / 2; ++i) {
                lists[i] = mergeTwoLists(lists[i], lists[i + k]);
            }
            n = k;
        }
        return lists[0];
    }
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1), *cur = dummy;
        while (l1 && l2) {
            if (l1->val < l2->val) {
                cur->next = l1;
                l1 = l1->next;
            } else {
                cur->next = l2;
```

```
                l2 = l2->next;
            }
            cur = cur->next;
        }
        if (l1) cur->next = l1;
        if (l2) cur->next = l2;
        return dummy->next;
    }
};
```

我们再来看另一种解法，这种解法利用了最小堆这种数据结构，首先把k个链表的首元素都加入最小堆中，它们会自动排好序。然后每次取出最小的那个元素加入最终结果的链表中，然后把取出元素的下一个元素再加入堆中，下次仍从堆中取出最小的元素做相同的操作，以此类推，直到堆中没有元素了，此时k个链表也合并为了一个链表，返回首节点即可，参见代码如下：

解法二：

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        auto cmp = [](ListNode*& a, ListNode*& b) {
            return a->val > b->val;
        };
        priority_queue<ListNode*, vector<ListNode*>, decltype(cmp) > q(cmp);
        for (auto node : lists) {
            if (node) q.push(node);
        }
        ListNode *dummy = new ListNode(-1), *cur = dummy;
        while (!q.empty()) {
            auto t = q.top(); q.pop();
            cur->next = t;
            cur = cur->next;
            if (cur->next) q.push(cur->next);
        }
        return dummy->next;
    }
};
```

下面这种解法利用到了混合排序的思想，也属于分治法的一种，做法是将原链表分成两段，然后对每段调用递归函数，suppose 返回的 left 和 right 已经合并好了，然后再对 left 和 right 进行合并，合并的方法就使用之前那道 Merge Two Sorted Lists 中的任意一个解法即可，这里使用了递归的写法，而本题解法一中用的是迭代的写法，参见代码如下：

解法三：

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
```

```
        return helper(lists, 0, (int)lists.size() - 1);
    }
    ListNode* helper(vector<ListNode*>& lists, int start, int end) {
        if (start > end) return NULL;
        if (start == end) return lists[start];
        int mid = start + (end - start) / 2;
        ListNode *left = helper(lists, start, mid);
        ListNode *right = helper(lists, mid + 1, end);
        return mergeTwoLists(left, right);
    }
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;
        if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

下面这种解法利用到了计数排序的思想，由留言区二楼热心网友闽 A2436 提供，思路是将所有的结点值出现的最大值和最小值都记录下来，然后记录每个结点值出现的次数，这样从最小值遍历到最大值的时候，就会按顺序经过所有的结点值，根据其出现的次数，建立相对应个数的结点。但是这种解法有个特别需要注意的地方，那就是合并后的链表结点都是重新建立的，若在某些情况下，不能新建结点，而只能交换或者重新链接结点的话，那么此解法就不能使用，但好在本题并没有这种限制，可以完美过 OJ，参见代码如下：

解法四：

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        ListNode *dummy = new ListNode(-1), *cur = dummy;
        unordered_map<int, int> m;
        int mx = INT_MIN, mn = INT_MAX;
        for (auto node : lists) {
            ListNode *t = node;
            while (t) {
                mx = max(mx, t->val);
                mn = min(mn, t->val);
                ++m[t->val];
                t = t->next;
            }
        }
        for (int i = mn; i <= mx; ++i) {
            if (!m.count(i)) continue;
```

```
            for (int j = 0; j < m[i]; ++j) {
                cur->next = new ListNode(i);
                cur = cur->next;
            }
        }
        return dummy->next;
    }
};
```

24. Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

You may not modify the values in the list's nodes, only nodes itself may be changed.

Example:

```
Given 1->2->3->4, you should return the list as 2->1->4->3.
```

这道题不算难，是基本的链表操作题，我们可以分别用递归和迭代来实现。对于迭代实现，还是需要建立 dummy 节点，注意在连接节点的时候，最好画个图，以免把自己搞晕了，参见代码如下：

解法一：

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode *dummy = new ListNode(-1), *pre = dummy;
        dummy->next = head;
        while (pre->next && pre->next->next) {
            ListNode *t = pre->next->next;
            pre->next->next = t->next;
            t->next = pre->next;
            pre->next = t;
            pre = t->next;
        }
        return dummy->next;
    }
};
```

递归的写法就更简洁了，实际上利用了回溯的思想，递归遍历到链表末尾，然后先交换末尾两个，然后依次往前交换：

解法二：

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *t = head->next;
        head->next = swapPairs(head->next->next);
        t->next = head;
        return t;
    }
};
```

25. Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.

Example:

Given this linked list: `1->2->3->4->5`

For *k* = 2, you should return: `2->1->4->3->5`

For *k* = 3, you should return: `3->2->1->4->5`

Note:

- Only constant extra memory is allowed.
- You may not alter the values in the list's nodes, only nodes itself may be changed.

这道题让我们以每k个为一组来翻转链表，实际上是把原链表分成若干小段，然后分别对其进行翻转，那么肯定总共需要两个函数，一个是用来分段的，一个是用来翻转的，以题目中给的例子来看，对于给定链表 1->2->3->4->5，一般在处理链表问题时，大多时候都会在开头再加一个 dummy node，因为翻转链表时头结点可能会变化，为了记录当前最新的头结点的位置而引入的 dummy node，加入 dummy node 后的链表变为 -1->1->2->3->4->5，如果k为3的话，目标是将 1,2,3 翻转一下，那么需要一些指针，pre 和 next 分别指向要翻转的链表的前后的位置，然后翻转后 pre 的位置更新到如下新的位置：

```
-1->1->2->3->4->5
 |        |  |
pre      cur next

-1->3->2->1->4->5
    |     |  |
   cur   pre next
```

以此类推，只要 cur 走过k个节点，那么 next 就是 cur->next，就可以调用翻转函数来进行局部翻转了，注意翻转之后新的 cur 和 pre 的位置都不同了，那么翻转之后，cur 应该更新为 pre->next，而如果不需要翻转的话，cur 更新为 cur->next，代码如下所示：

解法一：

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (!head || k == 1) return head;
        ListNode *dummy = new ListNode(-1), *pre = dummy, *cur = head;
        dummy->next = head;
        for (int i = 1; cur; ++i) {
            if (i % k == 0) {
                pre = reverseOneGroup(pre, cur->next);
                cur = pre->next;
            } else {
                cur = cur->next;
            }
        }
        return dummy->next;
    }
    ListNode* reverseOneGroup(ListNode* pre, ListNode* next) {
        ListNode *last = pre->next, *cur = last->next;
        while(cur != next) {
            last->next = cur->next;
            cur->next = pre->next;
            pre->next = cur;
            cur = last->next;
        }
        return last;
    }
};
```

我们也可以在一个函数中完成，首先遍历整个链表，统计出链表的长度，然后如果长度大于等于k，交换节点，当 k=2 时，每段只需要交换一次，当 k=3 时，每段需要交换2此，所以i从1开始循环，注意交换一段后更新 pre 指针，然后 num 自减k，直到 num<k 时循环结束，参见代码如下：

解法二：

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode *dummy = new ListNode(-1), *pre = dummy, *cur = pre;
        dummy->next = head;
        int num = 0;
        while (cur = cur->next) ++num;
        while (num >= k) {
            cur = pre->next;
            for (int i = 1; i < k; ++i) {
                ListNode *t = cur->next;
                cur->next = t->next;
                t->next = pre->next;
```

```
                pre->next = t;
            }
            pre = cur;
            num -= k;
        }
        return dummy->next;
    }
};
```

我们也可以使用递归来做，用 head 记录每段的开始位置，cur 记录结束位置的下一个节点，然后调用 reverse 函数来将这段翻转，然后得到一个 new_head，原来的 head 就变成了末尾，这时候后面接上递归调用下一段得到的新节点，返回 new_head 即可，参见代码如下：

解法三：

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode *cur = head;
        for (int i = 0; i < k; ++i) {
            if (!cur) return head;
            cur = cur->next;
        }
        ListNode *new_head = reverse(head, cur);
        head->next = reverseKGroup(cur, k);
        return new_head;
    }
    ListNode* reverse(ListNode* head, ListNode* tail) {
        ListNode *pre = tail;
        while (head != tail) {
            ListNode *t = head->next;
            head->next = pre;
            pre = head;
            head = t;
        }
        return pre;
    }
};
```

29. Divide Two Integers

Given two integers `dividend` and `divisor`, divide two integers without using multiplication, division and mod operator.

Return the quotient after dividing `dividend` by `divisor`.

The integer division should truncate toward zero.

Example 1:

```
Input: dividend = 10, divisor = 3
Output: 3
```

Example 2:

```
Input: dividend = 7, divisor = -3
Output: -2
```

Note:

- Both dividend and divisor will be 32-bit signed integers.
- The divisor will never be 0.
- Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231, 231 − 1]. For the purpose of this problem, assume that your function returns 231 − 1 when the division result overflows.

这道题让我们求两数相除，而且规定不能用乘法，除法和取余操作，那么这里可以用另一神器位操作 Bit Manipulation，思路是，如果被除数大于或等于除数，则进行如下循环，定义变量t等于除数，定义计数p，当t的两倍小于等于被除数时，进行如下循环，t扩大一倍，p扩大一倍，然后更新 res 和m。这道题的 OJ 给的一些 test case 非常的讨厌，因为输入的都是 int 型，比如被除数是 -2147483648，在 int 范围内，当除数是 -1 时，结果就超出了 int 范围，需要返回 INT_MAX，所以对于这种情况就在开始用 if 判定，将其和除数为0的情况放一起判定，返回 INT_MAX。然后还要根据被除数和除数的正负来确定返回值的正负，这里采用长整型 long 来完成所有的计算，最后返回值乘以符号即可，代码如下：

解法一：

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (dividend == INT_MIN && divisor == -1) return INT_MAX;
        long m = labs(dividend), n = labs(divisor), res = 0;
        int sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;
        if (n == 1) return sign == 1 ? m : -m;
        while (m >= n) {
            long t = n, p = 1;
            while (m >= (t << 1)) {
                t <<= 1;
                p <<= 1;
            }
            res += p;
            m -= t;
        }
        return sign == 1 ? res : -res;
    }
};
```

我们可以通过递归的方法来解使上面的解法变得更加简洁：

解法二：

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        long m = labs(dividend), n = labs(divisor), res = 0;
        if (m < n) return 0;
        long t = n, p = 1;
        while (m > (t << 1)) {
            t <<= 1;
            p <<= 1;
        }
        res += p + divide(m - t, n);
        if ((dividend < 0) ^ (divisor < 0)) res = -res;
        return res > INT_MAX ? INT_MAX : res;
    }
};
```

30. Substring with Concatenation of All Words

You are given a string, s, and a list of words, words, that are all of the same length. Find all starting indices of substring(s) in s that is a concatenation of each word in words exactly once and without any intervening characters.

Example 1:

```
Input:
  s = "barfoothefoobarman",
  words = ["foo","bar"]
Output: [0,9]
Explanation: Substrings starting at index 0 and 9 are "barfoor" and
"foobar" respectively.
The output order does not matter, returning [9,0] is fine too.
```

Example 2:

```
Input:
  s = "wordgoodgoodgoodbestword",
  words = ["word","good","best","word"]
Output: []
```

这道题让我们求串联所有单词的子串，就是说给定一个长字符串，再给定几个长度相同的单词，让找出串联给定所有单词的子串的起始位置，还是蛮有难度的一道题。假设 words 数组中有n个单词，每个单词的长度均为 len，那么实际上这道题就让我们出所有长度为 nxlen 的子串，使得其刚好是由 words 数组中的所有单词组成。那么就需要经常判断s串中长度为 len 的子串是否是 words 中的单词，为了快速

的判断，可以使用 HashMap，同时由于 words 数组可能有重复单词，就要用 HashMap 来建立所有的单词和其出现次数之间的映射，即统计每个单词出现的次数。

遍历s中所有长度为 nxlen 的子串，当剩余子串的长度小于 nxlen 时，就不用再判断了。所以i从0开始，到 (int)s.size() - nxlen 结束就可以了，注意这里一定要将 s.size() 先转为整型数，再进行解法。一定要形成这样的习惯，一旦 size() 后面要减去数字时，先转为 int 型，因为 size() 的返回值是无符号型，一旦减去一个比自己大的数字，则会出错。对于每个遍历到的长度为 nxlen 的子串，需要验证其是否刚好由 words 中所有的单词构成，检查方法就是每次取长度为 len 的子串，看其是否是 words 中的单词。为了方便比较，建立另一个 HashMap，当取出的单词不在 words 中，直接 break 掉，否则就将其在新的 HashMap 中的映射值加1，还要检测若其映射值超过原 HashMap 中的映射值，也 break 掉，因为就算当前单词在 words 中，但若其出现的次数超过 words 中的次数，还是不合题意的。在 for 循环外面，若j正好等于n，说明检测的n个长度为 len 的子串都是 words 中的单词，并且刚好构成了 words，则将当前位置i加入结果 res 即可，具体参见代码如下：

解法一：

```
class Solution {
public:
    vector<int> findSubstring(string s, vector<string>& words) {
        if (s.empty() || words.empty()) return {};
        vector<int> res;
        int n = words.size(), len = words[0].size();
        unordered_map<string, int> wordCnt;
        for (auto &word : words) ++wordCnt[word];
        for (int i = 0; i <= (int)s.size() - n * len; ++i) {
            unordered_map<string, int> strCnt;
            int j = 0;
            for (j = 0; j < n; ++j) {
                string t = s.substr(i + j * len, len);
                if (!wordCnt.count(t)) break;
                ++strCnt[t];
                if (strCnt[t] > wordCnt[t]) break;
            }
            if (j == n) res.push_back(i);
        }
        return res;
    }
};
```

这道题还有一种 O(n) 时间复杂度的解法，设计思路非常巧妙，但是感觉很难想出来，博主目测还未到达这种水平。这种方法不再是一个字符一个字符的遍历，而是一个词一个词的遍历，比如根据题目中的例子，字符串s的长度n为 18，words 数组中有两个单词 (cnt=2)，每个单词的长度 len 均为3，那么遍历的顺序为 0，3，6，8，12，15，然后偏移一个字符 1，4，7，9，13，16，然后再偏移一个字符 2，5，8，10，14，17，这样就可以把所有情况都遍历到，还是先用一个 HashMap m1 来记录 words 里的所有词，然后从0开始遍历，用 left 来记录左边界的位置，count 表示当前已经匹配的单词的个数。然后一个单词一个单词的遍历，如果当前遍历的到的单词t在 m1 中存在，那么将其加入另一个 HashMap m2 中，如果在 m2 中个数小于等于 m1 中的个数，那么 count 自增1，如果大于了，则需要做一些处理，比如下面这种情况：s = barfoofoo, words = {bar, foo, abc}，给 words 中新加了一个 abc ，目的是为了遍历到 barfoo 不会停止，当遍历到第二 foo 的时候, m2[foo]=2, 而此时

m1[foo]=1，这时候已经不连续了，所以要移动左边界 left 的位置，先把第一个词 t1=bar 取出来，然后将 m2[t1] 自减1，如果此时 m2[t1]<m1[t1] 了，说明一个匹配没了，那么对应的 count 也要自减1，然后左边界加上个 len，这样就可以了。如果某个时刻 count 和 cnt 相等了，说明成功匹配了一个位置，将当前左边界 left 存入结果 res 中，此时去掉最左边的一个词，同时 count 自减1，左边界右移 len，继续匹配。如果匹配到一个不在 m1 中的词，说明跟前面已经断开了，重置 m2，count 为0，左边界 left 移到 j+len，参见代码如下：

解法二：

```
class Solution {
public:
    vector<int> findSubstring(string s, vector<string>& words) {
        if (s.empty() || words.empty()) return {};
        vector<int> res;
        int n = s.size(), cnt = words.size(), len = words[0].size();
        unordered_map<string, int> m1;
        for (string w : words) ++m1[w];
        for (int i = 0; i < len; ++i) {
            int left = i, count = 0;
            unordered_map<string, int> m2;
            for (int j = i; j <= n - len; j += len) {
                string t = s.substr(j, len);
                if (m1.count(t)) {
                    ++m2[t];
                    if (m2[t] <= m1[t]) {
                        ++count;
                    } else {
                        while (m2[t] > m1[t]) {
                            string t1 = s.substr(left, len);
                            --m2[t1];
                            if (m2[t1] < m1[t1]) --count;
                            left += len;
                        }
                    }
                    if (count == cnt) {
                        res.push_back(left);
                        --m2[s.substr(left, len)];
                        --count;
                        left += len;
                    }
                } else {
                    m2.clear();
                    count = 0;
                    left = j + len;
                }
            }
        }
        return res;
    }
```

```
};
```

31. Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place and use only constant extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

这道题让我们求下一个排列顺序，由题目中给的例子可以看出来，如果给定数组是降序，则说明是全排列的最后一种情况，则下一个排列就是最初始情况，可以参见之前的博客 Permutations。再来看下面一个例子，有如下的一个数组

1 2 7 4 3 1

下一个排列为：

1 3 1 2 4 7

那么是如何得到的呢，我们通过观察原数组可以发现，如果从末尾往前看，数字逐渐变大，到了2时才减小的，然后再从后往前找第一个比2大的数字，是3，那么我们交换2和3，再把此时3后面的所有数字转置一下即可，步骤如下：

1 2 7 4 3 1

1 2 7 4 3 1

1 3 7 4 2 1

1 3 1 2 4 7

解法一：

```
class Solution {
public:
    void nextPermutation(vector<int> &num) {
        int i, j, n = num.size();
        for (i = n - 2; i >= 0; --i) {
            if (num[i + 1] > num[i]) {
                for (j = n - 1; j > i; --j) {
                    if (num[j] > num[i]) break;
```

```
                }
                swap(num[i], num[j]);
                reverse(num.begin() + i + 1, num.end());
                return;
            }
        }
        reverse(num.begin(), num.end());
    }
};
```

下面这种写法更简洁一些，但是整体思路和上面的解法没有什么区别，参见代码如下：

解法二：

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {int n = nums.size(), i = n - 2, j = n - 1;
        while (i >= 0 && nums[i] >= nums[i + 1]) --i;
        if (i >= 0) {
            while (nums[j] <= nums[i]) --j;
            swap(nums[i], nums[j]);
        }
        reverse(nums.begin() + i + 1, nums.end());
    }
};
```

32. Longest Valid Parentheses

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

Example 1:

```
Input: "(()"
Output: 2
Explanation: The longest valid parentheses substring is "()"
```

Example 2:

```
Input: ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()()"
```

这道求最长有效括号比之前那道 Valid Parentheses 难度要大一些，这里还是借助栈来求解，需要定义个 start 变量来记录合法括号串的起始位置，遍历字符串，如果遇到左括号，则将当前下标压入栈，如果遇到右括号，如果当前栈为空，则将下一个坐标位置记录到 start，如果栈不为空，则将栈顶元素取出，此时若栈为空，则更新结果和 i - start + 1 中的较大值，否则更新结果和 i - st.top() 中的较大值，参见代码如下：

解法一：

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int res = 0, start = 0, n = s.size();
        stack<int> st;
        for (int i = 0; i < n; ++i) {
            if (s[i] == '(') st.push(i);
            else if (s[i] == ')') {
                if (st.empty()) start = i + 1;
                else {
                    st.pop();
                    res = st.empty() ? max(res, i - start + 1) : max(res, i - st.top());
                }
            }
        }
        return res;
    }
};
```

还有一种利用动态规划 Dynamic Programming 的解法，可参见网友喜刷刷的博客。这里使用一个一维 dp 数组，其中 dp[i] 表示以 s[i-1] 结尾的最长有效括号长度（注意这里没有对应 s[i]，是为了避免取 dp[i-1] 时越界从而让 dp 数组的长度加了1），s[i-1] 此时必须是有效括号的一部分，那么只要 dp[i] 为正数的话，说明 s[i-1] 一定是右括号，因为有效括号必须是闭合的。当括号有重合时，比如 "(())"，会出现多个右括号相连，此时更新最外边的右括号的 dp[i] 时是需要前一个右括号的值 dp[i-1]，因为假如 dp[i-1] 为正数，说明此位置往前 dp[i-1] 个字符组成的子串都是合法的子串，需要再看前面一个位置，假如是左括号，说明在 dp[i-1] 的基础上又增加了一个合法的括号，所以长度加上2。但此时还可能出现的情况是，前面的左括号前面还有合法括号，比如 "()(())"，此时更新最后面的右括号的时候，知道第二个右括号的 dp 值是2，那么最后一个右括号的 dp 值不仅是第二个括号的 dp 值再加2，还可以连到第一个右括号的 dp 值，整个最长的有效括号长度是6。所以在更新当前右括号的 dp 值时，首先要计算出第一个右括号的位置，通过 i-3-dp[i-1] 来获得，由于这里定义的 dp[i] 对应的是字符 s[i-1]，所以需要再加1，变成 j = i-2-dp[i-1]，这样若当前字符 s[i-1] 是左括号，或者j小于0（说明没有对应的左括号），或者 s[j] 是右括号，此时将 dp[i] 重置为0，否则就用 dp[i-1] + 2 + dp[j] 来更新 dp[i]。这里由于进行了 padding，可能对应关系会比较晕，大家可以自行带个例子一步一步执行，应该是不难理解的，参见代码如下：

解法二：

```
class Solution {
public:
```

```
    int longestValidParentheses(string s) {
        int res = 0, n = s.size();
        vector<int> dp(n + 1);
        for (int i = 1; i <= n; ++i) {
            int j = i - 2 - dp[i - 1];
            if (s[i - 1] == '(' || j < 0 || s[j] == ')') {
                dp[i] = 0;
            } else {
                dp[i] = dp[i - 1] + 2 + dp[j];
                res = max(res, dp[i]);
            }
        }
        return res;
    }
};
```

此题还有一种不用额外空间的解法，使用了两个变量 left 和 right，分别用来记录到当前位置时左括号和右括号的出现次数，当遇到左括号时，left 自增1，右括号时 right 自增1。对于最长有效的括号的子串，一定是左括号等于右括号的情况，此时就可以更新结果 res 了，一旦右括号数量超过左括号数量了，说明当前位置不能组成合法括号子串，left 和 right 重置为0。但是对于这种情况 "(()" 时，在遍历结束时左右子括号数都不相等，此时没法更新结果 res，但其实正确答案是2，怎么处理这种情况呢？答案是再反向遍历一遍，采取类似的机制，稍有不同的是此时若 left 大于 right 了，则重置0，这样就可以 cover 所有的情况了，参见代码如下：

解法三：

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int res = 0, left = 0, right = 0, n = s.size();
        for (int i = 0; i < n; ++i) {
            (s[i] == '(') ? ++left : ++right;
            if (left == right) res = max(res, 2 * right);
            else if (right > left) left = right = 0;
        }
        left = right = 0;
        for (int i = n - 1; i >= 0; --i) {
            (s[i] == '(') ? ++left : ++right;
            if (left == right) res = max(res, 2 * left);
            else if (left > right) left = right = 0;
        }
        return res;
    }
};
```

33. Search in Rotated Sorted Array

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., `[0,1,2,4,5,6,7]` might become `[4,5,6,7,0,1,2]`).

You are given a target value to search. If found in the array return its index, otherwise return `-1`.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O* (log *n*).

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

这道题让在旋转数组中搜索一个给定值，若存在返回坐标，若不存在返回 -1。我们还是考虑二分搜索法，但是这道题的难点在于不知道原数组在哪旋转了，还是用题目中给的例子来分析，对于数组 [0 1 2 4 5 6 7] 共有下列七种旋转方法（红色表示中点之前或者之后一定为有序的）：

0 1 2 **4 5 6 7**

7 0 1 **2 4 5 6**

6 7 0 **1 2 4 5**

5 6 7 **0 1 2 4**

4 5 6 7 0 1 2

2 4 5 6 7 0 1

1 2 4 5 6 7 0

二分搜索法的关键在于获得了中间数后，判断下面要搜索左半段还是右半段，观察上面红色的数字都是升序的（Github 中可能无法显示颜色，参见博客园上的帖子），可以得出出规律，如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的，我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边了，代码如下：

解法一：

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
```

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

看了上面的解法，你可能会产生个疑问，为啥非得用中间的数字跟最右边的比较呢？难道跟最左边的数字比较不行吗，当中间的数字大于最左边的数字时，左半段也是有序的啊，如下所示（蓝色表示中点之前一定为有序的）：

0 1 2 4 5 6 7

7 0 1 2 4 5 6

6 7 0 1 2 4 5

5 6 7 0 1 2 4

4 5 6 7 0 1 2

2 4 5 6 7 0 1

1 2 4 5 6 7 0

貌似也可以做，但是有一个问题，那就是在二分搜索中，nums[mid] 和 nums[left] 还有可能相等的，当数组中只有两个数字的时候，比如 [3, 1]，那该去取那一边呢？由于只有两个数字且 nums[mid] 不等于 target，target 只有可能在右半边出现。最好的方法就是让其无法进入左半段，就需要左半段是有序的，而且由于一定无法同时满足 nums[left] <= target && nums[mid] > target，因为 nums[left] 和 nums[mid] 相等，同一个数怎么可能同时大于等于 target，又小于 target。由于这个条件不满足，则直接进入右半段继续搜索即可，所以等于的情况要加到 nums[mid] > nums[left] 的情况中，变成大于等于，参见代码如下：

解法二：

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            if (nums[mid] >= nums[left]) {
```

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

讨论：这道题的二分搜索的解法实际上是博主之前的总结帖 LeetCode Binary Search Summary 二分搜索法小结 中的第五类，也是必须要将 right 初始化为 nums.size()-1，且循环条件必须为小于等于。二分搜索法真是博大精深，变化莫测啊～

34. Find First and Last Position of Element in Sorted Array

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

Your algorithm's runtime complexity must be in the order of *O* (log *n*).

If the target is not found in the array, return `[-1, -1]`.

Example 1:

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

Example 2:

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

这道题让我们在一个有序整数数组中寻找相同目标值的起始和结束位置，而且限定了时间复杂度为O(logn)，这是典型的二分查找法的时间复杂度，所以这里也需要用此方法，思路是首先对原数组使用二分查找法，找出其中一个目标值的位置，然后向两边搜索找出起始和结束的位置，代码如下：

解法一：

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int idx = search(nums, 0, nums.size() - 1, target);
        if (idx == -1) return {-1, -1};
        int left = idx, right = idx;
```

```
        while (left > 0 && nums[left - 1] == nums[idx]) --left;
        while (right < nums.size() - 1 && nums[right + 1] == nums[idx]) ++right;
        return {left, right};
    }
    int search(vector<int>& nums, int left, int right, int target) {
        if (left > right) return -1;
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        if (nums[mid] < target) return search(nums, mid + 1, right, target);
        else return search(nums, left, mid - 1, target);
    }
};
```

可能有些人会觉得上面的算法不是严格意义上的 O(logn) 的算法，因为在最坏的情况下会变成 O(n)，比如当数组里的数全是目标值的话，从中间向两边找边界就会一直遍历完整个数组，那么下面来看一种真正意义上的 O(logn) 的算法，使用两次二分查找法，第一次找到左边界，第二次调用找到右边界即可，具体代码如下：

解法二：

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        vector<int> res(2, -1);
        int left = 0, right = nums.size();
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        if (right == nums.size() || nums[right] != target) return res;
        res[0] = right;
        right = nums.size();
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] <= target) left = mid + 1;
            else right = mid;
        }
        res[1] = right - 1;
        return res;
    }
};
```

其实我们也可以只使用一个二分查找的子函数，来同时查找出第一个和最后一个位置。如何只用查找第一个大于等于目标值的二分函数来查找整个范围呢，这里用到了一个小 trick，首先来查找起始位置的 target，就是在数组中查找第一个大于等于 target 的位置，当返回的位置越界，或者该位置上的值不等于 target 时，表示数组中没有 target，直接返回 {-1, -1} 即可。若查找到了 target 值，则再查找第一个

大于等于 target+1 的位置，然后把返回的位置减1，就是 target 的最后一个位置，即便是返回的值越界了，减1后也不会越界，这样就实现了使用一个二分查找函数来解题啦，参见代码如下：

解法三：

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int start = firstGreaterEqual(nums, target);
        if (start == nums.size() || nums[start] != target) return {-1, -1};
        return {start, firstGreaterEqual(nums, target + 1) - 1};
    }
    int firstGreaterEqual(vector<int>& nums, int target) {
        int left = 0, right = nums.size();
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        return right;
    }
};
```

35. Search Insert Position

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Example 1:

```
Input: [1,3,5,6], 5
Output: 2
```

Example 2:

```
Input: [1,3,5,6], 2
Output: 1
```

Example 3:

```
Input: [1,3,5,6], 7
Output: 4
```

Example 4:

```
Input: [1,3,5,6], 0
Output: 0
```

这道题基本没有什么难度，实在不理解为啥还是 Medium 难度的，完完全全的应该是 Easy 啊（貌似现在已经改为 Easy 类了），三行代码搞定的题，只需要遍历一遍原数组，若当前数字大于或等于目标值，则返回当前坐标，如果遍历结束了，说明目标值比数组中任何一个数都要大，则返回数组长度n即可，代码如下：

解法一：

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        for (int i = 0; i < nums.size(); ++i) {
            if (nums[i] >= target) return i;
        }
        return nums.size();
    }
};
```

当然，我们还可以用二分搜索法来优化时间复杂度，而且个人认为这种方法应该是面试官们想要考察的算法吧，属于博主之前的总结帖 LeetCode Binary Search Summary 二分搜索法小结 中第二类 - 查找不小于目标值的数，参见代码如下：

解法二：

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        if (nums.back() < target) return nums.size();
        int left = 0, right = nums.size();
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        return right;
    }
};
```

36. Valid Sudoku

Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.

3. Each of the 9 `3x3` sub-boxes of the grid must contain the digits `1-9` without repetition.

A partially filled sudoku which is valid.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

Example 1:

```
Input:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: true
```

Example 2:

```
Input:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner
being
    modified to 8. Since there are two 8's in the top left 3x3 sub-box, it
is invalid.
```

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.
- The given board contain only digits `1-9` and the character `'.'`.
- The given board size is always `9x9`.

这道题让验证一个方阵是否为数独矩阵，想必数独游戏我们都玩过，就是给一个 9x9 大小的矩阵，可以分为9个 3x3 大小的矩阵，要求是每个小矩阵内必须都是1到9的数字不能有重复，同时大矩阵的横纵列也不能有重复数字，是一种很经典的益智类游戏，经常在飞机上看见有人拿着纸笔在填数，感觉是消磨时光的利器。这道题给了一个残缺的二维数组，让我们判断当前的这个数独数组是否合法，即要满足上述的条件。判断标准是看各行各列是否有重复数字，以及每个小的 3x3 的小方阵里面是否有重复数字，如果都无重复，则当前矩阵是数独矩阵，但不代表待数独矩阵有解，只是单纯的判断当前未填完的矩阵是否是数独矩阵。那么根据数独矩阵的定义，在遍历每个数字的时候，就看看包含当前位置的行和列以及 3x3 小方阵中是否已经出现该数字，这里需要三个 boolean 型矩阵，大小跟原数组相同，分别记录各行，各列，各小方阵是否出现某个数字，其中行和列标志下标很好对应，就是小方阵的下标需要稍稍转换一下，具体代码如下：

解法一：

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        vector<vector<bool>> rowFlag(9, vector<bool>(9));
        vector<vector<bool>> colFlag(9, vector<bool>(9));
        vector<vector<bool>> cellFlag(9, vector<bool>(9));
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') continue;
                int c = board[i][j] - '1';
                if (rowFlag[i][c] || colFlag[c][j] || cellFlag[3 * (i / 3) + j / 3][c]) return false;
                rowFlag[i][c] = true;
                colFlag[c][j] = true;
                cellFlag[3 * (i / 3) + j / 3][c] = true;
            }
        }
        return true;
    }
};
```

我们也可以对空间进行些优化，只使用一个 HashSet 来记录已经存在过的状态，将每个状态编码成为一个字符串，能将如此大量信息的状态编码成一个单一的字符串还是需要有些技巧的。对于每个1到9内的数字来说，其在每行每列和每个小区间内都是唯一的，将数字放在一个括号中，每行上的数字就将行号放在括号左边，每列上的数字就将列数放在括号右边，每个小区间内的数字就将在小区间内的行列数分别放在括号的左右两边，这样每个数字的状态都是独一无二的存在，就可以在 HashSet 中愉快地查找是否有重复存在啦，参见代码如下：

解法二：

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        unordered_set<string> st;
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
```

```
                if (board[i][j] == '.') continue;
                string t = "(" + to_string(board[i][j]) + ")";
                string row = to_string(i) + t, col = t + to_string(j),
cell = to_string(i / 3) + t + to_string(j / 3);
                if (st.count(row) || st.count(col) || st.count(cell))
return false;
                st.insert(row);
                st.insert(col);
                st.insert(cell);
            }
        }
        return true;
    }
};
```

37. Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

Empty cells are indicated by the character `'.'`.

A sudoku puzzle...

...and its solution numbers marked in red.

Note:

```
* The given board contain only digits `1-9`and the character `'.'`.
* You may assume that the given Sudoku puzzle will have a single unique
solution.
* The given board size is always `9x9`.
```

这道求解数独的题是在之前那道 Valid Sudoku 的基础上的延伸，之前那道题让我们验证给定的数组是否为数独数组，这道让求解数独数组，跟此题类似的有 Permutations，Combinations， N-Queens 等等，其中尤其是跟 N-Queens 的解题思路及其相似，对于每个需要填数字的格子带入1到9，每代入一个数字都判定其是否合法，如果合法就继续下一次递归，结束时把数字设回 '.'，判断新加入的数字是否合法时，只需要判定当前数字是否合法，不需要判定这个数组是否为数独数组，因为之前加进的数字都是合法的，这样可以使程序更加高效一些，整体思路是这样的，但是实现起来可以有不同的形式。一种实现形式是递归带上横纵坐标，由于是一行一行的填数字，且是从0行开始的，所以当i到达9的时候，说

明所有的数字都成功的填入了，直接返回 ture。当j大于等于9时，当前行填完了，需要换到下一行继续填，则继续调用递归函数，横坐标带入 i+1。否则看若当前数字不为点，说明当前位置不需要填数字，则对右边的位置调用递归。若当前位置需要填数字，则应该尝试填入1到9内的所有数字，让c从1遍历到9，每当试着填入一个数字，都需要检验是否有冲突，使用另一个子函数 isValid 来检验是否合法，假如不合法，则跳过当前数字。若合法，则将当前位置赋值为这个数字，并对右边位置调用递归，若递归函数返回 true，则说明可以成功填充，直接返回 true。不行的话，需要重置状态，将当前位置恢复为点。若所有数字都尝试了，还是不行，则最终返回 false。检测当前数组是否合法的原理跟之前那道 Valid Sudoku 非常的相似，但更简单一些，因为这里只需要检测新加入的这个数字是否会跟其他位置引起冲突，分别检测新加入数字的行列和所在的小区间内是否有重复的数字即可，参见代码如下：

解法一：

```
class Solution {
public:
    void solveSudoku(vector<vector<char>>& board) {
        helper(board, 0, 0);
    }
    bool helper(vector<vector<char>>& board, int i, int j) {
        if (i == 9) return true;
        if (j >= 9) return helper(board, i + 1, 0);
        if (board[i][j] != '.') return helper(board, i, j + 1);
        for (char c = '1'; c <= '9'; ++c) {
            if (!isValid(board, i , j, c)) continue;
            board[i][j] = c;
            if (helper(board, i, j + 1)) return true;
            board[i][j] = '.';
        }
        return false;
    }
    bool isValid(vector<vector<char>>& board, int i, int j, char val) {
        for (int x = 0; x < 9; ++x) {
            if (board[x][j] == val) return false;
        }
        for (int y = 0; y < 9; ++y) {
            if (board[i][y] == val) return false;
        }
        int row = i - i % 3, col = j - j % 3;
        for (int x = 0; x < 3; ++x) {
            for (int y = 0; y < 3; ++y) {
                if (board[x + row][y + col] == val) return false;
            }
        }
        return true;
    }
};
```

还有另一种递归的写法，这里就不带横纵坐标参数进去，由于递归需要有 boolean 型的返回值，所以不能使用原函数。因为没有横纵坐标，所以每次遍历都需要从开头0的位置开始，这样无形中就有了大量的重复检测，导致这种解法虽然写法简洁一些，但击败率是没有上面的解法高的。这里的检测数组冲突的子函数写法也比上面简洁不少，只用了一个 for 循环，用来同时检测行列和小区间是否有冲突，注意正确的坐标转换即可，参见代码如下：

解法二：

```
class Solution {
public:
    void solveSudoku(vector<vector<char>>& board) {
        helper(board);
    }
    bool helper(vector<vector<char>>& board) {
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] != '.') continue;
                for (char c = '1'; c <= '9'; ++c) {
                    if (!isValid(board, i, j, c)) continue;
                    board[i][j] = c;
                    if (helper(board)) return true;
                    board[i][j] = '.';
                }
                return false;
            }
        }
        return true;
    }
    bool isValid(vector<vector<char>>& board, int i, int j, char val) {
        for (int k = 0; k < 9; ++k) {
            if (board[k][j] != '.' && board[k][j] == val) return false;
            if (board[i][k] != '.' && board[i][k] == val) return false;
            int row = i / 3 * 3 + k / 3, col = j / 3 * 3 + k % 3;
            if (board[row][col] != '.' && board[row][col] == val) return
false;
        }
        return true;
    }
};
```

39. Combination Sum

Given a set of candidate numbers (`candidates`) (without duplicates) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sums to `target`.

The same repeated number may be chosen from `candidates` unlimited number of times.

Note:

- All numbers (including `target`) will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7,
A solution set is:
[
  [7],
  [2,2,3]
]
```

Example 2:

```
Input: candidates = [2,3,5], target = 8,
A solution set is:
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

像这种结果要求返回所有符合要求解的题十有八九都是要利用到递归，而且解题的思路都大同小异，相类似的题目有 Path Sum II，Subsets II，Permutations，Permutations II，Combinations 等等，如果仔细研究这些题目发现都是一个套路，都是需要另写一个递归函数，这里我们新加入三个变量，start 记录当前的递归到的下标，out 为一个解，res 保存所有已经得到的解，每次调用新的递归函数时，此时的 target 要减去当前数组的的数，具体看代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        vector<int> out;
        combinationSumDFS(candidates, target, 0, out, res);
        return res;
    }
    void combinationSumDFS(vector<int>& candidates, int target, int start, vector<int>& out, vector<vector<int>>& res) {
        if (target < 0) return;
        if (target == 0) {res.push_back(out); return;}
        for (int i = start; i < candidates.size(); ++i) {
            out.push_back(candidates[i]);
            combinationSumDFS(candidates, target - candidates[i], i, out, res);
```

```
            out.pop_back();
        }
    }
};
```

我们也可以不使用额外的函数，就在一个函数中完成递归，还是要先给数组排序，然后遍历，如果当前数字大于 target，说明肯定无法组成 target，由于排过序，之后的也无法组成 target，直接 break 掉。如果当前数字正好等于 target，则当前单个数字就是一个解，组成一个数组然后放到结果 res 中。然后将当前位置之后的数组取出来，调用递归函数，注意此时的 target 要减去当前的数字，然后遍历递归结果返回的二维数组，将当前数字加到每一个数组最前面，然后再将每个数组加入结果 res 即可，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        sort(candidates.begin(), candidates.end());
        for (int i = 0; i < candidates.size(); ++i) {
            if (candidates[i] > target) break;
            if (candidates[i] == target) {res.push_back({candidates[i]}); break;}
            vector<int> vec = vector<int>(candidates.begin() + i, candidates.end());
            vector<vector<int>> tmp = combinationSum(vec, target - candidates[i]);
            for (auto a : tmp) {
                a.insert(a.begin(), candidates[i]);
                res.push_back(a);
            }
        }
        return res;
    }
};
```

我们也可以用迭代的解法来做，建立一个三维数组 dp，这里 dp[i] 表示目标数为 i+1 的所有解法集合。这里的i就从1遍历到 target 即可，对于每个i，都新建一个二维数组 cur，然后遍历 candidates 数组，如果遍历到的数字大于i，说明当前及之后的数字都无法组成i，直接 break 掉。否则如果相等，那么把当前数字自己组成一个数组，并且加到 cur 中。否则就遍历 dp[i - candidates[j] - 1] 中的所有数组，如果当前数字大于数组的首元素，则跳过，因为结果要求是要有序的。否则就将当前数字加入数组的开头，并且将数组放入 cur 之中即可，参见代码如下：

解法三：

```
class Solution {
public:
```

```
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<vector<int>>> dp;
        sort(candidates.begin(), candidates.end());
        for (int i = 1; i <= target; ++i) {
            vector<vector<int>> cur;
            for (int j = 0; j < candidates.size(); ++j) {
                if (candidates[j] > i) break;
                if (candidates[j] == i) {cur.push_back({candidates[j]}); break;}
                for (auto a : dp[i - candidates[j] - 1]) {
                    if (candidates[j] > a[0]) continue;
                    a.insert(a.begin(), candidates[j]);
                    cur.push_back(a);
                }
            }
            dp.push_back(cur);
        }
        return dp[target - 1];
    }
};
```

40. Combination Sum II

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sums to `target`.

Each number in `candidates` may only be used once in the combination.

Note:

- All numbers (including `target`) will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [10,1,2,7,6,1,5], target = 8,
A solution set is:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

Example 2:

```
Input: candidates = [2,5,2,1,2], target = 5,
A solution set is:
[
  [1,2,2],
  [5]
]
```

这道题跟之前那道 Combination Sum 本质没有区别，只需要改动一点点即可，之前那道题给定数组中的数字可以重复使用，而这道题不能重复使用，只需要在之前的基础上修改两个地方即可，首先在递归的 for 循环里加上 if (i > start && num[i] == num[i - 1]) continue; 这样可以防止 res 中出现重复项，然后就在递归调用 helper 里面的参数换成 i+1，这样就不会重复使用数组中的数字了，代码如下：

```
class Solution {
public:
    vector<vector<int>> combinationSum2(vector<int>& num, int target) {
        vector<vector<int>> res;
        vector<int> out;
        sort(num.begin(), num.end());
        helper(num, target, 0, out, res);
        return res;
    }
    void helper(vector<int>& num, int target, int start, vector<int>& out, vector<vector<int>>& res) {
        if (target < 0) return;
        if (target == 0) { res.push_back(out); return; }
        for (int i = start; i < num.size(); ++i) {
            if (i > start && num[i] == num[i - 1]) continue;
            out.push_back(num[i]);
            helper(num, target - num[i], i + 1, out, res);
            out.pop_back();
        }
    }
};
```

41. First Missing Positive

Given an unsorted integer array, find the smallest missing positive integer.

Example 1:

```
Input: [1,2,0]
Output: 3
```

Example 2:

```
Input: [3,4,-1,1]
Output: 2
```

Example 3:

```
Input: [7,8,9,11,12]
Output: 1
```

Note:

Your algorithm should run in *O* (*n*) time and uses constant extra space.

这道题让我们找缺失的首个正数，由于限定了 O(n) 的时间，所以一般的排序方法都不能用，最开始博主没有看到还限制了空间复杂度，所以想到了用 HashSet 来解，这个思路很简单，把所有的数都存入 HashSet 中，然后循环从1开始递增找数字，哪个数字找不到就返回哪个数字，如果一直找到了最大的数字（这里是 nums 数组的长度），则加1后返回结果 res，参见代码如下：

解法一：

```
// NOT constant space
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        unordered_set<int> st(nums.begin(), nums.end());
        int res = 1, n = nums.size();
        while (res <= n) {
            if (!st.count(res)) return res;
            ++res;
        }
        return res;
    }
};
```

但是上面的解法不是 O(1) 的空间复杂度，所以需要另想一种解法，既然不能建立新的数组，那么只能覆盖原有数组，思路是把1放在数组第一个位置 nums[0]，2放在第二个位置 nums[1]，即需要把 nums[i] 放在 nums[nums[i] - 1]上，遍历整个数组，如果 nums[i] != i + 1, 而 nums[i] 为整数且不大于 n，另外 nums[i] 不等于 nums[nums[i] - 1] 的话，将两者位置调换，如果不满足上述条件直接跳过，最后再遍历一遍数组，如果对应位置上的数不正确则返回正确的数，参见代码如下：

解法二：

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n; ++i) {
            while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] !=
nums[i]) {
                swap(nums[i], nums[nums[i] - 1]);
```

```
            }
        }
        for (int i = 0; i < n; ++i) {
            if (nums[i] != i + 1) return i + 1;
        }
        return n + 1;
    }
};
```

42. Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

Example:

```
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
```

这道收集雨水的题跟之前的那道 Largest Rectangle in Histogram 有些类似，但是又不太一样，先来看一种方法，这种方法是基于动态规划 Dynamic Programming 的，维护一个一维的 dp 数组，这个 DP 算法需要遍历两遍数组，第一遍在 dp[i] 中存入i位置左边的最大值，然后开始第二遍遍历数组，第二次遍历时找右边最大值，然后和左边最大值比较取其中的较小值，然后跟当前值 A[i] 相比，如果大于当前值，则将差值存入结果，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    int trap(vector<int>& height) {
        int res = 0, mx = 0, n = height.size();
        vector<int> dp(n, 0);
        for (int i = 0; i < n; ++i) {
            dp[i] = mx;
            mx = max(mx, height[i]);
        }
        mx = 0;
        for (int i = n - 1; i >= 0; --i) {
            dp[i] = min(dp[i], mx);
            mx = max(mx, height[i]);
            if (dp[i] > height[i]) res += dp[i] - height[i];
        }
        return res;
```

```
    }
};
```

Java 解法一：

```
public class Solution {
    public int trap(int[] height) {
        int res = 0, mx = 0, n = height.length;
        int[] dp = new int[n];
        for (int i = 0; i < n; ++i) {
            dp[i] = mx;
            mx = Math.max(mx, height[i]);
        }
        mx = 0;
        for (int i = n - 1; i >= 0; --i) {
            dp[i] = Math.min(dp[i], mx);
            mx = Math.max(mx, height[i]);
            if (dp[i] - height[i] > 0) res += dp[i] - height[i];
        }
        return res;
    }
}
```

再看一种只需要遍历一次即可的解法，这个算法需要 left 和 right 两个指针分别指向数组的首尾位置，从两边向中间扫描，在当前两指针确定的范围内，先比较两头找出较小值，如果较小值是 left 指向的值，则从左向右扫描，如果较小值是 right 指向的值，则从右向左扫描，若遇到的值比当较小值小，则将差值存入结果，如遇到的值大，则重新确定新的窗口范围，以此类推直至 left 和 right 指针重合，参见代码如下：

C++ 解法二：

```
class Solution {
public:
    int trap(vector<int>& height) {
        int res = 0, l = 0, r = height.size() - 1;
        while (l < r) {
            int mn = min(height[l], height[r]);
            if (mn == height[l]) {
                ++l;
                while (l < r && height[l] < mn) {
                    res += mn - height[l++];
                }
            } else {
                --r;
                while (l < r && height[r] < mn) {
                    res += mn - height[r--];
                }
            }
```

```
        }
        return res;
    }
};
```

Java 解法二：

```
public class Solution {
    public int trap(int[] height) {
        int res = 0, l = 0, r = height.length - 1;
        while (l < r) {
            int mn = Math.min(height[l], height[r]);
            if (height[l] == mn) {
                ++l;
                while (l < r && height[l] < mn) {
                    res += mn - height[l++];
                }
            } else {
                --r;
                while (l < r && height[r] < mn) {
                    res += mn - height[r--];
                }
            }
        }
        return res;
    }
}
```

我们可以对上面的解法进行进一步优化，使其更加简洁：

C++ 解法三：

```
class Solution {
public:
    int trap(vector<int>& height) {
        int l = 0, r = height.size() - 1, level = 0, res = 0;
        while (l < r) {
            int lower = height[(height[l] < height[r]) ? l++ : r--];
            level = max(level, lower);
            res += level - lower;
        }
        return res;
    }
};
```

Java 解法三：

```java
public class Solution {
    public int trap(int[] height) {
        int l = 0, r = height.length - 1, level = 0, res = 0;
        while (l < r) {
            int lower = height[(height[l] < height[r]) ? l++ : r--];
            level = Math.max(level, lower);
            res += level - lower;
        }
        return res;
    }
}
```

下面这种解法是用 stack 来做的，博主一开始都没有注意到这道题的 tag 还有 stack，所以以后在总结的时候还是要多多留意一下标签啊。其实用 stack 的方法博主感觉更容易理解，思路是，遍历高度，如果此时栈为空，或者当前高度小于等于栈顶高度，则把当前高度的坐标压入栈，注意这里不直接把高度压入栈，而是把坐标压入栈，这样方便在后来算水平距离。当遇到比栈顶高度大的时候，就说明有可能会有坑存在，可以装雨水。此时栈里至少有一个高度，如果只有一个的话，那么不能形成坑，直接跳过，如果多余一个的话，那么此时把栈顶元素取出来当作坑，新的栈顶元素就是左边界，当前高度是右边界，只要取二者较小的，减去坑的高度，长度就是右边界坐标减去左边界坐标再减1，二者相乘就是盛水量啦，参见代码如下：

C++ 解法四：

```java
class Solution {
public:
    int trap(vector<int>& height) {
        stack<int> st;
        int i = 0, res = 0, n = height.size();
        while (i < n) {
            if (st.empty() || height[i] <= height[st.top()]) {
                st.push(i++);
            } else {
                int t = st.top(); st.pop();
                if (st.empty()) continue;
                res += (min(height[i], height[st.top()]) - height[t]) * (i - st.top() - 1);
            }
        }
        return res;
    }
};
```

Java 解法四：

```java
class Solution {
    public int trap(int[] height) {
        Stack<Integer> s = new Stack<Integer>();
        int i = 0, n = height.length, res = 0;
```

```
        while (i < n) {
            if (s.isEmpty() || height[i] <= height[s.peek()]) {
                s.push(i++);
            } else {
                int t = s.pop();
                if (s.isEmpty()) continue;
                res += (Math.min(height[i], height[s.peek()]) - height[t]) * (i - s.peek() - 1);
            }
        }
        return res;
    }
}
```

43. Multiply Strings

Given two non-negative integers `num1` and `num2` represented as strings, return the product of `num1` and `num2`, also represented as a string.

Example 1:

```
Input: num1 = "2", num2 = "3"
Output: "6"
```

Example 2:

```
Input: num1 = "123", num2 = "456"
Output: "56088"
```

Note:

1. The length of both `num1` and `num2` is < 110.
2. Both `num1` and `num2` contain only digits `0-9`.
3. Both `num1` and `num2` do not contain any leading zero, except the number 0 itself.
4. You must not use any built-in BigInteger libraryor convert the inputs to integer directly.

这道题让我们求两个字符串数字的相乘，输入的两个数和返回的数都是以字符串格式储存的，这样做的原因可能是这样可以计算超大数相乘，可以不受 int 或 long 的数值范围的约束，那么该如何来计算乘法呢，小时候都学过多位数的乘法过程，都是每位相乘然后错位相加，那么这里就是用到这种方法，举个例子，比如 89 x 76，那么根据小学的算术知识，不难写出计算过程如下：

```
    8 9  <- num2
    7 6  <- num1
-------
    5 4
  4 8
  6 3
5 6
-------
6 7 6 4
```

如果自己再写些例子出来，不难发现，两数相乘得到的乘积的长度其实其实不会超过两个数字的长度之和，若 num1 长度为m，num2 长度为n，则 num1 x num2 的长度不会超过 m+n，还有就是要明白乘的时候为什么要错位，比如6乘8得到的 48 为啥要跟6乘9得到的 54 错位相加，因为8是十位上的数字，其本身相当于80，所以错开的一位实际上末尾需要补的0。还有一点需要观察出来的就是，num1 和 num2 中任意位置的两个数字相乘，得到的两位数在最终结果中的位置是确定的，比如 num1 中位置为 i的数字乘以 num2 中位置为j的数字，那么得到的两位数字的位置为 i+j 和 i+j+1，明白了这些后，就可以进行错位相加了，累加出最终的结果。

由于要从个位上开始相乘，所以从 num1 和 num2 字符串的尾部开始往前遍历，分别提取出对应位置上的字符，将其转为整型后相乘。然后确定相乘后的两位数所在的位置 p1 和 p2，由于 p2 相较于 p1 是低位，所以将得到的两位数 mul 先加到 p2 位置上去，这样可能会导致 p2 位上的数字大于9，所以将十位上的数字要加到高位 p1 上去，只将余数留在 p2 位置，这样每个位上的数字都变成一位。然后要做的是从高位开始，将数字存入结果 res 中，记住 leading zeros 要跳过，最后处理下 corner case，即若结果 res 为空，则返回 "0"，否则返回结果 res，代码如下：

```
class Solution {
public:
    string multiply(string num1, string num2) {
        string res = "";
        int m = num1.size(), n = num2.size();
        vector<int> vals(m + n);
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                int mul = (num1[i] - '0') * (num2[j] - '0');
                int p1 = i + j, p2 = i + j + 1, sum = mul + vals[p2];
                vals[p1] += sum / 10;
                vals[p2] = sum % 10;
            }
        }
        for (int val : vals) {
            if (!res.empty() || val != 0) res.push_back(val + '0');
        }
        return res.empty() ? "0" : res;
    }
};
```

44. Wildcard Matching

Given an input string (`s`) and a pattern (`p`), implement wildcard pattern matching with support for `'?'` and `'*'`.

```
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
```

The matching should cover the entire input string (not partial).

Note:

- `s` could be empty and contains only lowercase letters `a-z`.
- `p` could be empty and contains only lowercase letters `a-z`, and characters like `?` or `*`.

Example 1:

```
Input:
s = "aa"
p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input:
s = "aa"
p = "*"
Output: true
Explanation: '*' matches any sequence.
```

Example 3:

```
Input:
s = "cb"
p = "?a"
Output: false
Explanation: '?' matches 'c', but the second letter is 'a', which does not
match 'b'.
```

Example 4:

```
Input:
s = "adceb"
p = "*a*b"
Output: true
Explanation: The first '*' matches the empty sequence, while the second '*'
matches the substring "dce".
```

Example 5:

```
Input:
s = "acdcb"
p = "a*c?b"
Output: false
```

这道题通配符外卡匹配问题还是小有难度的，有特殊字符 *'* *和* *'?'，其中* *'?'* *能代替任何字符，* *'* 能代替任何字符串，注意跟另一道 Regular Expression Matching 正则匹配的题目区分开来。两道题的星号的作用是不同的，注意对比区分一下。这道题最大的难点，就是对于星号的处理，可以匹配任意字符串，简直像开了挂一样，就是说在星号对应位置之前，不管你s中有任何字符串，我大星号都能匹配你，主角光环啊。但即便叼如斯的星号，也有其处理不了的问题，那就是一旦p中有s中不存在的字符，那么一定无法匹配，因为星号只能增加字符，不能消除字符，再有就是星号一旦确定了要匹配的字符串，对于星号位置后面的匹配情况也就鞭长莫及了。所以p串中星号的位置很重要，用 jStar 来表示，还有星号匹配到s串中的位置，使用 iStart 来表示，这里 iStar 和 jStar 均初始化为 -1，表示默认情况下是没有星号的。然后再用两个变量i和j分别指向当前s串和p串中遍历到的位置。

开始进行匹配，若i小于s串的长度，进行 while 循环。若当前两个字符相等，或着p中的字符是问号，则i和j分别加1。若 p[j] 是星号，要记录星号的位置，jStar 赋为j，此时j再自增1，iStar 赋为i。若当前 p[j] 不是星号，并且不能跟 p[i] 匹配上，此时就要靠星号了，若之前星号没出现过，那么就直接跪，比如 s = "aa" 和 p = "c"，*此时 s[0] 和 p[0] 无法匹配，虽然 p[1] 是星号，但还是跪。如果星号之前出现过，可以强行续一波命，比如 s = "aa" 和 p =* "c"，当发现 s[1] 和 p[1] 无法匹配时，但是好在之前 p[0] 出现了星号，把 s[1] 交给 p[0] 的星号去匹配。至于如何知道之前有没有星号，这时就能看出 iStar 的作用了，因为其初始化为 -1，而遇到星号时，其就会被更新为i，只要检测 iStar 的值，就能知道是否可以使用星号续命。虽然成功续了命，匹配完了s中的所有字符，但是之后还要检查p串，此时没匹配完的p串里只能剩星号，不能有其他的字符，将连续的星号过滤掉，如果j不等于p的长度，则返回 false，参见代码如下：

解法一：

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int i = 0, j = 0, iStar = -1, jStar = -1, m = s.size(), n =
p.size();
        while (i < m) {
            if (j < n && (s[i] == p[j] || p[j] == '?')) {
                ++i; ++j;
            } else if (j < n && p[j] == '*') {
                iStar = i;
                jStar = j++;
            } else if (iStar >= 0) {
                i = ++iStar;
                j = jStar + 1;
            } else return false;
        }
        while (j < n && p[j] == '*') ++j;
        return j == n;
```

```
    }
};
```

这道题也能用动态规划 Dynamic Programming 来解，写法跟之前那道题 Regular Expression Matching 很像，但是还是不一样。外卡匹配和正则匹配最大的区别就是在星号的使用规则上，对于正则匹配来说，星号不能单独存在，前面必须要有一个字符，而星号存在的意义就是表明前面这个字符的个数可以是任意个，包括0个，那么就是说即使前面这个字符并没有在s中出现过也无所谓，只要后面的能匹配上就可以了。而外卡匹配就不是这样的，外卡匹配中的星号跟前面的字符没有半毛钱关系，如果前面的字符没有匹配上，那么直接返回 false 了，根本不用管星号。而星号存在的作用是可以表示任意的字符串，当然只是当匹配字符串缺少一些字符的时候起作用，当匹配字符串p包含目标字符串s中没有的字符时，将无法成功匹配。

对于这种玩字符串的题目，动态规划 Dynamic Programming 是一大神器，因为字符串跟其子串之间的关系十分密切，正好适合 DP 这种靠推导状态转移方程的特性。那么先来定义dp数组吧，使用一个二维 dp 数组，其中 dp[i][j] 表示 s中前i个字符组成的子串和p中前j个字符组成的子串是否能匹配。大小初始化为 (m+1) x (n+1)，加1的原因是要包含 dp[0][0] 的情况，因为若s和p都为空的话，也应该返回 true，所以也要初始化 dp[0][0] 为 true。还需要提前处理的一种情况是，当s为空，p为连续的星号时的情况。由于星号是可以代表空串的，所以只要s为空，那么连续的星号的位置都应该为 true，所以先将连续星号的位置都赋为 true。然后就是推导一般的状态转移方程了，如何更新 dp[i][j]，首先处理比较 tricky 的情况，若p中第j个字符是星号，由于星号可以匹配空串，所以如果p中的前 j-1 个字符跟s中前i个字符匹配成功了（ dp[i][j-1] 为true）的话，则 dp[i][j] 也能为 true。或者若p中的前j个字符跟s中的前i-1个字符匹配成功了（ dp[i-1][j] 为true ）的话，则 dp[i][j] 也能为 true（因为星号可以匹配任意字符串，再多加一个任意字符也没问题）。若p中的第j个字符不是星号，对于一般情况，假设已经知道了s中前 i-1 个字符和p中前 j-1 个字符的匹配情况（即 dp[i-1][j-1] ），现在只需要匹配s中的第i个字符跟p中的第j个字符，若二者相等（ s[i-1] == p[j-1] ），或者p中的第j个字符是问号（ p[j-1] == '?' ），再与上 dp[i-1][j-1] 的值，就可以更新 dp[i][j] 了，参见代码如下：

解法二：

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size(), n = p.size();
        vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
        dp[0][0] = true;
        for (int i = 1; i <= n; ++i) {
            if (p[i - 1] == '*') dp[0][i] = dp[0][i - 1];
        }
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (p[j - 1] == '*') {
                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
                } else {
                    dp[i][j] = (s[i - 1] == p[j - 1] || p[j - 1] == '?')
&& dp[i - 1][j - 1];
                }
            }
        }
        return dp[m][n];
```

```
    }
};
```

其实这道题也可以使用递归来做，因为子串或者子数组这种形式，天然适合利用递归来做。但是愣了吧唧的递归跟暴力搜索并没有啥太大的区别，很容易被 OJ 毙掉，比如评论区六楼的那个 naive 的递归，其实完全是按照题目要求来的。首先判断s串，若为空，那么再看p串，若p为空，则为 true，或者跳过星号，继续调用递归。若s串不为空，且p串为空，则直接 false。若s串和p串均不为空，进行第一个字符的匹配，若相等，或者 p[0] 是问号，则跳过首字符，对后面的子串调用递归。若 p[0] 是星号，先尝试跳过s串的首字符，调用递归，若递归返回 true，则当前返回 true。否则尝试跳过p串的首字符，调用递归，若递归返回 true，则当前返回 true。但是很不幸，内存超出限制了 MLE，那么博主做了个简单的优化，跳过了连续的星号，参见评论区七楼的代码，但是这次时间超出了限制 TLE。博主想是不是取子串 substr() 操作太费时间，且调用递归的适合s串和p串又分别建立了副本，才导致的 TLE。于是想着用坐标变量来代替取子串，并且递归函数调用的s串和p串都加上引用，代码参见评论区八楼，但尼玛还是跪了，OJ 大佬，刀下留人啊。最后还是在论坛上找到了一个使用了神奇的剪枝的方法，这种解法的递归函数返回类型不是 bool 型，而是整型，有三种不同的状态，返回0表示匹配到了s串的末尾，但是未匹配成功；返回1表示未匹配到s串的末尾就失败了；返回2表示成功匹配。那么只有返回值大于1，才表示成功匹配。至于为何失败的情况要分类，就是为了进行剪枝。在递归函数中，若s串和p串都匹配完成了，返回状态2。若s串匹配完成了，但p串但当前字符不是星号，返回状态0。若s串未匹配完，p串匹配完了，返回状态1。若s串和p串均为匹配完，且当前字符成功匹配的话，对下一个位置调用递归。否则若p串当前字符是星号，首先跳过连续的星号。然后分别让星号匹配空串，一个字符，两个字符，....，直到匹配完整个s串，对每种情况分别调用递归函数，接下来就是最大的亮点了，也是最有用的剪枝，当前返回值为状态0或者2的时候，返回，否则继续遍历。如果仅仅是状态2的时候才返回，就像评论区八楼的代码，会有大量的重复计算，因为当返回值为状态0的时候，已经没有继续循环下去的必要了，非常重要的一刀剪枝，参见代码如下：

解法三：

```
class Solution {
public:
    bool isMatch(string s, string p) {
        return helper(s, p, 0, 0) > 1;
    }
    int helper(string& s, string& p, int i, int j) {
        if (i == s.size() && j == p.size()) return 2;
        if (i == s.size() && p[j] != '*') return 0;
        if (j == p.size()) return 1;
        if (s[i] == p[j] || p[j] == '?') {
            return helper(s, p, i + 1, j + 1);
        }
        if (p[j] == '*') {
            if (j + 1 < p.size() && p[j + 1] == '*') {
                return helper(s, p, i, j + 1);
            }
            for (int k = 0; k <= (int)s.size() - i; ++k) {
                int res = helper(s, p, i + k, j + 1);
                if (res == 0 || res == 2) return res;
            }
        }
```

```
        return 1;
    }
};
```

45. Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

Example:

```
Input: [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2.
    Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Note:

You can assume that you can always reach the last index.

这题是之前那道 Jump Game 的延伸，那题是问能不能到达最后一个数字，而此题只让求到达最后一个位置的最少跳跃数，貌似是默认一定能到达最后位置的? 此题的核心方法是利用贪婪算法 Greedy 的思想来解，想想为什么呢？ 为了较快的跳到末尾，想知道每一步能跳的范围，这里贪婪并不是要在能跳的范围中选跳力最远的那个位置，因为这样选下来不一定是最优解，这么一说感觉又有点不像贪婪算法了。其实这里贪的是一个能到达的最远范围，遍历当前跳跃能到的所有位置，然后根据该位置上的跳力来预测下一步能跳到的最远距离，贪出一个最远的范围，一旦当这个范围到达末尾时，当前所用的步数一定是最小步数。需要两个变量 cur 和 pre 分别来保存当前的能到达的最远位置和之前能到达的最远位置，只要 cur 未达到最后一个位置则循环继续，pre 先赋值为 cur 的值，表示上一次循环后能到达的最远位置，如果当前位置i小于等于 pre，说明还是在上一跳能到达的范围内，根据当前位置加跳力来更新 cur，更新 cur 的方法是比较当前的 cur 和 i + A[i] 之中的较大值，如果题目中未说明是否能到达末尾，还可以判断此时 pre 和 cur 是否相等，如果相等说明 cur 没有更新，即无法到达末尾位置，返回 -1，代码如下：

解法一：

```
class Solution {
public:
    int jump(vector<int>& nums) {
        int res = 0, n = nums.size(), i = 0, cur = 0;
        while (cur < n - 1) {
            ++res;
            int pre = cur;
            for (; i <= pre; ++i) {
                cur = max(cur, i + nums[i]);
            }
            if (pre == cur) return -1; // May not need this
```

```
        }
        return res;
    }
};
```

还有一种写法，跟上面那解法略有不同，但是本质的思想还是一样的，关于此解法的详细分析可参见网友 实验室小纸贴校外版的博客，这里 cur 是当前能到达的最远位置，last 是上一步能到达的最远位置，遍历数组，首先用 i + nums[i] 更新 cur，这个在上面解法中讲过了，然后判断如果当前位置到达了 last，即上一步能到达的最远位置，说明需要再跳一次了，将 last 赋值为 cur，并且步数 res 自增1，这里小优化一下，判断如果 cur 到达末尾了，直接 break 掉即可，代码如下：

解法二：

```
class Solution {
public:
    int jump(vector<int>& nums) {
        int res = 0, n = nums.size(), last = 0, cur = 0;
        for (int i = 0; i < n - 1; ++i) {
            cur = max(cur, i + nums[i]);
            if (i == last) {
                last = cur;
                ++res;
                if (cur >= n - 1) break;
            }
        }
        return res;
    }
};
```

46. Permutations

Given a collection of distinct integers, return all possible permutations.

Example:

```
Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

这道题是求全排列问题，给的输入数组没有重复项，这跟之前的那道 Combinations 和类似，解法基本相同，但是不同点在于那道不同的数字顺序只算一种，是一道典型的组合题，而此题是求全排列问题，还是用递归 DFS 来求解。这里需要用到一个 visited 数组来标记某个数字是否访问过，然后在 DFS 递归函数从的循环应从头开始，而不是从 level 开始，这是和 Combinations 不同的地方，其余思路大体相同。这里再说下 level 吧，其本质是记录当前已经拼出的个数，一旦其达到了 nums 数组的长度，说明此时已经是一个全排列了，因为再加数字的话，就会超出。还有就是，为啥这里的 level 要从0开始遍历，因为这是求全排列，每个位置都可能放任意一个数字，这样会有个问题，数字有可能被重复使用，由于全排列是不能重复使用数字的，所以需要用一个 visited 数组来标记某个数字是否使用过，代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& num) {
        vector<vector<int>> res;
        vector<int> out, visited(num.size(), 0);
        permuteDFS(num, 0, visited, out, res);
        return res;
    }
    void permuteDFS(vector<int>& num, int level, vector<int>& visited, vector<int>& out, vector<vector<int>>& res) {
        if (level == num.size()) {res.push_back(out); return;}
        for (int i = 0; i < num.size(); ++i) {
            if (visited[i] == 1) continue;
            visited[i] = 1;
            out.push_back(num[i]);
            permuteDFS(num, level + 1, visited, out, res);
            out.pop_back();
            visited[i] = 0;
        }
    }
};
```

上述解法的最终生成顺序为：[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]] 。

还有一种递归的写法，更简单一些，这里是每次交换 num 里面的两个数字，经过递归可以生成所有的排列情况。这里你可能注意到，为啥在递归函数中， push_back() 了之后没有返回呢，而解法一或者是 Combinations 的递归解法在更新结果 res 后都 return 了呢？其实如果你仔细看代码的话，此时 start 已经大于等于 num.size() 了，而下面的 for 循环的i是从 start 开始的，根本就不会执行 for 循环里的内容，就相当于 return 了，博主偷懒就没写了。但其实为了避免混淆，最好还是加上，免得和前面的搞混了，代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& num) {
        vector<vector<int>> res;
```

```
        permuteDFS(num, 0, res);
        return res;
    }
    void permuteDFS(vector<int>& num, int start, vector<vector<int>>& res)
{
        if (start >= num.size()) res.push_back(num);
        for (int i = start; i < num.size(); ++i) {
            swap(num[start], num[i]);
            permuteDFS(num, start + 1, res);
            swap(num[start], num[i]);
        }
    }
};
```

上述解法的最终生成顺序为：[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,2,1], [3,1,2]]

最后再来看一种方法，这种方法是 CareerCup 书上的方法，也挺不错的，这道题是思想是这样的：

当 n=1 时，数组中只有一个数 a1，其全排列只有一种，即为 a1

当 n=2 时，数组中此时有 a1a2，其全排列有两种，a1a2 和 a2a1，那么此时考虑和上面那种情况的关系，可以发现，其实就是在 a1 的前后两个位置分别加入了 a2

当 n=3 时，数组中有 a1a2a3，此时全排列有六种，分别为 a1a2a3, a1a3a2, a2a1a3, a2a3a1, a3a1a2, 和 a3a2a1。那么根据上面的结论，实际上是在 a1a2 和 a2a1 的基础上在不同的位置上加入 a3 而得到的。

_ a1 _ a2 _ : a3a1a2, a1a3a2, a1a2a3

_ a2 _ a1 _ : a3a2a1, a2a3a1, a2a1a3

解法三：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& num) {
        if (num.empty()) return vector<vector<int>>(1, vector<int>());
        vector<vector<int>> res;
        int first = num[0];
        num.erase(num.begin());
        vector<vector<int>> words = permute(num);
        for (auto &a : words) {
            for (int i = 0; i <= a.size(); ++i) {
                a.insert(a.begin() + i, first);
                res.push_back(a);
                a.erase(a.begin() + i);
            }
        }
        return res;
    }
};
```

上述解法的最终生成顺序为：[[1,2,3], [2,1,3], [2,3,1], [1,3,2], [3,1,2], [3,2,1]]

上面的三种解法都是递归的，我们也可以使用迭代的方法来做。其实下面这个解法就上面解法的迭代写法，核心思路都是一样的，都是在现有的排列的基础上，每个空位插入一个数字，从而生成各种的全排列的情况，参见代码如下：

解法四：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& num) {
        vector<vector<int>> res{{}};
        for (int a : num) {
            for (int k = res.size(); k > 0; --k) {
                vector<int> t = res.front();
                res.erase(res.begin());
                for (int i = 0; i <= t.size(); ++i) {
                    vector<int> one = t;
                    one.insert(one.begin() + i, a);
                    res.push_back(one);
                }
            }
        }
        return res;
    }
};
```

上述解法的最终生成顺序为：[[3,2,1], [2,3,1], [2,1,3], [3,1,2], [1,3,2], [1,2,3]]

下面这种解法就有些耍赖了，用了 STL 的内置函数 next_permutation()，专门就是用来返回下一个全排列，耳边又回响起了诸葛孔明的名言，我从未见过如此...投机取巧...的解法！

解法五：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& num) {
        vector<vector<int>> res;
        sort(num.begin(), num.end());
        res.push_back(num);
        while (next_permutation(num.begin(), num.end())) {
            res.push_back(num);
        }
        return res;
    }
};
```

上述解法的最终生成顺序为：[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

47. Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

Example:

```
Input: [1,1,2]
Output:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

这道题是之前那道 Permutations 的延伸，由于输入数组有可能出现重复数字，如果按照之前的算法运算，会有重复排列产生，我们要避免重复的产生，在递归函数中要判断前面一个数和当前的数是否相等，如果相等，且其对应的 visited 中的值为1，当前的数字才能使用（下文中会解释这样做的原因），否则需要跳过，这样就不会产生重复排列了，代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> res;
        vector<int> out, visited(nums.size(), 0);
        sort(nums.begin(), nums.end());
        permuteUniqueDFS(nums, 0, visited, out, res);
        return res;
    }
    void permuteUniqueDFS(vector<int>& nums, int level, vector<int>& visited, vector<int>& out, vector<vector<int>>& res) {
        if (level >= nums.size()) {res.push_back(out); return;}
        for (int i = 0; i < nums.size(); ++i) {
            if (visited[i] == 1) continue;
            if (i > 0 && nums[i] == nums[i - 1] && visited[i - 1] == 0) continue;
            visited[i] = 1;
            out.push_back(nums[i]);
            permuteUniqueDFS(nums, level + 1, visited, out, res);
            out.pop_back();
            visited[i] = 0;
        }
    }
};
```

在使用上面的方法的时候，一定要能弄清楚递归函数的 for 循环中两个 if 的剪枝的意思。在此之前，要弄清楚 level 的含义，这里用数组 out 来拼排列结果，level就是当前已经拼成的个数，其实就是 out 数组的长度。我们看到，for 循环的起始是从0开始的，而本题的解法二，三，四都是用了一个 start 变量，从而 for 循环都是从 start 开始，一定要分清楚 start 和本解法中的 level 的区别。由于递归的 for 都是从0开始，难免会重复遍历到数字，而全排列不能重复使用数字，意思是每个 nums 中的数字在全排列中只能使用一次（当然这并不妨碍 nums 中存在重复数字）。不能重复使用数字就靠 visited 数组来保证，这就是第一个 if 剪枝的意义所在。关键来看第二个 if 剪枝的意义，这里说当前数字和前一个数字相同，且前一个数字的 visited 值为0的时候，必须跳过。这里的前一个数 visited 值为0，并不代表前一个数字没有被处理过，也可能是递归结束后恢复状态时将 visited 值重置为0了，实际上就是这种情况，下面打印了一些中间过程的变量值，如下所示：

```
level = 0, i = 0 => out: {}
level = 1, i = 0 => out: {1 } skipped 1
level = 1, i = 1 => out: {1 }
level = 2, i = 0 => out: {1 2 } skipped 1
level = 2, i = 1 => out: {1 2 } skipped 1
level = 2, i = 2 => out: {1 2 }
level = 3 => saved  {1 2 2}
level = 3, i = 0 => out: {1 2 2 } skipped 1
level = 3, i = 1 => out: {1 2 2 } skipped 1
level = 3, i = 2 => out: {1 2 2 } skipped 1
level = 2, i = 2 => out: {1 2 2 } -> {1 2 } recovered
level = 1, i = 1 => out: {1 2 } -> {1 } recovered
level = 1, i = 2 => out: {1 } skipped 2
level = 0, i = 0 => out: {1 } -> {} recovered
level = 0, i = 1 => out: {}
level = 1, i = 0 => out: {2 }
level = 2, i = 0 => out: {2 1 } skipped 1
level = 2, i = 1 => out: {2 1 } skipped 1
level = 2, i = 2 => out: {2 1 }
level = 3 => saved  {1 2 2}
level = 3, i = 0 => out: {2 1 2 } skipped 1
level = 3, i = 1 => out: {2 1 2 } skipped 1
level = 3, i = 2 => out: {2 1 2 } skipped 1
level = 2, i = 2 => out: {2 1 2 } -> {2 1 } recovered
level = 1, i = 0 => out: {2 1 } -> {2 } recovered
level = 1, i = 1 => out: {2 } skipped 1
level = 1, i = 2 => out: {2 }
level = 2, i = 0 => out: {2 2 }
level = 3 => saved  {1 2 2}
level = 3, i = 0 => out: {2 2 1 } skipped 1
level = 3, i = 1 => out: {2 2 1 } skipped 1
level = 3, i = 2 => out: {2 2 1 } skipped 1
level = 2, i = 0 => out: {2 2 1 } -> {2 2 } recovered
level = 2, i = 1 => out: {2 2 } skipped 1
level = 2, i = 2 => out: {2 2 } skipped 1
level = 1, i = 2 => out: {2 2 } -> {2 } recovered
level = 0, i = 1 => out: {2 } -> {} recovered
```

```
level = 0, i = 2 => out: {} skipped 2
```

注意看这里面的 skipped 1 表示的是第一个 if 剪枝起作用的地方，skipped 2 表示的是第二个 if 剪枝起作用的地方。我们主要关心的是第二个 if 剪枝，看上方第一个蓝色标记的那行，再上面的红色一行表示在 level = 1, i = 1 时递归调用结束后，恢复到起始状态，那么此时 out 数组中只有一个1，后面的2已经被 pop_back() 了，当然对应的 visited 值也重置为0了，这种情况下需要剪枝，当然不能再一次把2往里加，因为这种情况在递归中已经加入到结果 res 中了，所以到了 level = 1, i = 2 的状态时，nums[i] == nums[i-1] && visited[i-1] == 0 的条件满足了，剪枝就起作用了，这种重复的情况就被剪掉了。

还有一种比较简便的方法，在 Permutations 的基础上稍加修改，用 TreeSet 来保存结果，利用其不会有重复项的特点，然后在递归函数中 swap 的地方，判断如果i和 start 不相同，但是 nums[i] 和 nums[start] 相同的情况下跳过，继续下一个循环，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        set<vector<int>> res;
        permute(nums, 0, res);
        return vector<vector<int>> (res.begin(), res.end());
    }
    void permute(vector<int>& nums, int start, set<vector<int>>& res) {
        if (start >= nums.size()) res.insert(nums);
        for (int i = start; i < nums.size(); ++i) {
            if (i != start && nums[i] == nums[start]) continue;
            swap(nums[i], nums[start]);
            permute(nums, start + 1, res);
            swap(nums[i], nums[start]);
        }
    }
};
```

对于上面的解法，你可能会有疑问，我们不是在 swap 操作之前已经做了剪枝了么，为什么还是会有重复出现，以至于还要用 TreeSet 来取出重复呢。总感觉使用 TreeSet 去重复有点耍赖，可能并没有探究到本题深层次的内容。这是很好的想法，首先尝试将上面的 TreeSet 还原为 vector，并且在主函数调用递归之前给 nums 排个序（代码参见评论区三楼），然后测试一个最简单的例子：[1, 2, 2]，得到的结果为：

[[1,2,2], [2,1,2], [2,2,1], [2,2,1], [2,1,2]]

我们发现有重复项，那么剪枝究竟在做些什么，怎么还是没法防止重复项的产生！那个剪枝只是为了防止当 start = 1, i = 2 时，将两个2交换，这样可以防止 {1, 2, 2} 被加入两次。但是没法防止其他的重复情况，要闹清楚为啥，需要仔细分析一些中间过程，下面打印了一些中间过程的变量：

```
start = 0, i = 0 => {1 2 2}
start = 1, i = 1 => {1 2 2}
start = 2, i = 2 => {1 2 2}
start = 3 => saved  {1 2 2}
```

```
start = 1, i = 2 => {1 2 2} skipped
start = 0, i = 1 => {1 2 2} -> {2 1 2}
start = 1, i = 1 => {2 1 2}
start = 2, i = 2 => {2 1 2}
start = 3 => saved  {2 1 2}
start = 1, i = 2 => {2 1 2} -> {2 2 1}
start = 2, i = 2 => {2 2 1}
start = 3 => saved  {2 2 1}
start = 1, i = 2 => {2 2 1} -> {2 1 2} recovered
start = 0, i = 1 => {2 1 2} -> {1 2 2} recovered
start = 0, i = 2 => {1 2 2} -> {2 2 1}
start = 1, i = 1 => {2 2 1}
start = 2, i = 2 => {2 2 1}
start = 3 => saved  {2 2 1}
start = 1, i = 2 => {2 2 1} -> {2 1 2}
start = 2, i = 2 => {2 1 2}
start = 3 => saved  {2 1 2}
start = 1, i = 2 => {2 1 2} -> {2 2 1} recovered
start = 0, i = 2 => {2 2 1} -> {1 2 2} recovered
```

问题出在了递归调用之后的还原状态，参见上面的红色的两行，当 start = 0, i = 2 时，nums 已经还原到了 {1, 2, 2} 的状态，此时 nums[start] 不等于 nums[i]，剪枝在这已经失效了，那么交换后的 {2, 2, 1} 还会被存到结果 res 中，而这个状态在之前就已经存过了一次。

注意到当 start = 0, i = 1 时，nums 交换之后变成了 {2, 1, 2}，如果能保持这个状态，那么当 start = 0, i = 2 时，此时 nums[start] 就等于 nums[i] 了，剪枝操作就可以发挥作用了。怎么才能当递归结束后，不还原成为交换之前的状态的呢？答案就是不进行还原，这样还是能保存为之前交换后的状态。只是将最后一句 swap(nums[i], nums[start]) 删掉是不行的，因为递归函数的参数 nums 是加了&号，就表示引用了，那么之前调用递归函数之前的 nums 在递归函数中会被修改，可能还是无法得到我们想要的顺序，所以要把递归函数的 nums 参数的&号也同时去掉才行，参见代码如下：

解法三：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());
        permute(nums, 0, res);
        return res;
    }
    void permute(vector<int> nums, int start, vector<vector<int>>& res) {
        if (start >= nums.size()) res.push_back(nums);
        for (int i = start; i < nums.size(); ++i) {
            if (i != start && nums[i] == nums[start]) continue;
            swap(nums[i], nums[start]);
            permute(nums, start + 1, res);
        }
    }
```

```
};
```

好，再测试下 [1, 2, 2] 这个例子，并且把中间变量打印出来：

```
start = 0, i = 0 => {1 2 2}
start = 1, i = 1 => {1 2 2}
start = 2, i = 2 => {1 2 2}
start = 3 => saved  {1 2 2}
start = 1, i = 2 => {1 2 2} skipped
start = 0, i = 1 => {1 2 2} -> {2 1 2}
start = 1, i = 1 => {2 1 2}
start = 2, i = 2 => {2 1 2}
start = 3 => saved  {2 1 2}
start = 1, i = 2 => {2 1 2} -> {2 2 1}
start = 2, i = 2 => {2 2 1}
start = 3 => saved  {2 2 1}
start = 1, i = 2 => {2 2 1} recovered
start = 0, i = 1 => {2 1 2} recovered
start = 0, i = 2 => {2 1 2} skipped
```

明显发现短了许多，说明剪枝发挥了作用，看上面红色部分，当 start = 0, i = 1 时，递归函数调用完了之后，nums 数组保持了 {2, 1, 2} 的状态，那么到 start = 0, i = 2 的时候，nums[start] 就等于 nums[i] 了，剪枝操作就可以发挥作用了。

这时候你可能会想，调用完递归不恢复状态，感觉怪怪的，跟哥的递归模版不一样啊，容易搞混啊，而且一会加&号，一会不加的，这尼玛谁能分得清啊。别担心，I gotcha covered! 好，既然还是要恢复状态的话，就只能从剪枝入手了，原来那种 naive 的剪枝方法肯定无法使用，矛盾的焦点还是在于，当 start = 0, i = 2 时，nums 被还原成了 start = 0, i = 1 的交换前的状态 {1, 2, 2}，这个状态已经被处理过了，再去处理一定会产生重复，怎么才知道这被处理过了呢，当前的 i = 2，需要往前去找是否有重复出现，由于数组已经排序过了，如果有重复，那么前面数一定和当前的相同，所以用一个 while 循环，往前找和 nums[i] 相同的数字，找到了就停下，当然如果小于 start 了也要停下，那么如果没有重复数字的话，j 一定是等于 start-1 的，那么如果不等于的话，就直接跳过就可以了，这样就可以去掉所有的重复啦，参见代码如下：

解法四：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());
        permute(nums, 0, res);
        return res;
    }
    void permute(vector<int>& nums, int start, vector<vector<int>>& res) {
        if (start >= nums.size()) res.push_back(nums);
        for (int i = start; i < nums.size(); ++i) {
            int j = i - 1;
            while (j >= start && nums[j] != nums[i]) --j;
```

```
            if (j != start - 1) continue;
            swap(nums[i], nums[start]);
            permute(nums, start + 1, res);
            swap(nums[i], nums[start]);
        }
    }
};
```

同样，我们再测试下 [1, 2, 2] 这个例子，并且把中间变量打印出来：

```
start = 0, i = 0 => {1 2 2} , j = -1
start = 1, i = 1 => {1 2 2} , j = 0
start = 2, i = 2 => {1 2 2} , j = 1
start = 3 => saved  {1 2 2}
start = 1, i = 2 => {1 2 2} skipped, j = 1
start = 0, i = 1 => {1 2 2} -> {2 1 2}, j = -1
start = 1, i = 1 => {2 1 2} , j = 0
start = 2, i = 2 => {2 1 2} , j = 1
start = 3 => saved  {2 1 2}
start = 1, i = 2 => {2 1 2} -> {2 2 1}, j = 0
start = 2, i = 2 => {2 2 1} , j = 1
start = 3 => saved  {2 2 1}
start = 1, i = 2 => {2 2 1} -> {2 1 2} recovered
start = 0, i = 1 => {2 1 2} -> {1 2 2} recovered
start = 0, i = 2 => {1 2 2} skipped, j = 1
```

到 start = 0, i = 2 的时候，j 此时等于1了，明显不是 start-1，说明有重复了，直接 skip 掉，这样剪枝操作就可以发挥作用了。

之前的 Permutations 中的解法三也可以用在这里，只不过需要借助 TreeSet 来去重复，博主还未想出其他不用集合的去重复的方法，哪位看官大神们知道的话，请一定要留言告知博主，参见代码如下：

解法五：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        if (nums.empty()) return vector<vector<int>>(1, vector<int>());
        set<vector<int>> res;
        int first = nums[0];
        nums.erase(nums.begin());
        vector<vector<int>> words = permuteUnique(nums);
        for (auto &a : words) {
            for (int i = 0; i <= a.size(); ++i) {
                a.insert(a.begin() + i, first);
                res.insert(a);
                a.erase(a.begin() + i);
            }
        }
```

```
        return vector<vector<int>> (res.begin(), res.end());
    }
};
```

之前的 Permutations 中的解法四博主没法成功修改使其可以通过这道题，即便是将结果 res 用 TreeSet 来去重复，还是不对。同样，哪位看官大神们知道的话，请一定要留言告知博主。不过之前的 Permutations 中的解法五却可以原封不动的搬到这道题来，看来自带的 next_permutation() 函数就是叼啊，自带去重复功能，叼叼叼！参见代码如下：

解法六：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());
        res.push_back(nums);
        while (next_permutation(nums.begin(), nums.end())) {
            res.push_back(nums);
        }
        return res;
    }
};
```

48. Rotate Image

You are given an *n* x *n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Note:

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

Example 1:

```
Given input matrix =
[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],

rotate the input matrix in-place such that it becomes:
[
  [7,4,1],
  [8,5,2],
  [9,6,3]
]
```

Example 2:

```
Given input matrix =
[
  [ 5, 1, 9,11],
  [ 2, 4, 8,10],
  [13, 3, 6, 7],
  [15,14,12,16]
],

rotate the input matrix in-place such that it becomes:
[
  [15,13, 2, 5],
  [14, 3, 4, 1],
  [12, 6, 8, 9],
  [16, 7,10,11]
]
```

在计算机图像处理里，旋转图片是很常见的，由于图片的本质是二维数组，所以也就变成了对数组的操作处理，翻转的本质就是某个位置上数移动到另一个位置上，比如用一个简单的例子来分析：

```
1  2  3          7  4  1

4  5  6   -->    8  5  2

7  8  9          9  6  3
```

对于90度的翻转有很多方法，一步或多步都可以解，先来看一种直接的方法，这种方法是按顺时针的顺序去覆盖前面的数字，从四个顶角开始，然后往中间去遍历，每次覆盖的坐标都是同理，如下：

(i, j) <- (n-1-j, i) <- (n-1-i, n-1-j) <- (j, n-1-i)

这其实是个循环的过程，第一个位置又覆盖了第四个位置，这里i的取值范围是 [0, n/2)，j的取值范围是 [i, n-1-i)，至于为什么i和j是这个取值范围，为啥i不用遍历 [n/2, n)，若仔细观察这些位置之间的联系，不难发现，实际上j列的范围 [i, n-1-i) 顺时针翻转 90 度，正好就是i行的 [n/2, n) 的位置，这个方法每次循环换四个数字，如下所示：

```
1  2  3              7  2  1           7  4  1

4  5  6     -->      4  5  6    -->    8  5  2

7  8  9              9  8  3           9  6  3
```

解法一：

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        for (int i = 0; i < n / 2; ++i) {
            for (int j = i; j < n - 1 - i; ++j) {
                int tmp = matrix[i][j];
                matrix[i][j] = matrix[n - 1 - j][i];
                matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j];
                matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i];
                matrix[j][n - 1 - i] = tmp;
            }
        }
    }
};
```

还有一种解法，首先以从对角线为轴翻转，然后再以x轴中线上下翻转即可得到结果，如下图所示(其中蓝色数字表示翻转轴)：

```
1  2  3          9  6  3            7  4  1

4  5  6   -->    8  5  2    -->     8  5  2

7  8  9          7  4  1            9  6  3
```

解法二：

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        for (int i = 0; i < n - 1; ++i) {
            for (int j = 0; j < n - i; ++j) {
                swap(matrix[i][j], matrix[n - 1- j][n - 1 - i]);
            }
        }
        reverse(matrix.begin(), matrix.end());
    }
};
```

最后再来看一种方法，这种方法首先对原数组取其转置矩阵，然后把每行的数字翻转可得到结果，如下所示（其中蓝色数字表示翻转轴，Github 上可能无法显示颜色，请参见博客园上的帖子）：

```
1  2  3          1  4  7            7  4  1

4  5  6   -->    2  5  8    -->     8  5  2

7  8  9          3  6  9            9  6  3
```

解法三：

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                swap(matrix[i][j], matrix[j][i]);
            }
            reverse(matrix[i].begin(), matrix[i].end());
        }
    }
};
```

49. Group Anagrams

Given an array of strings, group anagrams together.

Example:

```
Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
Output:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

Note:

- All inputs will be in lowercase.
- The order of your output does not matter.

这道题让我们群组给定字符串集中所有的错位词，所谓的错位词就是两个字符串中字母出现的次数都一样，只是位置不同，比如 abc，bac, cba 等它们就互为错位词，那么如何判断两者是否是错位词呢，可以发现如果把错位词的字符顺序重新排列，那么会得到相同的结果，所以重新排序是判断是否互为错位词的方法，由于错位词重新排序后都会得到相同的字符串，以此作为 key，将所有错位词都保存到字符串数组中，建立 key 和当前的不同的错位词集合个数之间的映射，这里之所以没有建立 key 和其隶属的错位词集合之间的映射，是用了一个小 trick，从而避免了最后再将 HashMap 中的集合拷贝到结果 res 中。当检测到当前的单词不在 HashMap 中，此时知道这个单词将属于一个新的错位词集合，所以将其映射为当前的错位词集合的个数，然后在 res 中新增一个空集合，这样就可以通过其映射值，直接找到新的错位词集合的位置，从而将新的单词存入结果 res 中，参见代码如下：

解法一：

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        vector<vector<string>> res;
        unordered_map<string, int> m;
        for (string str : strs) {
            string t = str;
            sort(t.begin(), t.end());
            if (!m.count(t)) {
                m[t] = res.size();
                res.push_back({});
            }
            res[m[t]].push_back(str);
        }
        return res;
    }
};
```

下面这种解法没有用到排序，用一个大小为 26 的 int 数组来统计每个单词中字符出现的次数，然后将 int 数组转为一个唯一的字符串，跟字符串数组进行映射，这样就不用给字符串排序了，参见代码如下：

解法二：

```cpp
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        vector<vector<string>> res;
        unordered_map<string, vector<string>> m;
        for (string str : strs) {
            vector<int> cnt(26);
            string t;
            for (char c : str) ++cnt[c - 'a'];
            for (int i = 0; i < 26; ++i) {
                if (cnt[i] == 0) continue;
                t += string(1, i + 'a') + to_string(cnt[i]);
            }
            m[t].push_back(str);
        }
        for (auto a : m) {
            res.push_back(a.second);
        }
        return res;
    }
};
```

51. N-Queens

The *n* -queens puzzle is the problem of placing *n* queens on an *n* × *n* chessboard such that no two queens attack each other.

Given an integer *n* , return all distinct solutions to the *n* -queens puzzle.

Each solution contains a distinct board configuration of the *n* -queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

Example:

```
Input: 4
Output: [
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
Explanation: There exist two distinct solutions to the 4-queens puzzle as
shown above.
```

经典的N皇后问题，基本所有的算法书中都会包含的问题。可能有些人对国际象棋不太熟悉，大家都知道中国象棋中最叼的是车，横竖都能走，但是在国际象棋中还有更叼的，就是皇后，不但能横竖走，还能走两个斜线，有如 bug 一般的存在。所以经典的八皇后问题就应运而生了，在一个 8x8 大小的棋盘上如果才能放8个皇后，使得两两之间不能相遇，所谓一山不能容二虎，而这里有八个母老虎，互相都不能相遇。对于这类问题，没有太简便的方法，只能使用穷举法，就是尝试所有的组合，每放置一个新的皇后的时候，必须要保证跟之前的所有皇后不能冲突，若发生了冲突，说明当前位置不能放，要重新找地方，这个逻辑非常适合用递归来做。我们先建立一个长度为 nxn 的全是点的数组 queens，然后从第0行开始调用递归。在递归函数中，我们首先判断当前行数是否已经为n，是的话，说明所有的皇后都已经成功放置好了，所以我们只要将 queens 数组加入结果 res 中即可。否则的话，我们遍历该行的所有列的位置，行跟列的位置都确定后，我们要验证当前位置是否会产生冲突，那么就需要使用一个子函数来判断了，首先验证该列是否有冲突，就遍历之前的所有行，若某一行相同列也有皇后，则冲突返回 false；再验证两个对角线是否冲突，就是一些坐标转换，主要不要写错了，若都没有冲突，则说明该位置可以放皇后，放了新皇后之后，再对下一行调用递归即可，注意递归结束之后要返回状态，参见代码如下：

解法一：

```
class Solution {
public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> res;
        vector<string> queens(n, string(n, '.'));
        helper(0, queens, res);
        return res;
    }
    void helper(int curRow, vector<string>& queens,
vector<vector<string>>& res) {
        int n = queens.size();
        if (curRow == n) {
            res.push_back(queens);
            return;
        }
        for (int i = 0; i < n; ++i) {
```

```
            if (isValid(queens, curRow, i)) {
                queens[curRow][i] = 'Q';
                helper(curRow + 1, queens, res);
                queens[curRow][i] = '.';
            }
        }
    }
    bool isValid(vector<string>& queens, int row, int col) {
        for (int i = 0; i < row; ++i) {
            if (queens[i][col] == 'Q') return false;
        }
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; --i, --j) {
            if (queens[i][j] == 'Q') return false;
        }
        for (int i = row - 1, j = col + 1; i >= 0 && j < queens.size(); --i, ++j) {
            if (queens[i][j] == 'Q') return false;
        }
        return true;
    }
};
```

我们还可以只使用一个一维数组 queenCol 来保存所有皇后的列位置，初始化均为-1， 那么 queenCol[i] 就是表示第i个皇后在 (i, queenCol[i]) 位置，递归函数还是跟上面的解法相同，就是在当前行数等于n的时候，我们要将 queenCol 还原成一个 nxn 大小的矩阵，并存入结果 res 中。这种记录每个皇后的坐标的方法在验证冲突的时候比较简单，只要从第0行遍历到当前行，若跟之前的皇后的列数相同，直接返回false，叼就叼在判断对角线冲突非常简便，因为当两个点在同一条对角线上，那么二者的横坐标差的绝对值等于纵坐标差的绝对值，利用这条性质，可以快速的判断冲突，代码如下：

解法二：

```
class Solution {
public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> res;
        vector<int> queenCol(n, -1);
        helper(0, queenCol, res);
        return res;
    }
    void helper(int curRow, vector<int>& queenCol, vector<vector<string>>& res) {
        int n = queenCol.size();
        if (curRow == n) {
            vector<string> out(n, string(n, '.'));
            for (int i = 0; i < n; ++i) {
                out[i][queenCol[i]] = 'Q';
            }
            res.push_back(out);
            return;
```

```
        }
        for (int i = 0; i < n; ++i) {
            if (isValid(queenCol, curRow, i)) {
                queenCol[curRow] = i;
                helper(curRow + 1, queenCol, res);
                queenCol[curRow] = -1;
            }
        }
    }
    bool isValid(vector<int>& queenCol, int row, int col) {
        for (int i = 0; i < row; ++i) {
            if (col == queenCol[i] || abs(row - i) == abs(col - queenCol[i])) return false;
        }
        return true;
    }
};
```

52. N-Queens II

The *n* -queens puzzle is the problem of placing *n* queens on an *n* × *n* chessboard such that no two queens attack each other.

Given an integer *n* , return the number of distinct solutions to the *n* -queens puzzle.

Example:

```
Input: 4
Output: 2
Explanation: There are two distinct solutions to the 4-queens puzzle as shown below.
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

这道题是之前那道 N-Queens 的延伸，说是延伸其实我觉得两者顺序应该颠倒一样，上一道题比这道题还要稍稍复杂一些，两者本质上没有啥区别，都是要用回溯法 Backtracking 来解，如果理解了之前那道题的思路，此题只要做很小的改动即可，不再需要求出具体的皇后的摆法，只需要每次生成一种解法时，计数器加一即可，代码如下：

解法一：

```
class Solution {
public:
    int totalNQueens(int n) {
        int res = 0;
        vector<int> pos(n, -1);
        helper(pos, 0, res);
        return res;
    }
    void helper(vector<int>& pos, int row, int& res) {
        int n = pos.size();
        if (row == n) ++res;
        for (int col = 0; col < n; ++col) {
            if (isValid(pos, row, col)) {
                pos[row] = col;
                helper(pos, row + 1, res);
                pos[row] = -1;
            }
        }
    }
    bool isValid(vector<int>& pos, int row, int col) {
        for (int i = 0; i < row; ++i) {
            if (col == pos[i] || abs(row - i) == abs(col - pos[i])) {
                return false;
            }
        }
        return true;
    }
};
```

但是其实我们并不需要知道每一行皇后的具体位置，而只需要知道会不会产生冲突即可。对于每行要新加的位置，需要看跟之前的列，对角线，及逆对角线之间是否有冲突，所以我们需要三个布尔型数组，分别来记录之前的列 cols，对角线 diag，及逆对角线 anti_diag 上的位置，其中 cols 初始化大小为n，diag 和 anti_diag 均为 2n。列比较简单，是哪列就直接去 cols 中查找，而对角线的话，需要处理一下，如果我们仔细观察数组位置坐标的话，可以发现所有同一条主对角线的数，其纵坐标减去横坐标再加n，一定是相等的。同理，同一条逆对角线上的数字，其横纵坐标之和一定是相等的，根据这个，就可以快速判断主逆对角线上是否有冲突。任意一个有冲突的话，直接跳过当前位置，否则对于新位置，三个数组中对应位置都赋值为 true，然后对下一行调用递归，递归返回后记得还要还原状态，参见代码如下：

解法二：

```
class Solution {
```

```
public:
    int totalNQueens(int n) {
        int res = 0;
        vector<bool> cols(n), diag(2 * n), anti_diag(2 * n);
        helper(n, 0, cols, diag, anti_diag, res);
        return res;
    }
    void helper(int n, int row, vector<bool>& cols, vector<bool>& diag, vector<bool>& anti_diag, int& res) {
        if (row == n) ++res;
        for (int col = 0; col < n; ++col) {
            int idx1 = col - row + n, idx2 = col + row;
            if (cols[col] || diag[idx1] || anti_diag[idx2]) continue;
            cols[col] = diag[idx1] = anti_diag[idx2] = true;
            helper(n, row + 1, cols, diag, anti_diag, res);
            cols[col] = diag[idx1] = anti_diag[idx2] = false;
        }
    }
};
```