201. Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

Credits:
Special thanks to @amrsaqr for adding this problem and creating all test cases.

又是一道考察位操作Bit Operation的题，相似的题目在LeetCode中还真不少，比如Repeated DNA Sequences 求重复的DNA序列， Single Number 单独的数字，Single Number II 单独的数字之二 ， Grey Code 格雷码，和 Reverse Bits 翻转位 等等，那么这道题其实并不难，我们先从题目中给的例子来分析，[5, 7]里共有三个数字，分别写出它们的二进制为：

101　　110　　111

相与后的结果为100，仔细观察我们可以得出，最后的数是该数字范围内所有的数的左边共同的部分，如果上面那个例子不太明显，我们再来看一个范围[26, 30]，它们的二进制如下：

11010　　11011　　11100　　11101　　11110

发现了规律后，我们只要写代码找到左边公共的部分即可，我们可以从建立一个32位都是1的mask，然后每次向左移一位，比较m和n是否相同，不同再继续左移一位，直至相同，然后把m和mask相与就是最终结果，代码如下：

解法一：

```
class Solution {
public:
    int rangeBitwiseAnd(int m, int n) {
        int d = INT_MAX;
        while ((m & d) != (n & d)) {
            d <<= 1;
        }
        return m & d;
    }
};
```

此题还有另一种解法，不需要用mask，直接平移m和n，每次向右移一位，直到m和n相等，记录下所有平移的次数i，然后再把m左移i位即为最终结果，代码如下：

解法二：

```
class Solution {
public:
    int rangeBitwiseAnd(int m, int n) {
        int i = 0;
        while (m != n) {
            m >>= 1;
            n >>= 1;
            ++i;
        }
        return (m << i);
    }
};
```

下面这种方法有点叼，就一行搞定了，通过递归来做的，如果n大于m，那么就对m和n分别除以2，并且调用递归函数，对结果再乘以2，一定要乘回来，不然就不对了，就举一个最简单的例子，m = 10, n = 11，注意这里是二进制表示的，然后各自除以2，都变成了1，调用递归返回1，这时候要乘以2，才能变回10，参见代码如下：

解法三：

```
class Solution {
public:
    int rangeBitwiseAnd(int m, int n) {
        return (n > m) ? (rangeBitwiseAnd(m / 2, n / 2) << 1) : m;
    }
};
```

下面这种方法也不错，也很简单，如果m小于n就进行循环，n与上n-1，那么为什么要这样与呢，举个简单的例子呗，110与上(110-1)，得到100，这不就相当于去掉最低位的1么，n就这样每次去掉最低位的1，如果小于等于m了，返回此时的n即可，参见代码如下：

解法四：

```
class Solution {
public:
    int rangeBitwiseAnd(int m, int n) {
        while (m < n) n &= (n - 1);
        return n;
    }
};
```

207. Course Schedule

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

Example 1:

```
Input: 2, [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: 2, [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

Note:

1. The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

Hints:

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. There are several ways to represent a graph. For example, the input prerequisites is a graph represented by a list of edges. Is this graph representation appropriate?
3. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
4. Topological sort could also be done via BFS.

这道课程清单的问题对于我们学生来说应该不陌生，因为在选课的时候经常会遇到想选某一门课程，发现选它之前必须先上了哪些课程，这道题给了很多提示，第一条就告诉了这道题的本质就是在有向图中检测环。 LeetCode 中关于图的题很少，有向图的仅此一道，还有一道关于无向图的题是 Clone Graph。个人认为图这种数据结构相比于树啊，链表啊什么的要更为复杂一些，尤其是有向图，很麻烦。第二条提示是在讲如何来表示一个有向图，可以用边来表示，边是由两个端点组成的，用两个点来表示边。第三第四条提示揭示了此题有两种解法，DFS 和 BFS 都可以解此题。先来看 BFS 的解法，定义二维数组 graph 来表示这个有向图，一维数组 in 来表示每个顶点的入度。开始先根据输入来建立这个有向图，并将入度数组也初始化好。然后定义一个 queue 变量，将所有入度为0的点放入队列中，然后开始遍历队列，从 graph 里遍历其连接的点，每到达一个新节点，将其入度减一，如果此时该点入度为0，则放入队列末尾。直到遍历完队列中所有的值，若此时还有节点的入度不为0，则说明环存在，返回 false，反之则返回 true。代码如下：

解法一：

```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> graph(numCourses, vector<int>());
        vector<int> in(numCourses);
        for (auto a : prerequisites) {
            graph[a[1]].push_back(a[0]);
            ++in[a[0]];
        }
        queue<int> q;
        for (int i = 0; i < numCourses; ++i) {
            if (in[i] == 0) q.push(i);
        }
        while (!q.empty()) {
            int t = q.front(); q.pop();
            for (auto a : graph[t]) {
                --in[a];
                if (in[a] == 0) q.push(a);
            }
        }
        for (int i = 0; i < numCourses; ++i) {
            if (in[i] != 0) return false;
        }
        return true;
    }
};
```

下面来看 DFS 的解法，也需要建立有向图，还是用二维数组来建立，和 BFS 不同的是，像现在需要一个一维数组 visit 来记录访问状态，这里有三种状态，0表示还未访问过，1表示已经访问了，-1 表示有冲突。大体思路是，先建立好有向图，然后从第一个门课开始，找其可构成哪门课，暂时将当前课程标记为已访问，然后对新得到的课程调用 DFS 递归，直到出现新的课程已经访问过了，则返回 false，没有冲突的话返回 true，然后把标记为已访问的课程改为未访问。代码如下：

解法二：

```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> graph(numCourses, vector<int>());
        vector<int> visit(numCourses);
        for (auto a : prerequisites) {
            graph[a[1]].push_back(a[0]);
        }
        for (int i = 0; i < numCourses; ++i) {
            if (!canFinishDFS(graph, visit, i)) return false;
        }
        return true;
    }
```

```
    bool canFinishDFS(vector<vector<int>>& graph, vector<int>& visit, int i) {
        if (visit[i] == -1) return false;
        if (visit[i] == 1) return true;
        visit[i] = -1;
        for (auto a : graph[i]) {
            if (!canFinishDFS(graph, visit, a)) return false;
        }
        visit[i] = 1;
        return true;
    }
};
```

208. Implement Trie (Prefix Tree)

Implement a trie with `insert`, `search`, and `startsWith` methods.

Example:

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // returns true
trie.search("app");     // returns false
trie.startsWith("app"); // returns true
trie.insert("app");
trie.search("app");     // returns true
```

Note:

- You may assume that all inputs are consist of lowercase letters `a-z`.
- All inputs are guaranteed to be non-empty strings.

这道题让我们实现一个重要但又有些复杂的数据结构-字典树， 又称前缀树或单词查找树，详细介绍可以参见网友董的博客，例如，一个保存了8个键的trie结构，"A", "to", "tea", "ted", "ten", "i", "in", and "inn"，如下图所示：

字典树主要有如下三点性质：

\1. 根节点不包含字符，除根节点意外每个节点只包含一个字符。

\2. 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。

\3. 每个节点的所有子节点包含的字符串不相同。

字母树的插入（Insert）、删除（Delete）和查找（Find）都非常简单，用一个一重循环即可，即第i次循环找到前i 个字母所对应的子树，然后进行相应的操作。实现这棵字母树，我们用最常见的数组保存（静态开辟内存）即可，当然也可以开动态的指针类型（动态开辟内存）。至于结点对儿子的指向，一般有三种方法：

1、对每个结点开一个字母集大小的数组，对应的下标是儿子所表示的字母，内容则是这个儿子对应在大数组上的位置，即标号；

2、对每个结点挂一个链表，按一定顺序记录每个儿子是谁；

3、使用左儿子右兄弟表示法记录这棵树。

三种方法，各有特点。第一种易实现，但实际的空间要求较大；第二种，较易实现，空间要求相对较小，但比较费时；第三种，空间要求最小，但相对费时且不易写。

我们这里只来实现第一种方法，这种方法实现起来简单直观，字母的字典树每个节点要定义一个大小为 26 的子节点指针数组，然后用一个标志符用来记录到当前位置为止是否为一个词，初始化的时候讲 26 个子节点都赋为空。那么 insert 操作只需要对于要插入的字符串的每一个字符算出其的位置，然后找是否存在这个子节点，若不存在则新建一个，然后再查找下一个。查找词和找前缀操作跟 insert 操作都很类似，不同点在于若不存在子节点，则返回 false。查找次最后还要看标识位，而找前缀直接返回 true 即可。代码如下：

```
class TrieNode {
public:
    TrieNode *child[26];
    bool isWord;
    TrieNode(): isWord(false) {
        for (auto &a : child) a = nullptr;
    }
};

class Trie {
public:
    Trie() {
        root = new TrieNode();
    }
    void insert(string s) {
        TrieNode *p = root;
        for (auto &a : s) {
            int i = a - 'a';
            if (!p->child[i]) p->child[i] = new TrieNode();
            p = p->child[i];
        }
        p->isWord = true;
    }
    bool search(string key) {
        TrieNode *p = root;
        for (auto &a : key) {
            int i = a - 'a';
            if (!p->child[i]) return false;
            p = p->child[i];
        }
        return p->isWord;
    }
    bool startsWith(string prefix) {
```

```
        TrieNode *p = root;
        for (auto &a : prefix) {
            int i = a - 'a';
            if (!p->child[i]) return false;
            p = p->child[i];
        }
        return true;
    }

private:
    TrieNode* root;
};
```

209. Minimum Size Subarray Sum

Given an array of n positive integers and a positive integer s, find the minimal length of a contiguous subarray of which the sum ≥ s. If there isn't one, return 0 instead.

Example:

```
Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: the subarray [4,3] has the minimal length under the problem constraint.
```

Follow up:

If you have figured out the *O* (*n*) solution, try coding another solution of which the time complexity is *O* (*n* log *n*).

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

这道题给定了我们一个数字，让求子数组之和大于等于给定值的最小长度，注意这里是大于等于，不是等于。跟之前那道 Maximum Subarray 有些类似，并且题目中要求实现 O(n) 和 O(nlgn) 两种解法，那么先来看 O(n) 的解法，需要定义两个指针 left 和 right，分别记录子数组的左右的边界位置，然后让 right 向右移，直到子数组和大于等于给定值或者 right 达到数组末尾，此时更新最短距离，并且将 left 像右移一位，然后再 sum 中减去移去的值，然后重复上面的步骤，直到 right 到达末尾，且 left 到达临界位置，即要么到达边界，要么再往右移动，和就会小于给定值。代码如下：

解法一

```
// O(n)
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        if (nums.empty()) return 0;
```

```
        int left = 0, right = 0, sum = 0, len = nums.size(), res = len + 1;
        while (right < len) {
            while (sum < s && right < len) {
                sum += nums[right++];
            }
            while (sum >= s) {
                res = min(res, right - left);
                sum -= nums[left++];
            }
        }
        return res == len + 1 ? 0 : res;
    }
};
```

同样的思路，我们也可以换一种写法，参考代码如下：

解法二：

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int res = INT_MAX, left = 0, sum = 0;
        for (int i = 0; i < nums.size(); ++i) {
            sum += nums[i];
            while (left <= i && sum >= s) {
                res = min(res, i - left + 1);
                sum -= nums[left++];
            }
        }
        return res == INT_MAX ? 0 : res;
    }
};
```

下面再来看看 O(nlgn) 的解法，这个解法要用到二分查找法，思路是，建立一个比原数组长一位的 sums 数组，其中 sums[i] 表示 nums 数组中 [0, i - 1] 的和，然后对于 sums 中每一个值 sums[i]，用二分查找法找到子数组的右边界位置，使该子数组之和大于 sums[i] + s，然后更新最短长度的距离即可。代码如下：

解法三：

```
// O(nlgn)
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int len = nums.size(), sums[len + 1] = {0}, res = len + 1;
        for (int i = 1; i < len + 1; ++i) sums[i] = sums[i - 1] + nums[i - 1];
        for (int i = 0; i < len + 1; ++i) {
```

```
            int right = searchRight(i + 1, len, sums[i] + s, sums);
            if (right == len + 1) break;
            if (res > right - i) res = right - i;
        }
        return res == len + 1 ? 0 : res;
    }
    int searchRight(int left, int right, int key, int sums[]) {
        while (left <= right) {
            int mid = (left + right) / 2;
            if (sums[mid] >= key) right = mid - 1;
            else left = mid + 1;
        }
        return left;
    }
};
```

我们也可以不用为二分查找法专门写一个函数，直接嵌套在 for 循环中即可，参加代码如下：

解法四：

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int res = INT_MAX, n = nums.size();
        vector<int> sums(n + 1, 0);
        for (int i = 1; i < n + 1; ++i) sums[i] = sums[i - 1] + nums[i - 1];
        for (int i = 0; i < n; ++i) {
            int left = i + 1, right = n, t = sums[i] + s;
            while (left <= right) {
                int mid = left + (right - left) / 2;
                if (sums[mid] < t) left = mid + 1;
                else right = mid - 1;
            }
            if (left == n + 1) break;
            res = min(res, left - i);
        }
        return res == INT_MAX ? 0 : res;
    }
};
```

讨论：本题有一个很好的 Follow up，就是去掉所有数字是正数的限制条件，而去掉这个条件会使得累加数组不一定会是递增的了，那么就不能使用二分法，同时双指针的方法也会失效，只能另辟蹊径了。其实博主觉得同时应该去掉大于s的条件，只保留 sum=s 这个要求，因为这样就可以在建立累加数组后用 2sum 的思路，快速查找 s-sum 是否存在，如果有了大于的条件，还得继续遍历所有大于 s-sum 的值，效率提高不了多少。

210. Course Schedule II

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished
             course 0. So the correct course order is [0,1] .
```

Example 2:

```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both
             courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.
             So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3] .
```

Note:

1. The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

Hints:

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

这题是之前那道 Course Schedule 的扩展，那道题只让我们判断是否能完成所有课程，即检测有向图中是否有环，而这道题我们得找出要上的课程的顺序，即有向图的拓扑排序 Topological Sort，这样一来，难度就增加了，但是由于我们有之前那道的基础，而此题正是基于之前解法的基础上稍加修改，我们从 queue 中每取出一个数组就将其存在结果中，最终若有向图中有环，则结果中元素的个数不等于总课程数，那我们将结果清空即可。代码如下：

```cpp
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<int> res;
        vector<vector<int> > graph(numCourses, vector<int>(0));
        vector<int> in(numCourses, 0);
        for (auto &a : prerequisites) {
            graph[a.second].push_back(a.first);
            ++in[a.first];
        }
        queue<int> q;
        for (int i = 0; i < numCourses; ++i) {
            if (in[i] == 0) q.push(i);
        }
        while (!q.empty()) {
            int t = q.front();
            res.push_back(t);
            q.pop();
            for (auto &a : graph[t]) {
                --in[a];
                if (in[a] == 0) q.push(a);
            }
        }
        if (res.size() != numCourses) res.clear();
        return res;
    }
};
```

211. Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.`. A `.` means it can represent any one letter.

For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Note:
You may assume that all words are consist of lowercase letters `a-z`.

click to show hint.

You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.

LeetCode出新题的速度越来越快了，有点跟不上节奏的感觉了。这道题如果做过之前的那道 Implement Trie (Prefix Tree) 实现字典树(前缀树)的话就没有太大的难度了，还是要用到字典树的结构，唯一不同的地方就是search的函数需要重新写一下，因为这道题里面'.'可以代替任意字符，所以一旦有了'.'，就需要查找所有的子树，只要有一个返回true，整个search函数就返回true，典型的DFS的问题，其他部分跟上一道实现字典树没有太大区别，代码如下：

```
class WordDictionary {
public:
    struct TrieNode {
    public:
        TrieNode *child[26];
        bool isWord;
        TrieNode() : isWord(false) {
            for (auto &a : child) a = NULL;
        }
    };

    WordDictionary() {
        root = new TrieNode();
    }

    // Adds a word into the data structure.
    void addWord(string word) {
        TrieNode *p = root;
        for (auto &a : word) {
            int i = a - 'a';
            if (!p->child[i]) p->child[i] = new TrieNode();
            p = p->child[i];
        }
        p->isWord = true;
    }

    // Returns if the word is in the data structure. A word could
```

```
    // contain the dot character '.' to represent any one letter.
    bool search(string word) {
        return searchWord(word, root, 0);
    }

    bool searchWord(string &word, TrieNode *p, int i) {
        if (i == word.size()) return p->isWord;
        if (word[i] == '.') {
            for (auto &a : p->child) {
                if (a && searchWord(word, a, i + 1)) return true;
            }
            return false;
        } else {
            return p->child[word[i] - 'a'] && searchWord(word, p->child[word[i] - 'a'], i + 1);
        }
    }

private:
    TrieNode *root;
};

// Your WordDictionary object will be instantiated and called as such:
// WordDictionary wordDictionary;
// wordDictionary.addWord("word");
// wordDictionary.search("pattern");
```

讨论：这道题有个很好的Follow up，就是当搜索的单词中存在星号怎么搞，星号的定义和Wildcard Matching中一样，可以代表任意的字符串，包括空字符串，请参见评论区1楼。

212. Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

Example:

```
Input:
board = [
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
words = ["oath","pea","eat","rain"]

Output: ["eat","oath"]
```

Note:

1. All inputs are consist of lowercase letters `a-z`.
2. The values of `words` are distinct.

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

这道题是在之前那道 Word Search 的基础上做了些拓展，之前是给一个单词让判断是否存在，现在是给了一堆单词，让返回所有存在的单词，在这道题最开始更新的几个小时内，用 brute force 是可以通过 OJ 的，就是在之前那题的基础上多加一个 for 循环而已，但是后来出题者其实是想考察字典树的应用，所以加了一个超大的 test case，以至于 brute force 无法通过，强制我们必须要用字典树来求解。LeetCode 中有关字典树的题还有 Implement Trie (Prefix Tree) 和 Add and Search Word - Data structure design，那么我们在这题中只要实现字典树中的 insert 功能就行了，查找单词和前缀就没有必要了，然后 DFS 的思路跟之前那道 Word Search 基本相同，请参见代码如下：

```
class Solution {
public:
    struct TrieNode {
        TrieNode *child[26];
        string str;
        TrieNode() : str("") {
            for (auto &a : child) a = NULL;
        }
    };
    struct Trie {
        TrieNode *root;
        Trie() : root(new TrieNode()) {}
        void insert(string s) {
            TrieNode *p = root;
            for (auto &a : s) {
                int i = a - 'a';
                if (!p->child[i]) p->child[i] = new TrieNode();
```

```
                p = p->child[i];
            }
            p->str = s;
        }
    };
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        vector<string> res;
        if (words.empty() || board.empty() || board[0].empty()) return res;
        vector<vector<bool>> visit(board.size(), vector<bool>(board[0].size(), false));
        Trie T;
        for (auto &a : words) T.insert(a);
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j) {
                if (T.root->child[board[i][j] - 'a']) {
                    search(board, T.root->child[board[i][j] - 'a'], i, j, visit, res);
                }
            }
        }
        return res;
    }
    void search(vector<vector<char>>& board, TrieNode* p, int i, int j, vector<vector<bool>>& visit, vector<string>& res) {
        if (!p->str.empty()) {
            res.push_back(p->str);
            p->str.clear();
        }
        int d[][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        visit[i][j] = true;
        for (auto &a : d) {
            int nx = a[0] + i, ny = a[1] + j;
            if (nx >= 0 && nx < board.size() && ny >= 0 && ny < board[0].size() && !visit[nx][ny] && p->child[board[nx][ny] - 'a']) {
                search(board, p->child[board[nx][ny] - 'a'], nx, ny, visit, res);
            }
        }
        visit[i][j] = false;
    }
};
```

213. House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Example 1:

```
Input: [2,3,2]
Output: 3
Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2),
             because they are adjacent houses.
```

Example 2:

```
Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount you can rob = 1 + 3 = 4.
```

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

这道题是之前那道 House Robber 的拓展，现在房子排成了一个圆圈，则如果抢了第一家，就不能抢最后一家，因为首尾相连了，所以第一家和最后一家只能抢其中的一家，或者都不抢，那这里变通一下，如果把第一家和最后一家分别去掉，各算一遍能抢的最大值，然后比较两个值取其中较大的一个即为所求。那只需参考之前的 House Robber 中的解题方法，然后调用两边取较大值，代码如下：

解法一：

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() <= 1) return nums.empty() ? 0 : nums[0];
        return max(rob(nums, 0, nums.size() - 1), rob(nums, 1, nums.size()));
    }
    int rob(vector<int> &nums, int left, int right) {
        if (right - left <= 1) return nums[left];
        vector<int> dp(right, 0);
        dp[left] = nums[left];
        dp[left + 1] = max(nums[left], nums[left + 1]);
        for (int i = left + 2; i < right; ++i) {
            dp[i] = max(nums[i] + dp[i - 2], dp[i - 1]);
        }
```

```
        return dp.back();
    }
};
```

当然，我们也可以使用两个变量来代替整个 DP 数组，讲解与之前的帖子 House Robber 相同，分别维护两个变量 robEven 和 robOdd，顾名思义，robEven 就是要抢偶数位置的房子，robOdd 就是要抢奇数位置的房子。所以在遍历房子数组时，如果是偶数位置，那么 robEven 就要加上当前数字，然后和 robOdd 比较，取较大的来更新 robEven。这里就看出来了，robEven 组成的值并不是只由偶数位置的数字，只是当前要抢偶数位置而已。同理，当奇数位置时，robOdd 加上当前数字和 robEven 比较，取较大值来更新 robOdd，这种按奇偶分别来更新的方法，可以保证组成最大和的数字不相邻，最后别忘了在 robEven 和 robOdd 种取较大值返回，代码如下：

解法二：

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() <= 1) return nums.empty() ? 0 : nums[0];
        return max(rob(nums, 0, nums.size() - 1), rob(nums, 1,
nums.size()));
    }
    int rob(vector<int> &nums, int left, int right) {
        int robEven = 0, robOdd = 0;
        for (int i = left; i < right; ++i) {
            if (i % 2 == 0) {
                robEven = max(robEven + nums[i], robOdd);
            } else {
                robOdd = max(robEven, robOdd + nums[i]);
            }
        }
        return max(robEven, robOdd);
    }
};
```

另一种更为简洁的写法，讲解与之前的帖子 House Robber 相同，我们使用两个变量 rob 和 notRob，其中 rob 表示抢当前的房子，notRob 表示不抢当前的房子，那么在遍历的过程中，先用两个变量 preRob 和 preNotRob 来分别记录更新之前的值，由于 rob 是要抢当前的房子，那么前一个房子一定不能抢，所以使用 preNotRob 加上当前的数字赋给 rob，然后 notRob 表示不能抢当前的房子，那么之前的房子就可以抢也可以不抢，所以将 preRob 和 preNotRob 中的较大值赋给 notRob，参见代码如下：

解法三：

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() <= 1) return nums.empty() ? 0 : nums[0];
        return max(rob(nums, 0, nums.size() - 1), rob(nums, 1,
nums.size()));
```

```
    }
    int rob(vector<int> &nums, int left, int right) {
        int rob = 0, notRob = 0;
        for (int i = left; i < right; ++i) {
            int preRob = rob, preNotRob = notRob;
            rob = preNotRob + nums[i];
            notRob = max(preRob, preNotRob);
        }
        return max(rob, notRob);
    }
};
```

214. Shortest Palindrome

Given a string *s* , you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

Example 1:

```
Input: "aacecaaa"
Output: "aaacecaaa"
```

Example 2:

```
Input: "abcd"
Output: "dcbabcd"
```

Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases. Thanks to @Freezen for additional test cases.

这道题让我们求最短的回文串，LeetCode 中关于回文串的其他的题目有 Palindrome Number，Validate Palindrome，Palindrome Partitioning，Palindrome Partitioning II 和 Longest Palindromic Substring。题目让在给定字符串s的前面加上最少个字符，使之变成回文串，来看题目中给的两个例子，最坏的情况下是s中没有相同的字符，那么最小需要添加字符的个数为 s.size() - 1 个，第一个例子的字符串包含一个回文串，只需再在前面添加一个字符即可，还有一点需要注意的是，前面添加的字符串都是从s的末尾开始，一位一位往前添加的，那么只需要知道从s末尾开始需要添加到前面的个数。

首先还是先将待处理的字符串s翻转得到t，然后比较原字符串s和翻转字符串t，从第一个字符开始逐一比较，如果相等，说明s本身就是回文串，不用添加任何字符，直接返回即可；如果不相等，s去掉最后一位，t去掉第一位，继续比较，以此类推直至有相等，或者循环结束，这样就能将两个字符串在正确的位置拼接起来了，代码请参见评论区5楼。但这种写法却会超时 TLE，无法通过 OJ，所以需要用一些比较巧妙的方法来解。

这里使用双指针来找出字符串s的最长回文前缀的大概范围，这里所谓的最长回文前缀是指从开头开始到某个位置为止是回文串，比如 "abbac" 这个子串，可以知道前四个字符组成的回文串 "abba" 就是最长回文前缀。方法是指针i和j分别指向s串的开头和末尾，若 s[i] 和 s[j] 相等，则i自增1，j自减1，否则只有j自减1。需要注意的是，这样遍历一遍后，得到的范围 [0, i) 中的子串并不一定就是最大回文前缀，也可能还需要添加字符，举个例子来说，对于 "adcba"，在 for 循环执行之后，i=2，可以发现前面的 "ad" 并不是最长回文前缀，其本身甚至不是回文串，需要再次调用递归函数来填充使其变为回文串，但可以确定的是可以添加最少的字符数让其变为回文串。而且可以确定的是后面剩余的部分肯定不属于回文前缀，此时提取出剩下的字符，翻转一下加到最前面，而对范围 [0, i) 内的子串再次递归调用本函数，这样，在子函数最终会组成最短的回文串，从而使得整个的回文串就是最短的，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    string shortestPalindrome(string s) {
        int i = 0, n = s.size();
        for (int j = n - 1; j >= 0; --j) {
            if (s[i] == s[j]) ++i;
        }
        if (i == n) return s;
        string rem = s.substr(i);
        reverse(rem.begin(), rem.end());
        return rem + shortestPalindrome(s.substr(0, i)) + s.substr(i);
    }
};
```

Java 解法一：

```
public class Solution {
    public String shortestPalindrome(String s) {
        int i = 0, n = s.length();
        for (int j = n - 1; j >= 0; --j) {
            if (s.charAt(i) == s.charAt(j)) ++i;
        }
        if (i == n) return s;
        String rem = s.substring(i);
        String rem_rev = new StringBuilder(rem).reverse().toString();
        return rem_rev + shortestPalindrome(s.substring(0, i)) + rem;
    }
}
```

其实这道题的最快的解法是使用 KMP 算法，KMP 算法是一种专门用来匹配字符串的高效的算法，具体方法可以参见博主之前的这篇博文 KMP Algorithm 字符串匹配算法KMP小结。把s和其转置r连接起来，中间加上一个其他字符，形成一个新的字符串t，还需要一个和t长度相同的一位数组 next，其中 next[i] 表示从 t[i] 到开头的子串的相同前缀后缀的个数，具体可参考 KMP 算法中解释。最后把不相同的个数对应的字符串添加到s之前即可，代码如下：

C++ 解法二：

```
class Solution {
public:
    string shortestPalindrome(string s) {
        string r = s;
        reverse(r.begin(), r.end());
        string t = s + "#" + r;
        vector<int> next(t.size(), 0);
        for (int i = 1; i < t.size(); ++i) {
            int j = next[i - 1];
            while (j > 0 && t[i] != t[j]) j = next[j - 1];
            next[i] = (j += t[i] == t[j]);
        }
        return r.substr(0, s.size() - next.back()) + s;
    }
};
```

Java 解法二：

```
public class Solution {
    public String shortestPalindrome(String s) {
        String r = new StringBuilder(s).reverse().toString();
        String t = s + "#" + r;
        int[] next = new int[t.length()];
        for (int i = 1; i < t.length(); ++i) {
            int j = next[i - 1];
            while (j > 0 && t.charAt(i) != t.charAt(j)) j = next[j - 1];
            j += (t.charAt(i) == t.charAt(j)) ? 1 : 0;
            next[i] = j;
        }
        return r.substring(0, s.length() - next[t.length() - 1]) + s;
    }
}
```

从上面的 Java 和 C++ 的代码中，可以看出来 C++ 和 Java 在使用双等号上的明显的不同，感觉 Java 对于双等号对使用更加苛刻一些，比如 Java 中的双等号只对 primitive 类数据结构(比如 int, char 等)有效，但是即便有效，也不能将结果直接当1或者0来用。而对于一些从 Object 派生出来的类，比如 Integer 或者 String 等，不能直接用双等号来比较，而是要用其自带的 equals() 函数来比较，因为双等号判断的是不是同一个对象，而不是他们所表示的值是否相同。同样需要注意的是，Stack 的 peek() 函数取出的也是对象，不能直接和另一个 Stack 的 peek() 取出的对象直接双等，而是使用 equals 或者先将其中一个强行转换成 primitive 类，再和另一个强行比较。

下面这种方法的写法比较简洁，虽然不是明显的 KMP 算法，但是也有其的影子在里面。这种 Java 写法也是在找相同的前缀后缀，但是并没有每次把前缀后缀取出来比较，而是用两个指针分别指向对应的位置比较，然后用 end 指向相同后缀的起始位置，最后再根据 end 的值来拼接两个字符串。有意思的是这种方法对应的 C++ 写法会 TLE，跟上面正好相反，那么我们是否能得出 Java 的 substring 操作略慢，而 C++ 的 reverse 略慢呢，博主也仅仅是猜测而已。

Java 解法三：

```java
public class Solution {
    public String shortestPalindrome(String s) {
        int i = 0, end = s.length() - 1, j = end;
        char arr = s.toCharArray();
        while (i < j) {
            if (arr[i] == arr[j]) {
                ++i; --j;
            } else {
                i = 0; --end; j = end;
            }
        }
        return new StringBuilder(s.substring(end + 1)).reverse().toString() + s;
    }
}
```

215. Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

```
Input: [3,2,1,5,6,4] and k = 2
Output: 5
```

Example 2:

```
Input: [3,2,3,1,2,4,5,5,6] and k = 4
Output: 4
```

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

这道题让我们求数组中第k大的数字，怎么求呢，当然首先想到的是给数组排序，然后求可以得到第k大的数字。先看一种利用 C++ 的 STL 中的集成的排序方法，不用我们自己实现，这样的话这道题只要两行就完事了，代码如下：

解法一：

```cpp
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end());
        return nums[nums.size() - k];
    }
};
```

下面这种解法是利用了 priority_queue 的自动排序的特性，跟上面的解法思路上没有什么区别，当然我们也可以换成 multiset 来做，一个道理，参见代码如下：

解法二：

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int> q(nums.begin(), nums.end());
        for (int i = 0; i < k - 1; ++i) {
            q.pop();
        }
        return q.top();
    }
};
```

上面两种方法虽然简洁，但是确不是本题真正想考察的东西，可以说有一定的偷懒嫌疑。这道题最好的解法应该是下面这种做法，用到了快速排序 Quick Sort 的思想，这里排序的方向是从大往小排。对快排不熟悉的童鞋们随意上网搜些帖子看下吧，多如牛毛啊，总有一款适合你。核心思想是每次都要先找一个中枢点 Pivot，然后遍历其他所有的数字，像这道题从大往小排的话，就把大于中枢点的数字放到左半边，把小于中枢点的放在右半边，这样中枢点是整个数组中第几大的数字就确定了，虽然左右两部分各自不一定是完全有序的，但是并不影响本题要求的结果，因为左半部分的所有值都大于右半部分的任意值，所以我们求出中枢点的位置，如果正好是 k-1，那么直接返回该位置上的数字；如果大于 k-1，说明要求的数字在左半部分，更新右边界，再求新的中枢点位置；反之则更新右半部分，求中枢点的位置；不得不说，这个思路真的是巧妙啊～

解法三：

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (true) {
            int pos = partition(nums, left, right);
            if (pos == k - 1) return nums[pos];
            else if (pos > k - 1) right = pos - 1;
            else left = pos + 1;
        }
    }
    int partition(vector<int>& nums, int left, int right) {
        int pivot = nums[left], l = left + 1, r = right;
        while (l <= r) {
            if (nums[l] < pivot && nums[r] > pivot) {
                swap(nums[l++], nums[r--]);
            }
            if (nums[l] >= pivot) ++l;
            if (nums[r] <= pivot) --r;
        }
        swap(nums[left], nums[r]);
```

```
        return r;
    }
};
```

216. Combination Sum III

Find all possible combinations of ***k*** numbers that add up to a number ***n*** , given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Ensure that numbers within the set are sorted in ascending order.

Example 1:

Input: ***k*** = 3, ***n*** = 7

Output:

```
[[1,2,4]]
```

Example 2:

Input: ***k*** = 3, ***n*** = 9

Output:

```
[[1,2,6], [1,3,5], [2,3,4]]
```

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

这道题题是组合之和系列的第三道题，跟之前两道 Combination Sum，Combination Sum II 都不太一样，那两道的联系比较紧密，变化不大，而这道跟它们最显著的不同就是这道题的个数是固定的，为k。个人认为这道题跟那道 Combinations 更相似一些，但是那道题只是排序，对k个数字之和又没有要求。所以实际上这道题是它们的综合体，两者杂糅到一起就是这道题的解法了，n是k个数字之和，如果n小于0，则直接返回，如果n正好等于0，而且此时out中数字的个数正好为k，说明此时是一个正确解，将其存入结果res中，具体实现参见代码入下：

```
class Solution {
public:
    vector<vector<int> > combinationSum3(int k, int n) {
        vector<vector<int> > res;
        vector<int> out;
        combinationSum3DFS(k, n, 1, out, res);
        return res;
    }
    void combinationSum3DFS(int k, int n, int level, vector<int> &out,
vector<vector<int> > &res) {
```

```
        if (n < 0) return;
        if (n == 0 && out.size() == k) res.push_back(out);
        for (int i = level; i <= 9; ++i) {
            out.push_back(i);
            combinationSum3DFS(k, n - i, i + 1, out, res);
            out.pop_back();
        }
    }
};
```

218. The Skyline Problem

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are given the locations and height of all the buildings as shown on a cityscape photo (Figure A), write a program to output the skyline formed by these buildings collectively (Figure B).

The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and `Hi` is its height. It is guaranteed that `0 ≤ Li, Ri ≤ INT_MAX`, `0 < Hi ≤ INT_MAX`, and `Ri - Li > 0`. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]`.

The output is a list of "key points" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. A key point is the left endpoint of a horizontal line segment. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

Notes:

- The number of buildings in any input list is guaranteed to be in the range `[0, 10000]`.
- The input list is already sorted in ascending order by the left x position `Li`.
- The output list must be sorted by the x position.
- There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...[2 3], [4 5], [12 7], ...]`

Credits:
Special thanks to @stellari for adding this problem, creating these two awesome images and all test cases.

这道题一打开又是图又是这么长的题目的，看起来感觉应该是一道相当复杂的题，但是做完之后发现也就那么回事，虽然我不会做，是学习的别人的解法。这道求天际线的题目应该算是比较新颖的题，要是非要在之前的题目中找一道类似的题，也就只有 Merge Intervals 合并区间了吧，但是与那题不同的是，这道题不是求被合并成的空间，而是求轮廓线的一些关键的转折点，这就比较复杂了，我们通过仔细观察题目中给的那个例子可以发现，要求的红点都跟每个小区间的左右区间点有密切的关系，而且进一步发现除了每一个封闭区间的最右边的结束点是楼的右边界点，其余的都是左边界点，而且每个红点的纵坐标都是当前重合处的最高楼的高度，但是在右边界的那个楼的就不算了。在网上搜了很多帖子，发现网友Brian Gordon的帖子图文并茂，什么动画渐变啊，横向扫描啊，简直叼到没朋友啊，但是叼到极致后就懒的一句一句的去读了，这里博主还是讲解另一位网友百草园的博客吧。这里用到了multiset数据结构，其好处在于其中的元素是按堆排好序的，插入新元素进去还是有序的，而且执行删除元素也可方便的将元素删掉。这里为了区分左右边界，将左边界的高度存为负数，建立左边界和负高度的pair，再建立右边界和高度的pair，存入数组中，都存进去了以后，给数组按照左边界排序，这样我们就可以按顺序来处理那些关键的节点了。我们要在multiset中放入一个0，这样在某个没有和其他建筑重叠的右边界上，我们就可以将封闭点存入结果res中。下面我们按顺序遍历这些关键节点，如果遇到高度为负值的pair，说明是左边界，那么将正高度加入multiset中，然后取出此时集合中最高的高度，即最后一个数字，然后看是否跟pre相同，这里的pre是上一个状态的高度，初始化为0，所以第一个左边界的高度绝对不为0，所以肯定会存入结果res中。接下来如果碰到了一个更高的楼的左边界的话，新高度存入multiset的话会排在最后面，那么此时cur取来也跟pre不同，可以将新的左边界点加入结果res。第三个点遇到绿色建筑的左边界点时，由于其高度低于红色的楼，所以cur取出来还是红色楼的高度，跟pre相同，直接跳过。下面遇到红色楼的右边界，此时我们首先将红色楼的高度从multiset中删除，那么此时cur取出的绿色楼的高度就是最高啦，跟pre不同，则可以将红楼的右边界横坐标和绿楼的高度组成pair加到结果res中，这样就成功的找到我们需要的拐点啦，后面都是这样类似的情况。当某个右边界点没有跟任何楼重叠的话，删掉当前的高度，那么multiset中就只剩0了，所以跟当前的右边界横坐标组成pair就是封闭点啦，具体实现参看代码如下：

```
class Solution {
public:
    vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {
        vector<pair<int, int>> h, res;
        multiset<int> m;
        int pre = 0, cur = 0;
        for (auto &a : buildings) {
            h.push_back({a[0], -a[2]});
            h.push_back({a[1], a[2]});
        }
        sort(h.begin(), h.end());
        m.insert(0);
        for (auto &a : h) {
            if (a.second < 0) m.insert(-a.second);
            else m.erase(m.find(a.second));
            cur = *m.rbegin();
            if (cur != pre) {
                res.push_back({a.first, cur});
                pre = cur;
```

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

220. Contains Duplicate III

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the difference between **nums[i]** and **nums[j]** is at most *t* and the difference between *i* and *j* is at most *k*.

这道题跟之前两道Contains Duplicate 包含重复值和Contains Duplicate II 包含重复值之二的关联并不是很大，前两道起码跟重复值有关，这道题的焦点不是在重复值上面，反而是关注与不同的值之间的关系，这里有两个限制条件，两个数字的坐标差不能大于k，值差不能大于t。这道题如果用brute force会超时，所以我们只能另辟蹊径。这里我们使用map数据结构来解,用来记录数字和其下标之间的映射。这里需要两个指针i和j，刚开始i和j都指向0，然后i开始向右走遍历数组，如果i和j之差大于k，且m中有nums[j]，则删除并j加一。这样保证了m中所有的数的下标之差都不大于k，然后我们用map数据结构的lower_bound()函数来找一个特定范围，就是大于或等于nums[i] - t的地方，所有小于这个阈值的数和nums[i]的差的绝对值会大于t (可自行带数检验)。然后检测后面的所有的数字，如果数的差的绝对值小于等于t，则返回true。最后遍历完整个数组返回false。代码如下：

```
class Solution {
public:
    bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
        map<long long, int> m;
        int j = 0;
        for (int i = 0; i < nums.size(); ++i) {
            if (i - j > k) m.erase(nums[j++]);
            auto a = m.lower_bound((long long)nums[i] - t);
            if (a != m.end() && abs(a->first - nums[i]) <= t) return true;
            m[nums[i]] = i;
        }
        return false;
    }
};
```

221. Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Return 4.

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

这道题我刚看到的时候，马上联想到了之前的一道 Number of Islands，但是仔细一对比，发现又不太一样，那道题1的形状不确定，很适合 DFS 的特点，而这道题要找的是正方形，是非常有特点的形状，所以并不需要用到 DFS，要论相似，我倒认为这道 Maximal Rectangle 更相似一些。这道题的解法不止一种，我们先来看一种 brute force 的方法，这种方法的机理就是就是把数组中每一个点都当成正方形的左顶点来向右下方扫描，来寻找最大正方形。具体的扫描方法是，确定了左顶点后，再往下扫的时候，正方形的竖边长度就确定了，只需要找到横边即可，这时候我们使用直方图的原理，从其累加值能反映出上面的值是否全为1，之前也有一道关于直方图的题 Largest Rectangle in Histogram。通过这种方法我们就可以找出最大的正方形，参见代码如下：

解法一：

```
class Solution {
public:
    int maximalSquare(vector<vector<char> >& matrix) {
        int res = 0;
        for (int i = 0; i < matrix.size(); ++i) {
            vector<int> v(matrix[i].size(), 0);
            for (int j = i; j < matrix.size(); ++j) {
                for (int k = 0; k < matrix[j].size(); ++k) {
                    if (matrix[j][k] == '1') ++v[k];
                }
                res = max(res, getSquareArea(v, j - i + 1));
            }
        }
        return res;
    }
    int getSquareArea(vector<int> &v, int k) {
        if (v.size() < k) return 0;
        int count = 0;
        for (int i = 0; i < v.size(); ++i) {
            if (v[i] != k) count = 0;
            else ++count;
            if (count == k) return k * k;
        }
        return 0;
    }
};
```

下面这个方法用到了建立累计和数组的方法，可以参见之前那篇博客 Range Sum Query 2D - Immutable。原理是建立好了累加和数组后，我们开始遍历二维数组的每一个位置，对于任意一个位置 (i, j)，我们从该位置往 (0,0) 点遍历所有的正方形，正方形的个数为 min(i,j)+1，由于我们有了累加和矩阵，能快速的求出任意一个区域之和，所以我们能快速得到所有子正方形之和，比较正方形之和跟边长的平方是否相等，相等说明正方形中的数字均为1，更新 res 结果即可，参见代码如下：

解法二：

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int m = matrix.size(), n = matrix[0].size(), res = 0;
        vector<vector<int>> sum(m, vector<int>(n, 0));
        for (int i = 0; i < matrix.size(); ++i) {
            for (int j = 0; j < matrix[i].size(); ++j) {
                int t = matrix[i][j] - '0';
                if (i > 0) t += sum[i - 1][j];
                if (j > 0) t += sum[i][j - 1];
                if (i > 0 && j > 0) t -= sum[i - 1][j - 1];
                sum[i][j] = t;
                int cnt = 1;
                for (int k = min(i, j); k >= 0; --k) {
                    int d = sum[i][j];
                    if (i - cnt >= 0) d -= sum[i - cnt][j];
                    if (j - cnt >= 0) d -= sum[i][j - cnt];
                    if (i - cnt >= 0 && j - cnt >= 0) d += sum[i - cnt][j
- cnt];
                    if (d == cnt * cnt) res = max(res, d);
                    ++cnt;
                }
            }
        }
        return res;
    }
};
```

我们还可以进一步的优化时间复杂度到 O(n2)，做法是使用 DP，建立一个二维 dp 数组，其中 dp[i][j] 表示到达 (i, j) 位置所能组成的最大正方形的边长。我们首先来考虑边界情况，也就是当i或j为0的情况，那么在首行或者首列中，必定有一个方向长度为1，那么就无法组成长度超过1的正方形，最多能组成长度为1的正方形，条件是当前位置为1。边界条件处理完了，再来看一般情况的递推公式怎么办，对于任意一点 dp[i][j]，由于该点是正方形的右下角，所以该点的右边，下边，右下边都不用考虑，关心的就是左边，上边，和左上边。这三个位置的dp值 suppose 都应该算好的，还有就是要知道一点，只有当前 (i, j) 位置为1，dp[i][j] 才有可能大于0，否则 dp[i][j] 一定为0。当 (i, j) 位置为1，此时要看 dp[i-1][j-1], dp[i][j-1]，和 dp[i-1][j] 这三个位置，我们找其中最小的值，并加上1，就是 dp[i][j] 的当前值了，这个并不难想，毕竟不能有0存在，所以只能取交集，最后再用 dp[i][j] 的值来更新结果 res 的值即可，参见代码如下：

解法三：

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int m = matrix.size(), n = matrix[0].size(), res = 0;
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i == 0 || j == 0) dp[i][j] = matrix[i][j] - '0';
                else if (matrix[i][j] == '1') {
                    dp[i][j] = min(dp[i - 1][j - 1], min(dp[i][j - 1], dp[i - 1][j])) + 1;
                }
                res = max(res, dp[i][j]);
            }
        }
        return res * res;
    }
};
```

下面这种解法进一步的优化了空间复杂度，此时只需要用一个一维数组就可以解决，为了处理边界情况，padding 了一位，所以 dp 的长度是 m+1，然后还需要一个变量 pre 来记录上一个层的 dp 值，我们更新的顺序是行优先，就是先往下遍历，用一个临时变量t保存当前 dp 值，然后看如果当前位置为1，则更新 dp[i] 为 dp[i], dp[i-1], 和 pre 三者之间的最小值，再加上1，来更新结果 res，如果当前位置为0，则重置当前 dp 值为0，因为只有一维数组，每个位置会被重复使用，参见代码如下：

解法四：

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int m = matrix.size(), n = matrix[0].size(), res = 0, pre = 0;
        vector<int> dp(m + 1, 0);
        for (int j = 0; j < n; ++j) {
            for (int i = 1; i <= m; ++i) {
                int t = dp[i];
                if (matrix[i - 1][j] == '1') {
                    dp[i] = min(dp[i], min(dp[i - 1], pre)) + 1;
                    res = max(res, dp[i]);
                } else {
                    dp[i] = 0;
                }
                pre = t;
            }
        }
        return res * res;
    }
```

```
};
```

222. Count Complete Tree Nodes

Given a complete binary tree, count the number of nodes.

Note:

Definition of a complete binary tree from Wikipedia:
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

Example:

```
Input:
    1
   / \
  2   3
 / \  /
4  5 6

Output: 6
```

这道题给定了一棵完全二叉树，让我们求其节点的个数。很多人分不清完全二叉树和满二叉树的区别，下面让我们来看看维基百科上对二者的定义：

完全二叉树 (Complete Binary Tree)：

A Complete Binary Tree （CBT) is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

对于一颗二叉树，假设其深度为d（d>1）。除了第d层外，其它各层的节点数目均已达最大值，且第d层所有节点从左向右连续地紧密排列，这样的二叉树被称为完全二叉树；

换句话说，完全二叉树从根结点到倒数第二层满足完美二叉树，最后一层可以不完全填充，其叶子结点都靠左对齐。

完美二叉树 (Perfect Binary Tree)：

A Perfect Binary Tree(PBT) is a tree with all leaf nodes at the same depth. All internal nodes have degree 2.

二叉树的第i层至多拥有 2^(i - 1)个节点数；深度为k的二叉树至多总共有2^(k + 1) - 1个节点数，而总计拥有节点数匹配的，称为“满二叉树”；

完满二叉树 (Full Binary Tree):

A Full Binary Tree (FBT) is a tree in which every node other than the leaves has two children.

换句话说，所有非叶子结点的度都是2。（只要你有孩子，你就必然是有两个孩子。）

其实这道题的最暴力的解法就是直接用递归来统计结点的个数，根本不需要考虑什么完全二叉树还是完美二叉树，递归在手，遇 tree 不愁。直接一行搞定碉堡了，这可能是我见过最简洁的 brute force 的解法了吧，参见代码如下：

解法一：

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        return root ? (1 + countNodes(root->left) + countNodes(root->right)) : 0;
    }
};
```

我们还是要来利用一下完全二叉树这个条件，不然感觉对出题者不太尊重。通过上面对完全二叉树跟完美二叉树的定义比较，可以看出二者的关系是，完美二叉树一定是完全二叉树，而完全二叉树不一定是完美二叉树。那么这道题给的完全二叉树就有可能是完美二叉树，若是完美二叉树，节点个数很好求，为2的h次方减1，h为该完美二叉树的高度。若不是的话，只能老老实实的一个一个数结点了。思路是由 `root` 根结点往下，分别找最靠左边和最靠右边的路径长度，如果长度相等，则证明二叉树最后一层节点是满的，是满二叉树，直接返回节点个数，如果不相等，则节点个数为左子树的节点个数加上右子树的节点个数再加1(根节点)，其中左右子树节点个数的计算可以使用递归来计算，参见代码如下：

解法二：

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        int hLeft = 0, hRight = 0;
        TreeNode *pLeft = root, *pRight = root;
        while (pLeft) {
            ++hLeft;
            pLeft = pLeft->left;
        }
        while (pRight) {
            ++hRight;
            pRight = pRight->right;
        }
        if (hLeft == hRight) return pow(2, hLeft) - 1;
        return countNodes(root->left) + countNodes(root->right) + 1;
    }
};
```

我们也可以全用递归的形式来解，如下所示：

解法三：

```
class Solution {
```

```
public:
    int countNodes(TreeNode* root) {
        int hLeft = leftHeight(root);
        int hRight = rightHeight(root);
        if (hLeft == hRight) return pow(2, hLeft) - 1;
        return countNodes(root->left) + countNodes(root->right) + 1;
    }
    int leftHeight(TreeNode* root) {
        if (!root) return 0;
        return 1 + leftHeight(root->left);
    }
    int rightHeight(TreeNode* root) {
        if (!root) return 0;
        return 1 + rightHeight(root->right);
    }
};
```

这道题还有一个标签是 Binary Search，但是在论坛上看了一圈下来，并没有发现有经典的二分搜索的写法，只找到了下面这个类似二分搜索的解法，感觉应该不算严格意义上的二分搜素法吧，毕竟 left，right 变量和 while 循环都没有，只是隐约有点二分搜索法的影子在里面，即根据条件选左右分区。首先我们需要一个 getHeight 函数，这是用来统计当前结点的左子树的最大高度的，因为一直走的是左子结点，若当前结点不存在，则返回 -1。我们对当前结点调用 getHeight 函数，得到左子树的最大高度 h，若为 -1，则说明当前结点不存在，直接返回0。否则就对右子结点调用 getHeight 函数，若返回值为 h-1，说明左子树是一棵完美二叉树，则左子树的结点个数是 2^h-1 个，再加上当前结点，总共是 2^h 个，即 1<<h，此时再加上对右子结点调用递归函数的返回值即可。若对右子结点调用 getHeight 函数的返回值不为 h-1，说明右子树一定是完美树，且高度为 h-1，则总结点个数为 2^(h-1)-1，加上当前结点为 2^(h-1)，即 1<<(h-1)，然后再加上对左子结点调用递归函数的返回值即可。这样貌似也算一种二分搜索法吧，参见代码如下：

解法四：

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        int res = 0, h = getHeight(root);
        if (h < 0) return 0;
        if (getHeight(root->right) == h - 1) return (1 << h) +
countNodes(root->right);
        return (1 << (h - 1)) + countNodes(root->left);
    }
    int getHeight(TreeNode* node) {
        return node ? (1 + getHeight(node->left)) : -1;
    }
};
```

我们也可以写成迭代的形式，用一个 while 循环，感觉好处是调用 getHeight 函数的次数变少了，因为开头计算的高度h可以一直用，每下一层后，h自减1即可，参见代码如下：

解法五：

```cpp
class Solution {
public:
    int countNodes(TreeNode* root) {
        int res = 0, h = getHeight(root);
        if (h < 0) return 0;
        while (root) {
            if (getHeight(root->right) == h - 1) {
                res += 1 << h;
                root = root->right;
            } else {
                res += 1 << (h - 1);
                root = root->left;
            }
            --h;
        }
        return res;
    }
    int getHeight(TreeNode* node) {
        return node ? (1 + getHeight(node->left)) : -1;
    }
};
```

224. Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus sign `-`, non-negativeintegers and empty spaces ``.

Example 1:

```
Input: "1 + 1"
Output: 2
```

Example 2:

```
Input: " 2-1 + 2 "
Output: 3
```

Example 3:

```
Input: "(1+(4+5+2)-3)+(6+8)"
Output: 23
```

Note:

- You may assume that the given expression is always valid.

- Do not use the `eval` built-in library function.

这道题让我们实现一个基本的计算器来计算简单的算数表达式，而且题目限制了表达式中只有加减号，数字，括号和空格，没有乘除，那么就没啥计算的优先级之分了。于是这道题就变的没有那么复杂了。我们需要一个栈来辅助计算，用个变量sign来表示当前的符号，我们遍历给定的字符串s，如果遇到了数字，由于可能是个多位数，所以我们要用while循环把之后的数字都读进来，然后用sign*num来更新结果res；如果遇到了加号，则sign赋为1，如果遇到了符号，则赋为-1；如果遇到了左括号，则把当前结果res和符号sign压入栈，res重置为0，sign重置为1；如果遇到了右括号，结果res乘以栈顶的符号，栈顶元素出栈，结果res加上栈顶的数字，栈顶元素出栈。代码如下：

解法一：

```
class Solution {
public:
    int calculate(string s) {
        int res = 0, sign = 1, n = s.size();
        stack<int> st;
        for (int i = 0; i < n; ++i) {
            char c = s[i];
            if (c >= '0') {
                int num = 0;
                while (i < n && s[i] >= '0') {
                    num = 10 * num + (s[i++] - '0');
                }
                res += sign * num;
                --i;
            } else if (c == '+') {
                sign = 1;
            } else if (c == '-') {
                sign = -1;
            } else if (c == '(') {
                st.push(res);
                st.push(sign);
                res = 0;
                sign = 1;
            } else if (c == ')') {
                res *= st.top(); st.pop();
                res += st.top(); st.pop();
            }
        }
        return res;
    }
};
```

下面这种方法和上面的基本一样，只不过对于数字的处理略微不同，上面的方法是连续读入数字，而这种方法是使用了一个变量来保存读入的num，所以在遇到其他字符的时候，都要用sign*num来更新结果res，参见代码如下：

解法二：

```
class Solution {
public:
    int calculate(string s) {
        int res = 0, num = 0, sign = 1, n = s.size();
        stack<int> st;
        for (int i = 0; i < n; ++i) {
            char c = s[i];
            if (c >= '0') {
                num = 10 * num + (c - '0');
            } else if (c == '+' || c == '-') {
                res += sign * num;
                num = 0;
                sign = (c == '+') ? 1 : -1;
             } else if (c == '(') {
                st.push(res);
                st.push(sign);
                res = 0;
                sign = 1;
            } else if (c == ')') {
                res += sign * num;
                num = 0;
                res *= st.top(); st.pop();
                res += st.top(); st.pop();
            }
        }
        res += sign * num;
        return res;
    }
};
```

在做了Basic Calculator III之后，再反过头来看这道题，发现递归处理括号的方法在这道题也同样适用，我们用一个变量cnt，遇到左括号自增1，遇到右括号自减1，当cnt为0的时候，说明括号正好完全匹配，这个trick在验证括号是否valid的时候经常使用到。然后我们就是根据左右括号的位置提取出中间的子字符串调用递归函数，返回值赋给num，参见代码如下：

解法三：

```
class Solution {
public:
    int calculate(string s) {
        int res = 0, num = 0, sign = 1, n = s.size();
        for (int i = 0; i < n; ++i) {
            char c = s[i];
            if (c >= '0' && c <= '9') {
                num = 10 * num + (c - '0');
            } else if (c == '(') {
                int j = i, cnt = 0;
                for (; i < n; ++i) {
                    if (s[i] == '(') ++cnt;
```

```
                    if (s[i] == ')') --cnt;
                    if (cnt == 0) break;
                }
                num = calculate(s.substr(j + 1, i - j - 1));
            }
            if (c == '+' || c == '-' || i == n - 1) {
                res += sign * num;
                num = 0;
                sign = (c == '+') ? 1 : -1;
             }
        }
        return res;
    }
};
```

225. Implement Stack using Queues

Implement the following operations of a stack using queues.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- empty() -- Return whether the stack is empty.

Example:

```
MyStack stack = new MyStack();

stack.push(1);
stack.push(2);
stack.top();   // returns 2
stack.pop();   // returns 2
stack.empty(); // returns false
```

Notes:

- You must use *only* standard operations of a queue -- which means only `push to back`, `peek/pop from front`, `size`, and `is empty` operations are valid.
- Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
- You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and all test cases.

这道题让我们用队列来实现栈，队列和栈作为两种很重要的数据结构，它们最显著的区别就是，队列是先进先出，而栈是先进后出。题目要求中又给定了限制条件只能用 queue 的最基本的操作，像 back() 这样的操作是禁止使用的。那么怎么样才能让先进先出的特性模拟出先进后出呢，这里就需要另外一个队列来辅助操作，我们总共需要两个队列，其中一个队列用来放最后加进来的数，模拟栈顶元素。剩下所有的数都按顺序放入另一个队列中。当 push() 操作时，将新数字先加入模拟栈顶元素的队列中，如果此时队列中有数字，则将原本有的数字放入另一个队中，让新数字在这队中，用来模拟栈顶元素。当 top() 操作时，如果模拟栈顶的队中有数字则直接返回，如果没有则到另一个队列中通过平移数字取出最后一个数字加入模拟栈顶的队列中。当 pop() 操作时，先执行下 top() 操作，保证模拟栈顶的队列中有数字，然后再将该数字移除即可。当 empty() 操作时，当两个队列都为空时，栈为空。代码如下：

解法一：

```
class MyStack {
public:
    MyStack() {}
    void push(int x) {
        q2.push(x);
        while (q2.size() > 1) {
            q1.push(q2.front()); q2.pop();
        }
    }
    int pop() {
        int x = top(); q2.pop();
        return x;
    }
    int top() {
        if (q2.empty()) {
            for (int i = 0; i < (int)q1.size() - 1; ++i) {
                q1.push(q1.front()); q1.pop();
            }
            q2.push(q1.front()); q1.pop();
        }
        return q2.front();
    }
    bool empty() {
        return q1.empty() && q2.empty();
    }
private:
    queue<int> q1, q2;
};
```

这道题还有另一种解法，可以参见另一道类似的题 Implement Queue using Stacks，我个人来讲比较偏爱下面这种方法，比较好记，只要实现对了 push() 函数，后面三个直接调用队列的函数即可。这种方法的原理就是每次把新加入的数插到前头，这样队列保存的顺序和栈的顺序是相反的，它们的取出方式也是反的，那么反反得正，就是我们需要的顺序了。我们可以使用一个辅助队列，把q的元素也逆着顺序存入到辅助队列中，此时加入新元素x，再把辅助队列中的元素存回来，这样就是我们要的顺序了。当然，我们也可以直接对队列q操作，在队尾加入了新元素x后，将x前面所有的元素都按顺序取出并加到队列到末尾，这样下次就能直接取出x了，符合栈到后入先出到特性，其他三个操作也就直接调

用队列的操作即可，参见代码如下：

解法二：

```
class MyStack {
public:
    MyStack() {}
    void push(int x) {
        q.push(x);
        for (int i = 0; i < (int)q.size() - 1; ++i) {
            q.push(q.front()); q.pop();
        }
    }
    int pop() {
        int x = q.front(); q.pop();
        return x;
    }
    int top() {
        return q.front();
    }
    bool empty() {
        return q.empty();
    }
private:
    queue<int> q;
};
```

讨论：上面两种解法对于不同的输入效果不同，解法一花在 top() 函数上的时间多，所以适合于有大量 push **()** 操作，而 top() 和 pop() 比较少的输入。而第二种解法在 push() 上要花大量的时间，所以适合高频率的 top() 和 pop()，较少的 push()。两种方法各有千秋，互有利弊。

227. Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negativeintegers, `+`, `-`, `*`, `/` operators and empty spaces ``. The integer division should truncate toward zero.

Example 1:

```
Input: "3+2*2"
Output: 7
```

Example 2:

```
Input: " 3/2 "
Output: 1
```

Example 3:

```
Input: " 3+5 / 2 "
Output: 5
```

Note:

- You may assume that the given expression is always valid.
- Do not use the `eval` built-in library function.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

这道题是之前那道 Basic Calculator 的拓展，不同之处在于那道题的计算符号只有加和减，而这题加上了乘除，那么就牵扯到了运算优先级的问题，好在这道题去掉了括号，还适当的降低了难度，估计再出一道的话就该加上括号了。不管那么多，这道题先按木有有括号来处理，由于存在运算优先级，我们采取的措施是使用一个栈保存数字，如果该数字之前的符号是加或减，那么把当前数字压入栈中，注意如果是减号，则加入当前数字的相反数，因为减法相当于加上一个相反数。如果之前的符号是乘或除，那么从栈顶取出一个数字和当前数字进行乘或除的运算，再把结果压入栈中，那么完成一遍遍历后，所有的乘或除都运算完了，再把栈中所有的数字都加起来就是最终结果了，参见代码如下：

解法一：

```
class Solution {
public:
    int calculate(string s) {
        long res = 0, num = 0, n = s.size();
        char op = '+';
        stack<int> st;
        for (int i = 0; i < n; ++i) {
            if (s[i] >= '0') {
                num = num * 10 + s[i] - '0';
            }
            if ((s[i] < '0' && s[i] != ' ') || i == n - 1) {
                if (op == '+') st.push(num);
                if (op == '-') st.push(-num);
                if (op == '*' || op == '/') {
                    int tmp = (op == '*') ? st.top() * num : st.top() /
num;
                    st.pop();
                    st.push(tmp);
                }
                op = s[i];
                num = 0;
            }
        }
        while (!st.empty()) {
            res += st.top();
            st.pop();
        }
```

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

在做了 Basic Calculator III 之后，再反过头来看这道题，发现只要将处理括号的部分去掉直接就可以在这道题上使用，参见代码如下：

解法二：

```
class Solution {
public:
    int calculate(string s) {
        long res = 0, curRes = 0, num = 0, n = s.size();
        char op = '+';
        for (int i = 0; i < n; ++i) {
            char c = s[i];
            if (c >= '0' && c <= '9') {
                num = num * 10 + c - '0';
            }
            if (c == '+' || c == '-' || c == '*' || c == '/' || i == n - 1) {
                switch (op) {
                    case '+': curRes += num; break;
                    case '-': curRes -= num; break;
                    case '*': curRes *= num; break;
                    case '/': curRes /= num; break;
                }
                if (c == '+' || c == '-' || i == n - 1) {
                    res += curRes;
                    curRes = 0;
                }
                op = c;
                num = 0;
            }
        }
        return res;
    }
};
```

类似题目：

Basic Calculator III

Basic Calculator

Expression Add Operators

229. Majority Element II

Given an integer array of size *n* , find all elements that appear more than `⌊ n/3 ⌋` times.

Note: The algorithm should run in linear time and in O(1) space.

Example 1:

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

Example 2:

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

这道题让我们求出现次数大于 n/3 的数字，而且限定了时间和空间复杂度，那么就不能排序，也不能使用 HashMap，这么苛刻的限制条件只有一种方法能解了，那就是摩尔投票法 Moore Voting，这种方法在之前那道题 Majority Element 中也使用了。题目中给了一条很重要的提示，让先考虑可能会有多少个这样的数字，经过举了很多例子分析得出，任意一个数组出现次数大于 n/3 的数最多有两个，具体的证明博主就不会了，博主也不是数学专业的（热心网友用手走路提供了证明：如果有超过两个，也就是至少三个数字满足“出现的次数大于 n/3”，那么就意味着数组里总共有超过 3*(n/3) = n 个数字，这与已知的数组大小矛盾，所以，只可能有两个或者更少）。那么有了这个信息，使用投票法的核心是找出两个候选数进行投票，需要两遍遍历，第一遍历找出两个候选数，第二遍遍历重新投票验证这两个候选数是否为符合题意的数即可，选候选数方法和前面那篇 Majority Element 一样，由于之前那题题目中限定了一定会有大多数存在，故而省略了验证候选众数的步骤，这道题却没有这种限定，即满足要求的大多数可能不存在，所以要有验证，参加代码如下：

```
class Solution {
public:
    vector<int> majorityElement(vector<int>& nums) {
        vector<int> res;
        int a = 0, b = 0, cnt1 = 0, cnt2 = 0, n = nums.size();
        for (int num : nums) {
            if (num == a) ++cnt1;
            else if (num == b) ++cnt2;
            else if (cnt1 == 0) { a = num; cnt1 = 1; }
            else if (cnt2 == 0) { b = num; cnt2 = 1; }
            else { --cnt1; --cnt2; }
        }
        cnt1 = cnt2 = 0;
        for (int num : nums) {
            if (num == a) ++cnt1;
            else if (num == b) ++cnt2;
        }
        if (cnt1 > n / 3) res.push_back(a);
        if (cnt2 > n / 3) res.push_back(b);
        return res;
    }
};
```

230. Kth Smallest Element in a BST

Given a binary search tree, write a function `kthSmallest` to find the kth smallest element in it.

Note:
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Example 1:

```
Input: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
Output: 1
```

Example 2:

```
Input: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
Output: 3
```

Follow up:
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

这又是一道关于二叉搜索树 Binary Search Tree 的题， LeetCode 中关于 BST 的题有 Validate Binary Search Tree， Recover Binary Search Tree， Binary Search Tree Iterator， Unique Binary Search Trees， Unique Binary Search Trees II，Convert Sorted Array to Binary Search Tree 和 Convert Sorted List to Binary Search Tree。那么这道题给的提示是让我们用 BST 的性质来解题，最重要的性质是就是左<根<右，如果用中序遍历所有的节点就会得到一个有序数组。所以解题的关键还是中序遍历啊。关于二叉树的中序遍历可以参见我之前的博客 Binary Tree Inorder Traversal，里面有很多种方法可以用，先来看一种非递归的方法，中序遍历最先遍历到的是最小的结点，只要用一个计数器，每遍历一个结点，计数器自增1，当计数器到达k时，返回当前结点值即可，参见代码如下：

解法一：

```
class Solution {
```

```
public:
    int kthSmallest(TreeNode* root, int k) {
        int cnt = 0;
        stack<TreeNode*> s;
        TreeNode *p = root;
        while (p || !s.empty()) {
            while (p) {
                s.push(p);
                p = p->left;
            }
            p = s.top(); s.pop();
            ++cnt;
            if (cnt == k) return p->val;
            p = p->right;
        }
        return 0;
    }
};
```

当然，此题我们也可以用递归来解，还是利用中序遍历来解，代码如下：

解法二：

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        return kthSmallestDFS(root, k);
    }
    int kthSmallestDFS(TreeNode* root, int &k) {
        if (!root) return -1;
        int val = kthSmallestDFS(root->left, k);
        if (k == 0) return val;
        if (--k == 0) return root->val;
        return kthSmallestDFS(root->right, k);
    }
};
```

再来看一种分治法的思路，由于 BST 的性质，可以快速定位出第k小的元素是在左子树还是右子树，首先计算出左子树的结点个数总和 cnt，如果k小于等于左子树结点总和 cnt，说明第k小的元素在左子树中，直接对左子结点调用递归即可。如果k大于 cnt+1，说明目标值在右子树中，对右子结点调用递归函数，注意此时的k应为 k-cnt-1，应为已经减少了 cnt+1 个结点。如果k正好等于 cnt+1，说明当前结点即为所求，返回当前结点值即可，参见代码如下：

解法三：

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        int cnt = count(root->left);
```

```
        if (k <= cnt) {
            return kthSmallest(root->left, k);
        } else if (k > cnt + 1) {
            return kthSmallest(root->right, k - cnt - 1);
        }
        return root->val;
    }
    int count(TreeNode* node) {
        if (!node) return 0;
        return 1 + count(node->left) + count(node->right);
    }
};
```

这道题的 Follow up 中说假设该 BST 被修改的很频繁，而且查找第k小元素的操作也很频繁，问我们如何优化。其实最好的方法还是像上面的解法那样利用分治法来快速定位目标所在的位置，但是每个递归都遍历左子树所有结点来计算个数的操作并不高效，所以应该修改原树结点的结构，使其保存包括当前结点和其左右子树所有结点的个数，这样就可以快速得到任何左子树结点总数来快速定位目标值了。定义了新结点结构体，然后就要生成新树，还是用递归的方法生成新树，注意生成的结点的 count 值要累加其左右子结点的 count 值。然后在求第k小元素的函数中，先生成新的树，然后调用递归函数。在递归函数中，不能直接访问左子结点的 count 值，因为左子节结点不一定存在，所以要先判断，如果左子结点存在的话，那么跟上面解法的操作相同。如果不存在的话，当此时k为1的时候，直接返回当前结点值，否则就对右子结点调用递归函数，k自减1，参见代码如下：

解法四：

```
// Follow up
class Solution {
public:
    struct MyTreeNode {
        int val;
        int count;
        MyTreeNode *left;
        MyTreeNode *right;
        MyTreeNode(int x) : val(x), count(1), left(NULL), right(NULL) {}
    };

    MyTreeNode* build(TreeNode* root) {
        if (!root) return NULL;
        MyTreeNode *node = new MyTreeNode(root->val);
        node->left = build(root->left);
        node->right = build(root->right);
        if (node->left) node->count += node->left->count;
        if (node->right) node->count += node->right->count;
        return node;
    }

    int kthSmallest(TreeNode* root, int k) {
        MyTreeNode *node = build(root);
        return helper(node, k);
```

```
    }

    int helper(MyTreeNode* node, int k) {
        if (node->left) {
            int cnt = node->left->count;
            if (k <= cnt) {
                return helper(node->left, k);
            } else if (k > cnt + 1) {
                return helper(node->right, k - 1 - cnt);
            }
            return node->val;
        } else {
            if (k == 1) return node->val;
            return helper(node->right, k - 1);
        }
    }
};
```

233. Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

For example:
Given n = 13,
Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

Hint:

1. Beware of overflow.

这道题让我们比给定数小的所有数中1出现的个数，之前有道类似的题 Number of 1 Bits，那道题是求转为二进数后1的个数，博主开始以为这道题也是要用那题的方法，其实不是的，这题实际上相当于一道找规律的题。那么为了找出规律，我们就先来列举下所有含1的数字，并每10个统计下个数，如下所示：

1的个数　含1的数字　　数字范围

1　　1　　[1, 9]

11　　10 11 12 13 14 15 16 17 18 19　　[10, 19]

1　　21　　[20, 29]

1　　31　　[30, 39]

1　　41　　[40, 49]

1　　51　　[50, 59]

1　　61　　[60, 69]

1	71	[70, 79]
1	81	[80, 89]
1	91	[90, 99]
11	100 101 102 103 104 105 106 107 108 109	[100, 109]
21	110 111 112 113 114 115 116 117 118 119	[110, 119]
11	120 121 122 123 124 125 126 127 128 129	[120, 129]
...	...	...

通过上面的列举可以发现，100 以内的数字，除了10-19之间有 11 个 '1' 之外，其余都只有1个。如果不考虑 [10, 19] 区间上那多出来的 10 个 '1' 的话，那么在对任意一个两位数，十位数上的数字(加1)就代表1出现的个数，这时候再把多出的 10 个加上即可。比如 56 就有 (5+1)+10=16 个。如何知道是否要加上多出的 10 个呢，就要看十位上的数字是否大于等于2，是的话就要加上多余的 10 个 '1'。那么就可以用 (x+8)/10 来判断一个数是否大于等于2。对于三位数区间 [100, 199] 内的数也是一样，除了 [110, 119] 之间多出的10个数之外，共 21 个 '1'，其余的每 10 个数的区间都只有 11 个 '1'，所以 [100, 199] 内共有 21 + 11 * 9 = 120 个 '1'。那么现在想想 [0, 999] 区间内 '1' 的个数怎么求？根据前面的结果，[0, 99] 内共有 20 个，[100, 199] 内共有 120 个，而其他每 100 个数内 '1' 的个数也应该符合之前的规律，即也是 20 个，那么总共就有 120 + 20 * 9 = 300 个 '1'。那么还是可以用相同的方法来判断并累加1的个数，参见代码如下：

解法一：

```
class Solution {
public:
    int countDigitOne(int n) {
        int res = 0, a = 1, b = 1;
        while (n > 0) {
            res += (n + 8) / 10 * a + (n % 10 == 1) * b;
            b += n % 10 * a;
            a *= 10;
            n /= 10;
        }
        return res;
    }
};
```

解法二：

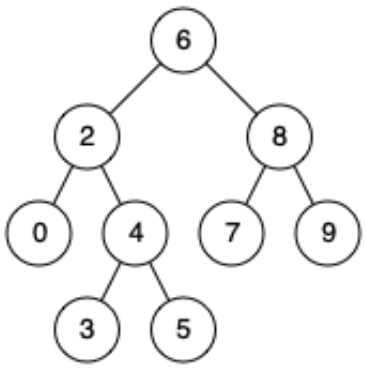
```cpp
class Solution {
public:
    int countDigitOne(int n) {
        int res = 0;
        for (long k = 1; k <= n; k *= 10) {
            long r = n / k, m = n % k;
            res += (r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0);
        }
        return res;
    }
};
```

235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Given binary search tree: root = [6,2,8,0,4,7,9,null,null,3,5]



Example 1:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a
descendant of itself according to the LCA definition.
```

Note:

- All of the nodes' values will be unique.
- p and q are different and both values will exist in the BST.

这道题让我们求二叉搜索树的最小共同父节点, LeetCode中关于BST的题有 Validate Binary Search Tree， Recover Binary Search Tree， Binary Search Tree Iterator， Unique Binary Search Trees， Unique Binary Search Trees II， Convert Sorted Array to Binary Search Tree , Convert Sorted List to Binary Search Tree 和 Kth Smallest Element in a BST。这道题我们可以用递归来求解，我们首先来看题目中给的例子，由于二叉搜索树的特点是左<根<右，所以根节点的值一直都是中间值，大于左子树的所有节点值，小于右子树的所有节点值，那么我们可以做如下的判断，如果根节点的值大于p和q之间的较大值，说明p和q都在左子树中，那么此时我们就进入根节点的左子节点继续递归，如果根节点小于p和q之间的较小值，说明p和q都在右子树中，那么此时我们就进入根节点的右子节点继续递归，如果都不是，则说明当前根节点就是最小共同父节点，直接返回即可，参见代码如下：

解法一：

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root) return NULL;
        if (root->val > max(p->val, q->val))
            return lowestCommonAncestor(root->left, p, q);
        else if (root->val < min(p->val, q->val))
            return lowestCommonAncestor(root->right, p, q);
        else return root;
    }
};
```

当然，此题也有非递归的写法，用个 while 循环来代替递归调用即可，然后不停的更新当前的根节点，也能实现同样的效果，代码如下：

解法二：

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        while (true) {
            if (root->val > max(p->val, q->val)) root = root->left;
            else if (root->val < min(p->val, q->val)) root = root->right;
            else break;
        }
        return root;
    }
};
```

236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Given the following binary tree: root = [3,5,1,6,2,0,8,null,null,7,4]

Example 1:

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Example 2:

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a
descendant of itself according to the LCA definition.
```

Note:

- All of the nodes' values will be unique.
- p and q are different and both values will exist in the binary tree.

这道求二叉树的最小共同父节点的题是之前那道 Lowest Common Ancestor of a Binary Search Tree 的 Follow Up。跟之前那题不同的地方是，这道题是普通是二叉树，不是二叉搜索树，所以就不能利用其特有的性质，我们只能在二叉树中来搜索p和q，然后从路径中找到最后一个相同的节点即为父节点，可以用递归来实现，在递归函数中，首先看当前结点是否为空，若为空则直接返回空，若为p或q中的任意一个，也直接返回当前结点。否则的话就对其左右子结点分别调用递归函数，由于这道题限制了p和q一定都在二叉树中存在，那么如果当前结点不等于p或q，p和q要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树，那么我们分别来讨论：

- 若p和q分别位于左右子树中，那么对左右子结点调用递归函数，会分别返回p和q结点的位置，而当前结点正好就是p和q的最小共同父结点，直接返回当前结点即可，这就是题目中的例子1的情况。

- 若p和q同时位于左子树，这里有两种情况，一种情况是 left 会返回p和q中较高的那个位置，而 right 会返回空，所以最终返回非空的 left 即可，这就是题目中的例子2的情况。还有一种情况是会返回p和q的最小父结点，就是说当前结点的左子树中的某个结点才是p和q的最小父结点，会被返回。

- 若p和q同时位于右子树，同样这里有两种情况，一种情况是 right 会返回p和q中较高的那个位置，而 left 会返回空，所以最终返回非空的 right 即可，还有一种情况是会返回p和q的最小父结点，就是说当前结点的右子树中的某个结点才是p和q的最小父结点，会被返回，写法很简洁，代码如下：

解法一：

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || p == root || q == root) return root;
        TreeNode *left = lowestCommonAncestor(root->left, p, q);
        TreeNode *right = lowestCommonAncestor(root->right, p , q);
        if (left && right) return root;
        return left ? left : right;
    }
};
```

上述代码可以进行优化一下，如果当前结点不为空，且既不是p也不是q，那么根据上面的分析，p和q的位置就有三种情况，p和q要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树。我们需要优化的情况就是当p和q同时为于左子树或右子树中，而且返回的结点并不是p或q，那么就是p和q的最小父结点了，已经求出来了，就不用再对右结点调用递归函数了，这是为啥呢？因为根本不会存在 left 既不是p也不是q，同时还有p或者q在 right 中。首先递归的第一句就限定了只要遇到了p或者q，就直接返回，之后又限定了只有当 left 和 right 同时存在的时候，才会返回当前结点，当前结点若不是p或q，则一定是最小父节点，否则 left 一定是p或者q。这里的逻辑比较绕，不太好想，多想想应该可以理清头绪吧，参见代码如下：

解法二：

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || p == root || q == root) return root;
        TreeNode *left = lowestCommonAncestor(root->left, p, q);
        if (left && left != p && left != q) return left;
        TreeNode *right = lowestCommonAncestor(root->right, p , q);
        if (left && right) return root;
        return left ? left : right;
    }
};
```

讨论：此题还有一种情况，题目中没有明确说明p和q是否是树中的节点，如果不是，应该返回nullptr，而上面的方法就不正确了，对于这种情况请参见 Cracking the Coding Interview 5th Edition 的第 233-234 页。

238. Product of Array Except Self

Given an array `nums` of *n* integers where *n* > 1, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Example:

```
Input:  [1,2,3,4]
Output: [24,12,8,6]
```

Note: Please solve it without division and in O(*n*).

Follow up:
Could you solve it with constant space complexity? (The output array does not count as extra space for the purpose of space complexity analysis.)

这道题给定我们一个数组，让我们返回一个新数组，对于每一个位置上的数是其他位置上数的乘积，并且限定了时间复杂度 O(n)，并且不让我们用除法。如果让用除法的话，那这道题就应该属于 Easy，因为可以先遍历一遍数组求出所有数字之积，然后除以对应位置的上的数字。但是这道题禁止我们使用除法，那么我们只能另辟蹊径。我们想，对于某一个数字，如果我们知道其前面所有数字的乘积，同时也知道后面所有的数乘积，那么二者相乘就是我们要的结果，所以我们只要分别创建出这两个数组即可，分别从数组的两个方向遍历就可以分别创建出乘积累积数组。参见代码如下：

C++ 解法一：

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n = nums.size();
        vector<int> fwd(n, 1), bwd(n, 1), res(n);
        for (int i = 0; i < n - 1; ++i) {
            fwd[i + 1] = fwd[i] * nums[i];
        }
        for (int i = n - 1; i > 0; --i) {
            bwd[i - 1] = bwd[i] * nums[i];
        }
        for (int i = 0; i < n; ++i) {
            res[i] = fwd[i] * bwd[i];
        }
        return res;
    }
};
```

Java 解法一：

```
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] res = new int[n];
        int[] fwd = new int[n], bwd = new int[n];
        fwd[0] = 1; bwd[n - 1] = 1;
        for (int i = 1; i < n; ++i) {
            fwd[i] = fwd[i - 1] * nums[i - 1];
        }
        for (int i = n - 2; i >= 0; --i) {
            bwd[i] = bwd[i + 1] * nums[i + 1];
```

```
        }
        for (int i = 0; i < n; ++i) {
            res[i] = fwd[i] * bwd[i];
        }
        return res;
    }
}
```

我们可以对上面的方法进行空间上的优化，由于最终的结果都是要乘到结果 res 中，所以可以不用单独的数组来保存乘积，而是直接累积到结果 res 中，我们先从前面遍历一遍，将乘积的累积存入结果 res 中，然后从后面开始遍历，用到一个临时变量 right，初始化为1，然后每次不断累积，最终得到正确结果，参见代码如下：

C++ 解法二：

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        vector<int> res(nums.size(), 1);
        for (int i = 1; i < nums.size(); ++i) {
            res[i] = res[i - 1] * nums[i - 1];
        }
        int right = 1;
        for (int i = nums.size() - 1; i >= 0; --i) {
            res[i] *= right;
            right *= nums[i];
        }
        return res;
    }
};
```

Java 解法二：

```
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length, right = 1;
        int[] res = new int[n];
        res[0] = 1;
        for (int i = 1; i < n; ++i) {
            res[i] = res[i - 1] * nums[i - 1];
        }
        for (int i = n - 1; i >= 0; --i) {
            res[i] *= right;
            right *= nums[i];
        }
        return res;
    }
}
```

239. Sliding Window Maximum

Given an array *nums* , there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

Example:

```
Input: _nums_ = [1,3,-1,-3,5,3,6,7], and _k_ = 3
Output: [3,3,5,5,6,7]
Explanation:

Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Note:
You may assume *k* is always valid, 1 ≤ k ≤ input array's size for non-empty array.

Follow up:
Could you solve it in linear time?

Hint:

1. How about using a data structure such as deque (double-ended queue)?
2. The queue size need not be the same as the window’s size.
3. Remove redundant elements and the queue should store only elements that need to be considered.

这道题给定了一个数组，还给了一个窗口大小k，让我们每次向右滑动一个数字，每次返回窗口内的数字的最大值。难点就在于如何找出滑动窗口内的最大值（这不废话么，求得不就是这个），那么最狂野粗暴的方法就是每次遍历窗口，找最大值呗，OJ 说呵呵哒，no way！我们希望窗口内的数字是有序的，但是每次给新窗口排序又太费时了，所以最好能有一种类似二叉搜索树的结构，可以在 lgn 的时间复杂度内完成插入和删除操作，那么使用 STL 自带的 multiset 就能满足我们的需求，这是一种基于红黑树的数据结构，可以自动对元素进行排序，又允许有重复值，完美契合。所以我们的思路就是，遍历每个数字，即窗口右移，若超过了k，则需要把左边界值删除，这里不能直接删除 nums[i-k]，因为集合中可能有重复数字，我们只想删除一个，而 erase 默认是将所有和目标值相同的元素都删掉，所以我们只能提供一个 iterator，代表一个确定的删除位置，先通过 find() 函数找到左边界 nums[i-k] 在集合中的位置，再删除即可。然后将当前数字插入到集合中，此时看若 i >= k-1，说明窗口大小正好是k，就需要将最大值加入结果 res 中，而由于 multiset 是按升序排列的，最大值在最后一个元素，我们可以通过 rbeng() 来取出，参见代码如下：

解法一：

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> res;
        multiset<int> st;
        for (int i = 0; i < nums.size(); ++i) {
            if (i >= k) st.erase(st.find(nums[i - k]));
            st.insert(nums[i]);
            if (i >= k - 1) res.push_back(*st.rbegin());
        }
        return res;
    }
};
```

我们也可以使用优先队列来做，即最大堆，不过此时我们里面放一个 pair 对儿，由数字和其所在位置组成的，这样我们就可以知道每个数字的位置了，而不用再进行搜索了。在遍历每个数字时，进行 while 循环，假如优先队列中最大的数字此时不在窗口中了，就要移除，判断方法就是将队首元素的 pair 对儿中的 second（位置坐标）跟 i-k 对比，小于等于就移除。然后将当前数字和其位置组成 pair 对儿加入优先队列中。此时看若 i >= k-1，说明窗口大小正好是k，就将最大值加入结果 res 中即可，参见代码如下：

解法二：

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> res;
        priority_queue<pair<int, int>> q;
        for (int i = 0; i < nums.size(); ++i) {
            while (!q.empty() && q.top().second <= i - k) q.pop();
            q.push({nums[i], i});
            if (i >= k - 1) res.push_back(q.top().first);
        }
        return res;
    }
};
```

题目中的 Follow up 要求我们代码的时间复杂度为 O(n)。提示我们要用双向队列 deque 来解题，并提示我们窗口中只留下有用的值，没用的全移除掉。果然 Hard 的题目我就是不会做，网上看到了别人的解法才明白，解法又巧妙有简洁，膜拜啊。大概思路是用双向队列保存数字的下标，遍历整个数组，如果此时队列的首元素是 i-k 的话，表示此时窗口向右移了一步，则移除队首元素。然后比较队尾元素和将要进来的值，如果小的话就都移除，然后此时我们把队首元素加入结果中即可，参见代码如下：

解法三：

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
```

```
        vector<int> res;
        deque<int> q;
        for (int i = 0; i < nums.size(); ++i) {
            if (!q.empty() && q.front() == i - k) q.pop_front();
            while (!q.empty() && nums[q.back()] < nums[i]) q.pop_back();
            q.push_back(i);
            if (i >= k - 1) res.push_back(nums[q.front()]);
        }
        return res;
    }
};
```

240. Search a 2D Matrix II

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

Given **target** = `5`, return `true`.

Given **target** = `20`, return `false`.

突然发现LeetCode很喜欢从LintCode上盗题，这是逼我去刷LintCode的节奏么?! 这道题让我们在一个二维数组中快速的搜索的一个数字，这个二维数组各行各列都是按递增顺序排列的，是之前那道Search a 2D Matrix 搜索一个二维矩阵的延伸，那道题的不同在于每行的第一个数字比上一行的最后一个数字大，是一个整体蛇形递增的数组。所以那道题可以将二维数组展开成一个一位数组用一次二查搜索。而这道题没法那么做，这道题有它自己的特点。如果我们观察题目中给的那个例子，我们可以发现有两个位置的数字很有特点，左下角和右上角的数。左下角的18，往上所有的数变小，往右所有数增加，那么我们就可以和目标数相比较，如果目标数大，就往右搜，如果目标数小，就往上搜。这样就可以判断目标数是否存在。当然我们也可以把起始数放在右上角，往左和下搜，停止条件设置正确就行。代码如下：

```
class Solution {
public:
```

```
    bool searchMatrix(vector<vector<int> > &matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;
        if (target < matrix[0][0] || target > matrix.back().back()) return false;
        int x = matrix.size() - 1, y = 0;
        while (true) {
            if (matrix[x][y] > target) --x;
            else if (matrix[x][y] < target) ++y;
            else return true;
            if (x < 0 || y >= matrix[0].size()) break;
        }
        return false;
    }
};
```

241. Different Ways to Add Parentheses

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are `+`, `-` and `*`.

Example 1:

```
Input: "2-1-1"
Output: [0, 2]
Explanation:
((2-1)-1) = 0
(2-(1-1)) = 2
```

Example 2:

```
Input: "2*3-4*5"
Output: [-34, -14, -10, -10, 10]
Explanation:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

这道题让给了一个可能含有加减乘的表达式，让我们在任意位置添加括号，求出所有可能表达式的不同值。这道题乍一看感觉还蛮难的，给人的感觉是既要在不同的位置上加括号，又要计算表达式的值，结果一看还是道 Medium 的题，直接尼克杨问号脸？！遇到了难题不要害怕，从最简单的例子开始分析，慢慢的找规律，十有八九就会在分析的过程中灵光一现，找到了破题的方法。这道题貌似默认输入都必须是合法的，虽然题目中没有明确的指出这一点，所以我们也就不必进行 valid 验证了。先从最简单的输入开始，若 input 是空串，那就返回一个空数组。若 input 是一个数字的话，那么括号加与不加

其实都没啥区别，因为不存在计算，但是需要将字符串转为整型数，因为返回的是一个整型数组。当然，input 是一个单独的运算符这种情况是不存在的，因为前面说了这道题默认输入的合法的。下面来看若 input 是数字和运算符的时候，比如 "1+1" 这种情况，那么加不加括号也没有任何影响，因为只有一个计算，结果一定是2。再复杂一点的话，比如题目中的例子1，input 是 "2-1-1" 时，就有两种情况了，(2-1)-1 和 2-(1-1)，由于括号的不同，得到的结果也不同，但如果我们把括号里的东西当作一个黑箱的话，那么其就变为 ()-1 和 2-()，其最终的结果跟括号内可能得到的值是息息相关的，那么再 general 一点，实际上就可以变成 () ? () 这种形式，两个括号内分别是各自的表达式，最终会分别计算得到两个整型数组，中间的问号表示运算符，可以是加，减，或乘。那么问题就变成了从两个数组中任意选两个数字进行运算，瞬间变成我们会做的题目了有木有？而这种左右两个括号代表的黑盒子就交给递归去计算，像这种分成左右两坨的 pattern 就是大名鼎鼎的分治法 Divide and Conquer 了，是必须要掌握的一个神器。类似的题目还有之前的那道 Unique Binary Search Trees II 用的方法一样，用递归来解，划分左右子树，递归构造。

好，继续来说这道题，我们不用新建递归函数，就用其本身来递归就行，先建立一个结果 res 数组，然后遍历 input 中的字符，根据上面的分析，我们希望在每个运算符的地方，将 input 分成左右两部分，从而扔到递归中去计算，从而可以得到两个整型数组 left 和 right，分别表示作用两部分各自添加不同的括号所能得到的所有不同的值，此时我们只要分别从两个数组中取数字进行当前的运算符计算，然后把结果存到 res 中即可。当然，若最终结果 res 中还是空的，那么只有一种情况，input 本身就是一个数字，直接转为整型存入结果 res 中即可，参见代码如下：

解法一：

```
class Solution {
public:
    vector<int> diffWaysToCompute(string input) {
        vector<int> res;
        for (int i = 0; i < input.size(); ++i) {
            if (input[i] == '+' || input[i] == '-' || input[i] == '*') {
                vector<int> left = diffWaysToCompute(input.substr(0, i));
                vector<int> right = diffWaysToCompute(input.substr(i + 1));
                for (int j = 0; j < left.size(); ++j) {
                    for (int k = 0; k < right.size(); ++k) {
                        if (input[i] == '+') res.push_back(left[j] + right[k]);
                        else if (input[i] == '-') res.push_back(left[j] - right[k]);
                        else res.push_back(left[j] * right[k]);
                    }
                }
            }
        }
        if (res.empty()) res.push_back(stoi(input));
        return res;
    }
};
```

我们也可以使用 HashMap 来保存已经计算过的情况，这样可以减少重复计算，从而提升运算速度，以空间换时间，岂不美哉，参见代码如下：

解法二：

```
class Solution {
public:
    unordered_map<string, vector<int>> memo;
    vector<int> diffWaysToCompute(string input) {
        if (memo.count(input)) return memo[input];
        vector<int> res;
        for (int i = 0; i < input.size(); ++i) {
            if (input[i] == '+' || input[i] == '-' || input[i] == '*') {
                vector<int> left = diffWaysToCompute(input.substr(0, i));
                vector<int> right = diffWaysToCompute(input.substr(i + 1));
                for (int j = 0; j < left.size(); ++j) {
                    for (int k = 0; k < right.size(); ++k) {
                        if (input[i] == '+') res.push_back(left[j] + right[k]);
                        else if (input[i] == '-') res.push_back(left[j] - right[k]);
                        else res.push_back(left[j] * right[k]);
                    }
                }
            }
        }
        if (res.empty()) res.push_back(stoi(input));
        memo[input] = res;
        return res;
    }
};
```

当然，这道题还可以用动态规划 Dynamic Programming 来做，但明显没有分治法来的简单，但是既然论坛里这么多陈独秀同学，博主还是要给以足够的尊重的。这里用一个三维数组 dp，其中 dp[i][j] 表示在第i个数字到第j个数字之间范围内的子串添加不同括号所能得到的不同值的整型数组，所以是个三位数组，需要注意的是我们需要对 input 字符串进行预处理，将数字跟操作分开，加到一个字符串数组 ops 中，并统计数字的个数 cnt，用这个 cnt 来初始化 dp 数组的大小，并同时要把 dp[i][j] 的数组中都加上第i个数字，通过 ops[i*2] 取得，当然还需要转为整型数。既然 dp 是个三维数组，那么肯定要用3个 for 循环来更新，这里采用的更新顺序跟之前那道 Burst Balloons 是一样的，都是从小区间往大区间更新，由于小区间之前更新过，所以我们将数字分为两部分 [i, j] 和 [j, i+len]，然后分别取出各自的数组 dp[i][j] 和 dp[j][i+len]，把对应的运算符也取出来，现在又变成了两个数组中任取两个数字进行运算，又整两个 for 循环，所以总共整了5个 for 循环嵌套，啊呀妈呀，看这整的，看不晕你算我输，参见代码如下：

解法三：

```
class Solution {
public:
    vector<int> diffWaysToCompute(string input) {
        if (input.empty()) return {};
```

```
        vector<string> ops;
        int n = input.size();
        for (int i = 0; i < n; ++i) {
            int j = i;
            while (j < n && isdigit(input[j])) ++j;
            ops.push_back(input.substr(i, j - i));
            if (j < n) ops.push_back(input.substr(j, 1));
            i = j;
        }
        int cnt = (ops.size() + 1) / 2;
        vector<vector<vector<int>>> dp(cnt, vector<vector<int>>(cnt,
vector<int>()));
        for (int i = 0; i < cnt; ++i) dp[i][i].push_back(stoi(ops[i *
2]));
        for (int len = 0; len < cnt; ++len) {
            for (int i = 0; i < cnt - len; ++i) {
                for (int j = i; j < i + len; ++j) {
                    vector<int> left = dp[i][j], right = dp[j + 1][i +
len];
                    string op = ops[j * 2 + 1];
                    for (int num1 : left) {
                        for (int num2 : right) {
                            if (op == "+") dp[i][i + len].push_back(num1 +
num2);
                            else if (op == "-") dp[i][i +
len].push_back(num1 - num2);
                            else dp[i][i + len].push_back(num1 * num2);
                        }
                    }
                }
            }
        }
        return dp[0][cnt - 1];
    }
};
```

244. Shortest Word Distance II

Design a class which receives a list of words in the constructor, and implements a method that takes two words *word1* and *word2* and return the shortest distance between these two words in the list. Your method will be called *repeatedly* many times with different parameters.

Example:
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.

```
Input: _word1_ = “coding”, _word2_ = “practice”
Output: 3

Input: _word1_ = "makes", _word2_ = "coding"
Output: 1
```

Note:
You may assume that *word1* does not equal to *word2* , and *word1* and *word2* are both in the list.

这道题是之前那道 Shortest Word Distance 的拓展，不同的是这次我们需要多次调用求最短单词距离的函数，那么用之前那道题的解法二和三就非常不高效，而当时摒弃的解法一的思路却可以用到这里，这里用 HashMap 来建立每个单词和其所有出现的位置的映射，然后在找最短单词距离时，只需要取出该单词在 HashMap 中映射的位置数组进行两两比较即可，参见代码如下：

解法一：

```
class WordDistance {
public:
    WordDistance(vector<string>& words) {
        for (int i = 0; i < words.size(); ++i) {
            m[words[i]].push_back(i);
        }
    }

    int shortest(string word1, string word2) {
        int res = INT_MAX;
        for (int i = 0; i < m[word1].size(); ++i) {
            for (int j = 0; j < m[word2].size(); ++j) {
                res = min(res, abs(m[word1][i] - m[word2][j]));
            }
        }
        return res;
    }

private:
    unordered_map<string, vector<int> > m;
};
```

我们可以优化上述的代码，使查询的复杂度由上面的 O(MN) 变为 O(M+N)，其中M和N为两个单词的长度，需要两个指针i和j来指向位置数组中的某个位置，开始初始化都为0，然后比较位置数组中的数字，将较小的一个的指针向后移动一位，直至其中一个数组遍历完成即可，参见代码如下：

解法二：

```
class WordDistance {
public:
    WordDistance(vector<string>& words) {
        for (int i = 0; i < words.size(); ++i) {
            m[words[i]].push_back(i);
```

```
        }
    }

    int shortest(string word1, string word2) {
        int i = 0, j = 0, res = INT_MAX;
        while (i < m[word1].size() && j < m[word2].size()) {
            res = min(res, abs(m[word1][i] - m[word2][j]));
            m[word1][i] < m[word2][j] ? ++i : ++j;
        }
        return res;
    }

private:
    unordered_map<string, vector<int> > m;
};
```

245. Shortest Word Distance III

Given a list of words and two words *word1* and *word2* , return the shortest distance between these two words in the list.

word1 and *word2* may be the same and they represent two individual words in the list.

Example:
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.

```
Input: _word1_ = “makes”, _word2_ = “coding”
Output: 1



Input: _word1_ = "makes", _word2_ = "makes"
Output: 3
```

Note:
You may assume *word1* and *word2* are both in the list.

这道题还是让我们求最短单词距离，有了之前两道题 Shortest Word Distance II 和 Shortest Word Distance 的基础，就大大降低了题目本身的难度。这里增加了一个条件，就是说两个单词可能会相同，所以在第一题中的解法的基础上做一些修改，博主最先想的解法是基于第一题中的解法二，由于会有相同的单词的情况，那么 p1 和 p2 就会相同，这样结果就会变成0，显然不对，所以要对 word1 和 word2 是否的相等的情况分开处理，如果相等了，由于 p1 和 p2 会相同，所以需要一个变量t来记录上一个位置，这样如果t不为 -1，且和当前的 p1 不同，可以更新结果，如果 word1 和 word2 不等，那么还是按原来的方法做，参见代码如下：

解法一：

```
class Solution {
```

```
public:
    int shortestWordDistance(vector<string>& words, string word1, string word2) {
        int p1 = -1, p2 = -1, res = INT_MAX;
        for (int i = 0; i < words.size(); ++i) {
            int t = p1;
            if (words[i] == word1) p1 = i;
            if (words[i] == word2) p2 = i;
            if (p1 != -1 && p2 != -1) {
                if (word1 == word2 && t != -1 && t != p1) {
                    res = min(res, abs(t - p1));
                } else if (p1 != p2) {
                    res = min(res, abs(p1 - p2));
                }
            }
        }
        return res;
    }
};
```

上述代码其实可以优化一下，我们并不需要变量t来记录上一个位置，将 p1 初始化为数组长度，p2 初始化为数组长度的相反数，然后当 word1 和 word2 相等的情况，用 p1 来保存 p2 的结果，p2 赋为当前的位置i，这样就可以更新结果了，如果 word1 和 word2 不相等，则还跟原来的做法一样，这种思路真是挺巧妙的，参见代码如下：

解法二：

```
class Solution {
public:
    int shortestWordDistance(vector<string>& words, string word1, string word2) {
        int p1 = words.size(), p2 = -words.size(), res = INT_MAX;
        for (int i = 0; i < words.size(); ++i) {
            if (words[i] == word1) p1 = word1 == word2 ? p2 : i;
            if (words[i] == word2) p2 = i;
            res = min(res, abs(p1 - p2));
        }
        return res;
    }
};
```

我们再来看一种更进一步优化的方法，只用一个变量 idx，这个 idx 的作用就相当于记录上一次的位置，当前 idx 不等 -1 时，说明当前i和 idx 不同，然后在 word1 和 word2 相同或者 words[i] 和 words[idx] 相同的情况下更新结果，最后别忘了将 idx 赋为i，参见代码如下；

解法三：

```
class Solution {
public:
```

```
    int shortestWordDistance(vector<string>& words, string word1, string word2) {
        int idx = -1, res = INT_MAX;
        for (int i = 0; i < words.size(); ++i) {
            if (words[i] == word1 || words[i] == word2) {
                if (idx != -1 && (word1 == word2 || words[i] != words[idx])) {
                    res = min(res, i - idx);
                }
                idx = i;
            }
        }
        return res;
    }
};
```

247. Strobogrammatic Number II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

Example:

```
Input:  n = 2
Output: ["11","69","88","96"]
```

这道题是之前那道 Strobogrammatic Number 的拓展，那道题让我们判断一个数是否是对称数，而这道题让找出长度为n的所有的对称数，这里肯定不能一个数一个数的来判断，那样太不高效了，而且 OJ 肯定也不会答应。题目中给了提示说可以用递归来做，而且提示了递归调用 n-2，而不是 n-1。先来列举一下n为 0,1,2,3,4 的情况：

n = 0: none

n = 1: 0, 1, 8

n = 2: 11, 69, 88, 96

n = 3: 101, 609, 808, 906, 111, 619, 818, 916, 181, 689, 888, 986

n = 4: 1001, 6009, 8008, 9006, 1111, 6119, 8118, 9116, 1691, 6699, 8698, 9696, 1881, 6889, 8888, 9886, 1961, 6969, 8968, 9966

注意观察 n=0 和 n=2，可以发现后者是在前者的基础上，每个数字的左右增加了 [1 1], [6 9], [8 8], [9 6]，看 n=1 和 n=3 更加明显，在0的左右增加 [1 1]，变成了 101, 在0的左右增加 [6 9]，变成了 609, 在0的左右增加 [8 8]，变成了 808, 在0的左右增加 [9 6]，变成了 906, 然后在分别在1和8的左右两边加那四组数，实际上是从 m=0 层开始，一层一层往上加的，需要注意的是当加到了n层的时候，左右两边不能加 [0 0]，因为0不能出现在两位数及多位数的开头，在中间递归的过程中，需要有在数字左右两边各加上0的那种情况，参见代码如下：

解法一：

```
class Solution {
public:
    vector<string> findStrobogrammatic(int n) {
        return find(n, n);
    }
    vector<string> find(int m, int n) {
        if (m == 0) return {""};
        if (m == 1) return {"0", "1", "8"};
        vector<string> t = find(m - 2, n), res;
        for (auto a : t) {
            if (m != n) res.push_back("0" + a + "0");
            res.push_back("1" + a + "1");
            res.push_back("6" + a + "9");
            res.push_back("8" + a + "8");
            res.push_back("9" + a + "6");
        }
        return res;
    }
};
```

这道题还有迭代的解法，感觉也相当的巧妙，需要从奇偶来考虑，奇数赋为 0,1,8，偶数赋为空，如果是奇数，就从 i=3 开始搭建，因为计算 i=3 需要 i=1，而已经初始化了 i=1 的情况，如果是偶数，从 i=2 开始搭建，也已经初始化了 i=0 的情况，所以可以用 for 循环来搭建到 i=n 的情况，思路和递归一样，写法不同而已，参见代码如下：

解法二：

```
class Solution {
public:
    vector<string> findStrobogrammatic(int n) {
        vector<string> one{"0", "1", "8"}, two{""}, res = two;
        if (n % 2 == 1) res = one;
        for (int i = (n % 2) + 2; i <= n; i += 2) {
            vector<string> t;
            for (auto a : res) {
                if (i != n) t.push_back("0" + a + "0");
                t.push_back("1" + a + "1");
                t.push_back("6" + a + "9");
                t.push_back("8" + a + "8");
                t.push_back("9" + a + "6");
            }
            res = t;
        }
        return res;
    }
};
```

248. Strobogrammatic Number III

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.

Example:

```
Input: low = "50", high = "100"
Output: 3
Explanation: 69, 88, and 96 are three strobogrammatic numbers.
```

Note:
Because the range might be a large number, the *low* and *high* numbers are represented as string.

这道题是之前那两道 Strobogrammatic Number II 和 Strobogrammatic Number 的拓展，又增加了难度，让找给定范围内的对称数的个数，我们当然不能一个一个的判断是不是对称数，也不能直接每个长度调用第二道中的方法，保存所有的对称数，然后再统计个数，这样 OJ 会提示内存超过允许的范围，所以这里的解法是基于第二道的基础上，不保存所有的结果，而是在递归中直接计数，根据之前的分析，需要初始化 n=0 和 n=1 的情况，然后在其基础上进行递归，递归的长度 len 从 low 到 high 之间遍历，然后看当前单词长度有没有达到 len，如果达到了，首先要去掉开头是0的多位数，然后去掉长度和 low 相同但小于 low 的数，和长度和 high 相同但大于 high 的数，然后结果自增1，然后分别给当前单词左右加上那五对对称数，继续递归调用，参见代码如下：

解法一：

```
class Solution {
public:
    int strobogrammaticInRange(string low, string high) {
        int res = 0;
        for (int i = low.size(); i <= high.size(); ++i) {
            find(low, high, "", i, res);
            find(low, high, "0", i, res);
            find(low, high, "1", i, res);
            find(low, high, "8", i, res);
        }
        return res;
    }
    void find(string low, string high, string path, int len, int &res) {
        if (path.size() >= len) {
            if (path.size() != len || (len != 1 && path[0] == '0'))
return;
            if ((len == low.size() && path.compare(low) < 0) || (len ==
high.size() && path.compare(high) > 0)) {
                return;
            }
```

```
            ++res;
        }
        find(low, high, "0" + path + "0", len, res);
        find(low, high, "1" + path + "1", len, res);
        find(low, high, "6" + path + "9", len, res);
        find(low, high, "8" + path + "8", len, res);
        find(low, high, "9" + path + "6", len, res);
    }
};
```

上述代码可以稍微优化一下，得到如下的代码：

解法二：

```
class Solution {
public:
    int strobogrammaticInRange(string low, string high) {
        int res = 0;
        find(low, high, "", res);
        find(low, high, "0", res);
        find(low, high, "1", res);
        find(low, high, "8", res);
        return res;
    }
    void find(string low, string high, string w, int &res) {
        if (w.size() >= low.size() && w.size() <= high.size()) {
            if (w.size() == high.size() && w.compare(high) > 0) {
                return;
            }
            if (!(w.size() > 1 && w[0] == '0') && !(w.size() == low.size()
&& w.compare(low) < 0)) {
                ++res;
            }
        }
        if (w.size() + 2 > high.size()) return;
        find(low, high, "0" + w + "0", res);
        find(low, high, "1" + w + "1", res);
        find(low, high, "6" + w + "9", res);
        find(low, high, "8" + w + "8", res);
        find(low, high, "9" + w + "6", res);
    }
};
```

250. Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

Example :

```
Input:  root = [5,1,5,5,5,null,5]

              5
             / \
            1   5
           / \   \
          5   5   5

Output: 4
```

这道题让我们求相同值子树的个数，就是所有节点值都相同的子树的个数，之前有道求最大 BST 子树的题 Largest BST Subtree，感觉挺像的，都是关于子树的问题，解题思路也可以参考一下，这里可以用递归来做，第一种解法的思路是先序遍历树的所有的节点，然后对每一个节点调用判断以当前节点为根的字数的所有节点是否相同，判断方法可以参考之前那题 Same Tree，用的是分治法的思想，分别对左右字数分别调用递归，参见代码如下：

解法一：

```
class Solution {
public:
    int res = 0;
    int countUnivalSubtrees(TreeNode* root) {
        if (!root) return res;
        if (isUnival(root, root->val)) ++res;
        countUnivalSubtrees(root->left);
        countUnivalSubtrees(root->right);
        return res;
    }
    bool isUnival(TreeNode *root, int val) {
        if (!root) return true;
        return root->val == val && isUnival(root->left, val) &&
isUnival(root->right, val);
    }
};
```

但是上面的那种解法不是很高效，含有大量的重复 check，我们想想能不能一次遍历就都搞定，这样想，符合条件的相同值的字数肯定是有叶节点的，而且叶节点也都相同(注意单独的一个叶节点也被看做是一个相同值子树)，那么可以从下往上 check，采用后序遍历的顺序，左右根，这里还是递归调用函数，对于当前遍历到的节点，如果对其左右子节点分别递归调用函数，返回均为 true 的话，那么说明当前节点的值和左右子树的值都相同，那么又多了一棵树，所以结果自增1，然后返回当前节点值和给定值(其父节点值)是否相同，从而回归上一层递归调用。这里特别说明一下在子函数中要使用的那个单竖杠或，为什么不用双竖杠的或，因为单竖杠的或是位或，就是说左右两部分都需要被计算，然后再或，C++ 这里将 true 当作1，false 当作0，然后进行 Bit OR 运算。不能使用双竖杠或的原因是，如果是双竖杠或，一旦左半边为 true 了，整个就直接是 true 了，右半边就不会再计算了，这样的话，一旦右子树中有值相同的子树也不会被计算到结果 res 中了，参见代码如下：

解法二：

```
class Solution {
public:
    int countUnivalSubtrees(TreeNode* root) {
        int res = 0;
        isUnival(root, -1, res);
        return res;
    }
    bool isUnival(TreeNode* root, int val, int& res) {
        if (!root) return true;
        if (!isUnival(root->left, root->val, res) | !isUnival(root->right, root->val, res)) {
            return false;
        }
        ++res;
        return root->val == val;
    }
};
```

我们还可以变一种写法，让递归函数直接返回以当前节点为根的相同值子树的个数，然后参数里维护一个引用类型的布尔变量，表示以当前节点为根的子树是否为相同值子树，首先对当前节点的左右子树分别调用递归函数，然后把结果加起来，现在要来看当前节点是不是和其左右子树节点值相同，当前首先要确认左右子节点的布尔型变量均为 true，这样保证左右子节点分别都是相同值子树的根，然后看如果左子节点存在，那么左子节点值需要和当前节点值相同，如果右子节点存在，那么右子节点值要和当前节点值相同，若上述条件均满足的话，说明当前节点也是相同值子树的根节点，返回值再加1，参见代码如下：

解法三：

```
class Solution {
public:
    int countUnivalSubtrees(TreeNode* root) {
        bool b = true;
        return isUnival(root, b);
    }
    int isUnival(TreeNode *root, bool &b) {
        if (!root) return 0;
        bool l = true, r = true;
        int res = isUnival(root->left, l) + isUnival(root->right, r);
        b = l && r && (root->left ? root->val == root->left->val : true) && (root->right ? root->val == root->right->val : true);
        return res + b;
    }
};
```

上面三种都是令人看得头晕的递归写法，那么我们也来看一种迭代的写法，迭代写法是在后序遍历 Binary Tree Postorder Traversal 的基础上修改而来，需要用 HashSet 来保存所有相同值子树的根节点，对于遍历到的节点，如果其左右子节点均不存在，那么此节点为叶节点，符合题意，加入结果 HashSet 中，如果左子节点不存在，那么右子节点必须已经在结果 HashSet 中，而且当前节点值需要和右子节点值相同才能将当前节点加入结果 HashSet 中，同样的，如果右子节点不存在，那么左子节点必须已经存在 HashSet 中，而且当前节点值要和左子节点值相同才能将当前节点加入结果 HashSet 中。最后，如果左右子节点均存在，那么必须都已经在 HashSet 中，并且左右子节点值都要和根节点值相同才能将当前节点加入结果 HashSet 中，其余部分跟后序遍历的迭代写法一样，参见代码如下：

解法四：

```
class Solution {
public:
    int countUnivalSubtrees(TreeNode* root) {
        if (!root) return 0;
        unordered_set<TreeNode*> res;
        stack<TreeNode*> st{{root}};
        TreeNode *head = root;
        while (!st.empty()) {
            TreeNode *t = st.top();
            if ((!t->left && !t->right) || t->left == head || t->right == head) {
                if (!t->left && !t->right) {
                    res.insert(t);
                } else if (!t->left && res.find(t->right) != res.end() && t->right->val == t->val) {
                    res.insert(t);
                } else if (!t->right && res.find(t->left) != res.end() && t->left->val == t->val) {
                    res.insert(t);
                } else if (t->left && t->right && res.find(t->left) != res.end() && res.find(t->right) != res.end() && t->left->val == t->val && t->right->val == t->val) {
                    res.insert(t);
                }
                st.pop();
                head = t;
            } else {
                if (t->right) st.push(t->right);
                if (t->left) st.push(t->left);
            }
        }
        return res.size();
    }
};
```