238. 除自身以外数组的乘积（中等）

1. 题目描述

   给你一个长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output ，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

   ```
   示例:
   输入: [1,2,3,4]
   输出: [24,12,8,6]
   ```

   提示：题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

   说明: 请不要使用除法，且在 O(n) 时间复杂度内完成此题。

   进阶：你可以在常数空间复杂度内完成这个题目吗？（ 出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

2. 简单实现

```cpp
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int len = nums.size();
        vector<int> ans = vector<int>(len, 1);
        int k = 1;
        for(int i = 0; i < len; i++){
            ans[i] = k;//  k为该数左边的乘积，此时数组存储的是除去当前元素左边的元素乘积
            k *= nums[i];
        }
        k = 1;
        for(int i = len - 1; i >= 0; i--){
            ans[i] *= k; // k为该数右边的乘积，此时数组等于左边的 * 该数右边的。
            k *= nums[i];
        }

        return ans;
    }
};
```

140. 单词拆分II（困难）

1. 题目描述

   给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。

   说明：

- 分隔时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

```
示例 1:
输入:
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
输出:
[
  "cats and dog",
  "cat sand dog"
]

示例 2:
输入:
s = "pineapplepenapple"
wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]
输出:
[
  "pine apple pen apple",
  "pineapple pen apple",
  "pine applepen apple"
]
解释: 注意你可以重复使用字典中的单词。

示例 3:
输入:
s = "catsandog"
wordDict = ["cats", "dog", "sand", "and", "cat"]
输出:
[]
```

2. 简单实现

别动不动就dfs，很容易超时，本题要dp

```
class Solution {
public:
    bool judge(vector<string>& wordDict, const string& s){
        for(int i = 0; i < wordDict.size(); i++)
            if(wordDict[i] == s)
                return true;
        return false;
    }
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        int size = wordDict.size();
        unordered_set<string> dict;
        int max_len = -1;//记录字典内最长的字符串长度
        for(int i = 0; i < size; i++){
            dict.insert(wordDict[i]);
            max_len = max(max_len, int(wordDict[i].size()));
        }
```

```
        vector<bool> dp1 = vector<bool>(s.size(), false);//初始化dp1，表示s[i]后能否添加空格
        for(int i = 0; i < s.size(); i++){
            if(dict.count(s.substr(0, i+1)) > 0){
                dp1[i] = true;
                continue;
            }
            for(int j = i-1; j>=0 && i-j<=max_len; j--){
                if(dp1[j] == true && dict.count(s.substr(j+1,i-j)) > 0){
                    dp1[i] = true;
                    break;
                }
            }
        }
        if(!dp1[s.size()-1]) return {};

        //dp[i]表示s[0..i]的所有拆分方法
        vector<vector<string>> dp = vector<vector<string>>(s.size());
        int len = s.size();
        for(int i = 0; i < len; i++){
            if(dict.count(s.substr(0, i+1)) > 0){
                if(i == len -1)
                    dp[i].push_back(s.substr(0, i+1));
                else
                    dp[i].push_back(s.substr(0, i+1) + ' ');
            }
            for(int j = i-1; j>=0 && i-j<=max_len; j--){
                string cur = s.substr(j+1,i-j);
                if(dp[j].size()>0 && dict.count(cur) > 0){
                    if(i == len -1)
                        for(int k = 0; k < dp[j].size(); k++)
                            dp[i].push_back(dp[j][k] + cur);
                    else
                        for(int k = 0; k < dp[j].size(); k++)
                            dp[i].push_back(dp[j][k] + cur + ' ');
                }
            }
        }
        return dp[s.size()-1];
    }
};
```

679. 24点游戏（困难）

1. 题目描述

你有 4 张写有 1 到 9 数字的牌。你需要判断是否能通过 *，/，+，-，(，) 的运算得到 24。

```
示例 1:
输入: [4, 1, 8, 7]
输出: True
解释: (8-4) * (7-1) = 24

示例 2:
输入: [1, 2, 1, 2]
输出: False
```

注意:

- 除法运算符 / 表示实数除法，而不是整数除法。例如 4 / (1 - 2/3) = 12 。
- 每个运算符对两个数进行运算。特别是我们不能用 - 作为一元运算符。例如，[1, 1, 1, 1] 作为输入时，表达式 -1 - 1 - 1 - 1 是不允许的。
- 你不能将数字连接在一起。例如，输入为 [1, 2, 1, 2] 时，不能写成 12 + 12 。

2. 简单实现

四个数字、有限运算符——暴力搜索，由于括号的作用就是优先级，可以用BFS的方式模拟这种优先级，每一步都用加减乘除合并两个数

注意题目中4张数字是无序的！！！

```cpp
class Solution {
public:
    bool judgePoint24(vector<int>& nums) {
        queue<vector<double>> q;
        vector<double> cur;
        for(int i = 0; i < nums.size(); i++)
            cur.push_back(double(nums[i]));
        q.push(cur);
        int n = 4;
        while(n--){
            if(n == 0){//只剩一个数字了
                while(!q.empty()){
                    if(abs(q.front()[0] - 24) < 1e-6)
                        return true;
                    q.pop();
                }
            }
            else{
                int size = q.size();
                while(size--){
                    vector<double> cur = q.front();
                    q.pop();
                    for(int i = 0; i < n; i++){
                        for(int j = i+1; j <= n; j++){//两个数字的组合
                            if(i == j) continue;
                            vector<double> temp = cur;
                            temp.erase(temp.begin() + j);
                            //-,/运算有序，所以i,j和j,i顺序都要来一遍
                            temp[i] = cur[i] + cur[j];
                            q.push(temp);
                            temp[i] = cur[i] - cur[j];
```

```
                            q.push(temp);
                            temp[i] = cur[i] * cur[j];
                            q.push(temp);
                            temp[i] = cur[i] / cur[j];
                            q.push(temp);
                            temp[i] = cur[j] - cur[i];
                            q.push(temp);
                            temp[i] = cur[j] / cur[i];
                            q.push(temp);
                        }
                    }
                }
            }
        }
        return false;
    }
};
```

104. 二叉树的最大深度（简单）

1. 题目描述

   给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

   说明: 叶子节点是指没有子节点的节点。

```
示例:
给定二叉树 [3,9,20,null,null,15,7],
   3
  / \
  9  20
    /  \
   15   7
返回它的最大深度 3
```

2. 简单实现

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root)
            return 0;
        else
            return 1+max(maxDepth(root->left),maxDepth(root->right));
    }
};
```

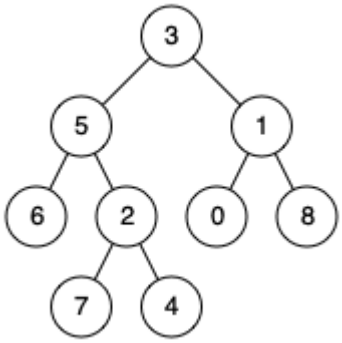
236. 二叉树的最近公共祖先（中等）

1. 题目描述

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]




```
示例 1:
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

2. 简单实现

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

322. 零钱兑换（中等）

1. 题目描述

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

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

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

说明: 你可以认为每种硬币的数量是无限的。

2. 简单实现

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp = vector<int>(amount+1, -1);
        dp[0] = 0;
        for(int i = 1; i <= amount; i++){
            for(int j = 0; j < coins.size(); j++){
                if(coins[j] <= i && dp[i-coins[j]] != -1){
                        if(dp[i] == -1)
                            dp[i] = dp[i-coins[j]] + 1;
                        else
                            dp[i] = min(dp[i],  dp[i-coins[j]] + 1);
                }
            }
        }
        return dp[amount];
    }
};
```

687. 最长同值路径（简单）

1. 题目描述

给定一个二叉树，找到最长的路径，这个路径中的每个节点具有相同值。 这条路径可以经过也可以不经过根节点。

注意：两个节点之间的路径长度由它们之间的边数表示。

示例 1:

输入:

```
          5
         / \
        4   5
       / \   \
      1   1   5
```

输出: 2

示例 2:

输入:

```
          1
         / \
        4   5
       / \   \
      4   4   5
```

输出: 2 注意: 给定的二叉树不超过10000个结点。 树的高度不超过1000。

2. 简单实现

```
class Solution {
public:
    int ans = 0;
    int helper(TreeNode* root){//返回以root为起点，可以达到的最长路径
        int h = 0;//答案是
        if(!root) return 0;
        int len = 0;//记录root为中间节点，可以达到的最长路径
        int l = helper(root->left);//以左子树为起点，可以达到的最长路径
        int r = helper(root->right);//以右子树为起点，可以达到的最长路径
        if(root->left && root->left->val == root->val){//可以和左子树相连
            h = l + 1;
            len += l + 1;
        }
        if(root->right && root->right->val == root->val){//可以和右子树相连
            h = max(h, r + 1);
            len += r + 1;
        }
        ans = max(ans, len);
        return h;
    }
    int longestUnivaluePath(TreeNode* root) {
        helper(root);
        return ans;
    }
};
```

44. 通配符匹配（困难）

1. 题目描述

给定一个字符串 (s) 和一个字符模式 (p) ，实现一个支持 '?' 和 '*' 的通配符匹配。

- '?' 可以匹配任何单个字符。
- '*' 可以匹配任意字符串（包括空字符串）。

两个字符串完全匹配才算匹配成功。

说明:

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 *。

```
示例 1:
输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:
输入:
s = "aa"
p = ""
输出: true
解释: '' 可以匹配任意字符串。

示例 3:
输入:
s = "cb"
p = "?a"
输出: false
解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。

示例 4:
输入:
s = "adceb"
p = "ab"
输出: true
解释: 第一个 '' 可以匹配空字符串, 第二个 '' 可以匹配字符串 "dce".

示例 5:
输入:
s = "acdcb"
p = "a*c?b"
输入: false
```

2. 简单实现——自动机

从10. 正则表达式匹配改的，只改NFA构造过程，遍历的不用变

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

```
        unordered_map<int, unordered_map<char, int>> g;//NFA
        //构造NFA
        int id = 0;
        int len_p = p.size();
        for(int i = 0; i < len_p; ){
            if(p[i] != '?' && p[i] != '*')
                g[id].insert(make_pair(p[i], id+1));
            else if(p[i] == '*'){
                for(int j = 0; j < 26; j++)
                    g[id].insert(make_pair('a'+j, id));
                g[id].insert(make_pair('#', id+1));
            }
            else if(p[i] == '?')
                for(int j = 0; j < 26; j++)
                    g[id].insert(make_pair('a'+j, id+1));
            id++;
            i++;
        }
        int end = id;//终止状态
        //利用NFA判断s
        unordered_set<int> curStates;
        curStates.insert(0);
        int tmp = 0;
        auto tmp_it = g[tmp].find('#');
        while(tmp_it != g[tmp].end()){//从tmp开始能够不通过任何字符转移到的所有状态
            curStates.insert(tmp_it->second);
            tmp = tmp_it->second;
            tmp_it = g[tmp].find('#');
        }
        for(int i = 0; i < s.size() && !curStates.empty(); i++){//遍历s
            unordered_set<int> nextStates;
            for(auto it = curStates.begin(); it != curStates.end(); it++){//每一个可
能的当前状态
                int cnt = g[*it].count(s[i]);//当前状态通过s[i]走一步后可以到达的状态数
                if(cnt > 0){
                    auto cur = g[*it].find(s[i]);
                    while(cnt--){//遍历所有可以一步到达的状态
                        int tmp = cur->second;
                        nextStates.insert(tmp);//加入nextStates
                        //把从tmp开始不通过任何字符能转移到的所有状态也加入nextStates
                        auto tmp_it = g[tmp].find('#');
                        while(tmp_it != g[tmp].end()){
                            tmp = tmp_it->second;
                            nextStates.insert(tmp);//通过'#'只能转移到一个状态
                            tmp_it = g[tmp].find('#');
                        }
                        cur++;
                    }
                }
            }
            curStates = nextStates;
        }
        //判断遍历完s后所有可以到达的状态，里面有没有终止状态
```

```
        for(auto it = curStates.begin(); it != curStates.end(); it++)
            if(*it == end)
                return true;
        return false;
    }
};
```

3. 最简解法

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

284. 顶端迭代器（中等）

1. 题目描述

给定一个迭代器类的接口，接口包含两个方法： next() 和 hasNext()。设计并实现一个支持 peek() 操作的顶端迭代器 -- 其本质就是把原本应由 next() 方法返回的元素 peek() 出来。

```
示例:
假设迭代器被初始化为列表 [1,2,3]。
调用 next() 返回 1，得到列表中的第一个元素。
现在调用 peek() 返回 2，下一个元素。在此之后调用 next() 仍然返回 2。
最后一次调用 next() 返回 3，末尾元素。在此之后调用 hasNext() 应该返回 false。
```

进阶：你将如何拓展你的设计？使之变得通用化，从而适应所有的类型，而不只是整数型？

2. 简单实现

关键是不会调用父类同名函数233333

```
/*
 * Below is the interface for Iterator, which is already defined for you.
 * **DO NOT** modify the interface for Iterator.
 *
```

```
 *  class Iterator {
 *      struct Data;
 *      Data* data;
 *      Iterator(const vector<int>& nums);
 *      Iterator(const Iterator& iter);
 *
 *      // Returns the next element in the iteration.
 *      int next();
 *
 *      // Returns true if the iteration has more elements.
 *      bool hasNext() const;
 *  };
 */
class PeekingIterator : public Iterator {
public:
    bool end;//是否有下一个数
    int nex;//下一个数
    PeekingIterator(const vector<int>& nums) : Iterator(nums) {
        // Initialize any member here.
        // **DO NOT** save a copy of nums and manipulate it directly.
        // You should only use the Iterator interface methods.
        if(Iterator::hasNext()){//有下一个数
            nex = Iterator::next();
            end = false;
        }
        else
            end = true;
    }
    // Returns the next element in the iteration without advancing the iterator.
    int peek() {
        return nex;
    }
    // hasNext() and next() should behave the same as in the Iterator interface.
    // Override them if needed.
    int next() {
        int ans = nex;
        if(Iterator::hasNext())
            nex = Iterator::next();
        else
            end = true;
        return ans;
    }
    bool hasNext() const {
        return !end;
    }
};
```

305. 岛屿数量II（困难）

要会员

332. 重新安排行程（中等）

1. 题目描述

给定一个机票的字符串二维数组 [from, to]，子数组中的两个成员分别表示飞机出发和降落的机场地点，对该行程进行重新规划排序。所有这些机票都属于一个从JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 出发。

说明:

- 如果存在多种有效的行程，你可以按字符自然排序返回最小的行程组合。例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前
- 所有的机场都用三个大写字母表示（机场代码）。
- 假定所有机票至少存在一种合理的行程。

```
示例 1:
输入: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
输出: ["JFK", "MUC", "LHR", "SFO", "SJC"]

示例 2:
输入: [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
输出: ["JFK","ATL","JFK","SFO","ATL","SFO"]
解释: 另一种有效的行程是 ["JFK","SFO","ATL","JFK","ATL","SFO"]。但是它自然排序更大更靠后。
```

2. 简单实现

有优先序的DFS

```
class Solution {
public:
    static bool cmp(string& a, string& b){//降序
        return a > b;
    }
    bool done;
    vector<string> ans;
    //当前在start，还有n个行程未安排
    void dfs(unordered_map<string, vector<string>>& g, const string& start, int n){
        if(done) return;
        if(n == 0){
            done = true;
            return;
        }
        if(g[start].size() == 0) return;
        for(int i = g[start].size()-1; i >= 0; i--){
            string next = g[start][i];
            g[start].erase(g[start].begin() + i);
            ans.push_back(next);
            dfs(g, next, n-1);
            if(done) return;
            g[start].insert(g[start].begin() + i, next);
            ans.pop_back();
        }
    }
    vector<string> findItinerary(vector<vector<string>>& tickets) {
        int n = tickets.size();
```

```
        unordered_map<string, vector<string>> g;//<start, <end>>
        for(int i = 0; i < tickets.size(); i++)
            g[tickets[i][0]].push_back(tickets[i][1]);
        for(auto it = g.begin(); it != g.end(); it++)
            sort(it->second.begin(), it->second.end(), cmp);
        string start = "JFK";
        ans.push_back(start);
        dfs(g, start, n);
        return ans;
    }
};
```

242. 有效的字母异位词（简单）

1. 题目描述

   给定两个字符串 s 和 t，编写一个函数来判断 t 是否是 s 的字母异位词。

   ```
   示例 1:
   输入: s = "anagram", t = "nagaram"
   输出: true

   示例 2:
   输入: s = "rat", t = "car"
   输出: false
   ```

   说明: 你可以假设字符串只包含小写字母。

   进阶: 如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况?

2. 简单实现

   排序不如直接哈希计数效率高，此外，对于进阶问题，也可以使用哈希表而不是固定大小的计数器。想象一下，分配一个大的数组来适应整个 Unicode 字符范围，这个范围可能超过 100万。哈希表是一种更通用的解决方案，可以适应任何字符范围

   ```
   class Solution {
   public:
       bool isAnagram(string s, string t) {
           if(s.length() != t.length())
               return false;
           int count[27] = {0};
           for(int i = 0; i < s.length(); i++)
               count[s[i]-'a']++;
           for(int i = 0; i < t.length(); i++)
               if(--count[t[i]-'a'] < 0)
                   return false;
           return true;
       }
   };
   ```

280. 摆动排序（中等）

要会员

406. 根据身高重建队列（中等）

1. 题目描述

   假设有打乱顺序的一群人站成一个队列。 每个人由一个整数对(h, k)表示，其中h是这个人的身高，k是排在这个人前面且身高大于或等于h的人数。 编写一个算法来重建这个队列。

   注意：总人数少于1100人。

```
示例
输入:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
输出:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        map<int, vector<int>> m;//<k, <h>>
        for(int i = 0; i < people.size(); i++)
            m[people[i][1]].push_back(people[i][0]);
        vector<vector<int>> ans;
        sort(m[0].begin(), m[0].end());
        for(int i = 0; i < m[0].size(); i++)
            ans.push_back({m[0][i], 0});
        for(auto it = m.begin(); it != m.end(); it++){
            if(it == m.begin())
                continue;
            sort(it->second.begin(), it->second.end());
            int k = it->first;
            for(int i = 0; i < it->second.size(); i++){
                int idx = 0;
                int cnt = 0;
                while(idx < ans.size() && cnt < k){
                    if(ans[idx][0] >= it->second[i])
                        cnt++;
                    idx++;
                }
                if(idx >= ans.size())
                    ans.push_back({it->second[i], k});
                else{
                     while(idx < ans.size() && ans[idx][0] <= it->second[i])
                         idx++;
                    ans.insert(ans.begin()+idx, {it->second[i], k});
                }
            }
        }
```

```
            return ans;
        }
    };
```

524. 通过删除字母匹配到字典里最长单词（中等）

1. 题目描述

   给定一个字符串和一个字符串字典，找到字典里面最长的字符串，该字符串可以通过删除给定字符串的某些字符来得到。如果答案不止一个，返回长度最长且字典顺序最小的字符串。如果答案不存在，则返回空字符串。

   ```
   示例 1:
   输入:
   s = "abpcplea", d = ["ale","apple","monkey","plea"]
   输出:
   "apple"

   示例 2:
   输入:
   s = "abpcplea", d = ["a","b","c"]
   输出:
   "a"
   ```

   说明:

   - 所有输入的字符串只包含小写字母。
   - 字典的大小不会超过 1000。
   - 所有输入的字符串长度不会超过 1000。

2. 简单实现

   将字典内字符串按长度从长到短、字典序从低到高的顺序排序，依次与s进行匹配

   匹配方法为贪心，即一旦s的字符与`d[i][idx]`相同，则idx可以+1

   ```
   class Solution {
   public:
       static bool cmp(string& a, string& b){
           if(a.size() != b.size())
               return a.size() > b.size();
           else
               return a < b;
       }
       string findLongestWord(string s, vector<string>& d) {
           int len1 = s.size();
           sort(d.begin(), d.end(), cmp);//排序
           for(int i = 0; i < d.size(); i++){
               int len2 = d[i].size();
               int idx = 0;
               for(int k = 0; k < len1; k++){//匹配
                   if(s[k] == d[i][idx]){
                       if(++idx == len2)//匹配上了
   ```

```
                        return d[i];
                    }
                }
            }
            return "";
        }
};
```

46. 全排列（中等）

1. 题目描述

   给定一个 没有重复 数字的序列，返回其所有可能的全排列。

```
示例:
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> temp;

    void dfs(vector<int>& nums) {
        int len = nums.size();
        if(len <= 0) ans.push_back(temp);
        else {
            for(int i = 0; i < len; i++) {
                int buf = nums[i];
                temp.push_back(buf);
                nums.erase(nums.begin()+i);

                dfs(nums);

                temp.pop_back();
                nums.insert(nums.begin()+i,buf);
            }
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        if(nums.size()<=0) return ans;
```

```
            else dfs(nums);
            return ans;
        }
    };
```

210. 课程表II（中等）

1. 题目描述

   现在你总共有 n 门课需要选，记为 0 到 n-1。

   在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]

   给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

   可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

```
示例 1:
输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。

示例 2:
输入: 4, [[1,0],[2,0],[3,1],[3,2]]
输出: [0,1,2,3] or [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。
```

   说明:

   - 输入的先决条件是由边缘列表表示的图形，而不是邻接矩阵。详情请参见图的表示法。
   - 你可以假定输入的先决条件中没有重复的边。

   提示:

   - 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
   - 通过 DFS 进行拓扑排序 - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
   - 拓扑排序也可以通过 BFS 完成。

2. 简单实现

   拓扑排序

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        unordered_map<int, vector<int>> g;
        vector<int> ins(numCourses, 0);
        for(int i = 0; i < prerequisites.size(); i++){
            g[prerequisites[i][1]].push_back(prerequisites[i][0]);
            ins[prerequisites[i][0]]++;
        }
        queue<int> q;
```

```
        for(int i = 0; i < numCourses; i++)
            if(ins[i] == 0)
                q.push(i);
        vector<int> ans;
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                int cur = q.front();
                q.pop();
                ans.push_back(cur);
                for(auto it = g[cur].begin(); it != g[cur].end(); it++){
                    if(--ins[*it] == 0){
                        q.push(*it);
                    }
                }
            }
        }
        if(ans.size() == numCourses)
            return ans;
        else
            return {};
    }
};
```

287. 寻找重复数（中等）

1. 题目描述

   给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

```
示例 1:
输入: [1,3,4,2,2]
输出: 2

示例 2:
输入: [3,1,3,4,2]
输出: 3
```

   说明：

   - 不能更改原数组（假设数组是只读的）。
   - 只能使用额外的 O(1) 的空间。
   - 时间复杂度小于 O(n2) 。
   - 数组中只有一个重复的数字，但它可能不止重复出现一次。

2. 二分法

```
class Solution {
public:
    int count(vector<int>& nums, int n){
        int ans = 0;
```

```
        for(int i = 0; i < nums.size(); i++)
            if(nums[i] <= n)
                ans++;
        return ans;
    }
    int findDuplicate(vector<int>& nums) {
        int l = 0, r =  nums.size() - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(count(nums, mid) <= mid)
                l = mid + 1;
            else
                r = mid;
        }
        return l;
    }
};
```

3. 弗洛伊德找环

首先，我们可以很容易地证明问题的约束意味着必须存在一个循环。因为 nums 中的每个数字都在 11 和 nn 之间，所以它必须指向存在的索引。此外，由于 00 不能作为 nums 中的值出现，nums[0] 不能作为循环的一部分。

把数组内的数值看作地址，则数组可以看作指针数组，每个地址值看作链表里面的next，则进而看做带环链表，则题目在找环入口

```
class Solution {
    public int findDuplicate(int[] nums) {
        // Find the intersection point of the two runners.
        int tortoise = nums[0];
        int hare = nums[0];
        do {
            tortoise = nums[tortoise];
            hare = nums[nums[hare]];
        } while (tortoise != hare);

        // Find the "entrance" to the cycle.
        int ptr1 = nums[0];
        int ptr2 = tortoise;
        while (ptr1 != ptr2) {
            ptr1 = nums[ptr1];
            ptr2 = nums[ptr2];
        }

        return ptr1;
    }
}
```

300. 最长上升子序列（中等）

1. 题目描述

给定一个无序的整数数组，找到其中最长上升子序列的长度。

```
示例:
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

说明:

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 O(n2) 。

进阶: 你能将算法的时间复杂度降低到 O(n log n) 吗?

2. 简单实现

考虑一个简单的贪心，如果我们要使上升子序列尽可能的长，则我们需要让序列上升得尽可能慢，因此我们希望每次在上升子序列最后加上的那个数尽可能的小。

基于上面的贪心思路，我们维护一个数组 $d[i]$ ，表示长度为 i 的最长上升子序列的末尾元素的最小值，用 len 记录目前最长上升子序列的长度，起始时 len 为 1，$d[1] = nums[0]$。

同时我们可以注意到 $d[i]$ 是关于 i 单调递增的。因为如果 $d[j] \geq d[i]$ 且 $j < i$，我们考虑从长度为 i 的最长上升子序列的末尾删除 $i - j$ 个元素，那么这个序列长度变为 j ，且第 j 个元素 x（末尾元素）必然小于 $d[i]$，也就小于 $d[j]$。那么我们就找到了一个长度为 j 的最长上升子序列，并且末尾元素比 $d[j]$ 小，从而产生了矛盾。因此数组 $d[]$ 的单调性得证。

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

根据 d 数组的单调性，我们可以使用二分查找寻找下标 i，优化时间复杂度。

最后整个算法流程为：

- 设当前已求出的最长上升子序列的长度为 len（初始时为 1），从前往后遍历数组 $nums$，在遍历到 $nums[i]$ 时：
  - 如果 $nums[i] > d[len]$ ，则直接加入到 d 数组末尾，并更新 $len = len + 1$；
  - 否则，在 d 数组中二分查找，找到第一个比 $nums[i]$ 小的数 $d[k]$ ，并更新 $d[k + 1] = nums[i]$。

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        if(nums.size() == 0) return 0;
        if(nums.size() == 1) return 1;
        vector<int> tail;
        tail.push_back(nums[0]);
        for(int i = 1; i < nums.size(); i++){
            if(tail[tail.size() - 1] < nums[i])
```

```
                tail.push_back(nums[i]);
            else{
                auto it = lower_bound(tail.begin(), tail.end(), nums[i]);
                if(*it > nums[i])
                    *it = nums[i];
            }
        }
        return tail.size();
    }
};
```

849. 到最近的人的最大距离（简单）

1. 题目描述

   在一排座位（ seats）中，1 代表有人坐在座位上，0 代表座位上是空的。

   至少有一个空座位，且至少有一人坐在座位上。

   亚历克斯希望坐在一个能够使他与离他最近的人之间的距离达到最大化的座位上。

   返回他到离他最近的人的最大距离。

```
示例 1:
输入：[1,0,0,0,1,0,1]
输出：2
解释：
如果亚历克斯坐在第二个空位（seats[2]）上，他到离他最近的人的距离为 2 。
如果亚历克斯坐在其它任何一个空位上，他到离他最近的人的距离为 1 。
因此，他到离他最近的人的最大距离是 2 。

示例 2:
输入：[1,0,0,0]
输出：3
解释：
如果亚历克斯坐在最后一个座位上，他离最近的人有 3 个座位远。
这是可能的最大距离，所以答案是 3 。
```

   提示：

   - 1 <= seats.length <= 20000
   - seats 中只含有 0 和 1，至少有一个 0，且至少有一个 1。

2. 简单实现——滑动窗口

```
class Solution {
public:
    int maxDistToClosest(vector<int>& seats) {
        int n = seats.size();
        int l = 0;
        while(l < n && seats[l] != 1) //找到最左侧的1
            l++;
        int ans = l;//坐seats[0]，与l的距离
        int r = l+1;
```

```
        while(r < n){
            while(r < n && seats[r] != 1)
                r++;
            if(r < n)//坐l,r中间
                ans = max(ans, (r - l) / 2);
            else//坐最右侧
                ans = max(ans, r-l-1);
            l = r;
            r++;
        }
        return ans;
    }
};
```

166. 分数到小数（中等）

1. 题目描述

   给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

   如果小数部分为循环小数，则将循环的部分括在括号内。

```
示例 1:
输入: numerator = 1, denominator = 2
输出: "0.5"

示例 2:
输入: numerator = 2, denominator = 1
输出: "2"

示例 3:
输入: numerator = 2, denominator = 3
输出: "0.(6)"
```

2. 简单实现

   模拟手算，遇到余数相同时就说明出现循环了

```
class Solution {
public:
    void getDecimal(long& a, long& b, string& ans){
        unordered_map<int, int> m;
        int idx = ans.size();
        while(1){
            a *= 10;
            ans += to_string(a/b);
            a %= b;
            if(a == 0)
                break;
            if(m.count(a) <= 0)
                m[a] = idx++;
            else{//余数部分出现重复，进入循环了
```

```
                if(ans[m[a]] == ans.back())
                    ans = ans.substr(0, m[a]) + '(' + ans.substr(m[a], ans.size()-
m[a]-1) + ')';
                else
                    ans = ans.substr(0, m[a]+1) + '(' + ans.substr(m[a]+1,
ans.size()-m[a]-1) + ')';
                break;
            }
        }
    }
    string fractionToDecimal(int numerator, int denominator) {
        if(numerator == 0) return "0";
        // if(denominator == 0) return "inf";
        string ans = "";
        if((numerator>0 && denominator<0) || (numerator<0 && denominator>0))
            ans += "-";
        long a = numerator;
        long b = denominator;
        a = labs(a);
        b = labs(b);
        long integer = a / b;
        ans += to_string(integer);//整数部分
        a %= b;
        if(a == 0) return ans;
        ans += ".";
        getDecimal(a, b, ans);//获取小数部分
        return ans;
    }
};
```

224. 基本计算器（困难）

1. 题目描述

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式可以包含左括号 (，右括号)，加号 + ，减号 -，非负整数和空格 。

```
示例 1:
输入: "1 + 1"
输出: 2

示例 2:
输入: " 2-1 + 2 "
输出: 3

示例 3:
输入: "(1+(4+5+2)-3)+(6+8)"
输出: 23
```

说明:

- 你可以假设所给定的表达式都是有效的。

- 请不要使用内置的库函数 eval。

2. 简单实现

变量含义解释：

为了方便，下文中所有local作用域代表离current 字符最近的左右括号圈起来的区域。

- `res` 代表现阶段的local作用域中的和，比如我要求 `(1+3+3)` 那么当遇到右括号的时候 `res` 的值最后是7
- `curnum` 代表每个被运算操作符隔开的元素，所以每遇见一个运算符会置零一次，同时把本身加到res里面
- `sign` 是和 `curnum` 对应的

算法思路：

每次遇到一个左括号就知道自己进入了一个新的local作用域，把之前没有处理完的和先推到栈里面，备用
遇到一个右括号，说明现在的作用域都处理好了，把推到栈里的没处理完的接着用作 `res` ，然后现在这个处理完的括号内的和作为 `curnum` 留着以后处理
感觉非常像程序运行的时候 call一个函数入栈和出栈过程了orz
以及有一个tricky的地方就是在字符串的末尾加了一个运算符，这样他会自行把最后的加起来。

ps：

- 我的输入变成了 st 主要是因为把stack命名成了 `s` 之后全写完了才看到，然后懒得一逼索性直接改了输入变量名，懒人鉴定完毕
- 代码没有优化有点丑，希望大家不要嫌弃
- 算法思路中的备用。。。大概是看微博里的做菜教程看多了，吃货鉴定完毕

```
class Solution {
public:
    int calculate(string st) {
        stack<int> s;
        stack<char> ope;
        st.push_back('+');
        long res=0;
        long curnum = 0;
        bool sign = true;
        for(auto& cur:st){
            if(cur==' ') continue;
            if(cur=='('){
                s.push(res);
                if(sign) ope.push('+');
                else ope.push('-');
                res = 0;
                curnum = 0;
                sign = true;
            }
            if(cur=='+'||cur=='-'){
                res = sign? res+curnum: res-curnum;
                curnum = 0;
```

```
                sign= cur=='+';

            }
            if(isdigit(cur)){
                curnum = curnum * 10 + cur - '0';
            }
            if(cur==')'){
                res = sign? res+curnum: res-curnum;
                curnum = res;
                res = s.top();
                sign= ope.top()=='+';
                ope.pop(); s.pop();
            }

        }
        return res;

    }
};
```

274. H指数（中等）

1. 题目描述

   给定一位研究者论文被引用次数的数组（被引用次数是非负整数）。编写一个方法，计算出研究者的 h 指数。

   h 指数的定义：h 代表“高引用次数”（high citations），一名科研人员的 h 指数是指他（她）的 （N 篇论文中）至多有 h 篇论文分别被引用了至少 h 次。（其余的 N - h 篇论文每篇被引用次数 不超过 h 次。）

   例如：某人的 h 指数是 20，这表示他已发表的论文中，每篇被引用了至少 20 次的论文总共有 20 篇。

   ```
   示例：
   输入：citations = [3,0,6,1,5]
   输出：3
   解释：给定数组表示研究者总共有 5 篇论文，每篇论文相应的被引用了 3, 0, 6, 1, 5 次。
        由于研究者有 3 篇论文每篇 至少 被引用了 3 次，其余两篇论文每篇被引用 不多于 3 次，所以她的 h 指数是 3。
   ```

   提示：如果 h 有多种可能的值，h 指数是其中最大的那个。

2. 简单实现

```
class Solution {
public:
    int hIndex(vector<int>& citations) {
        sort(citations.begin(), citations.end());//排序
        int n = citations.size();
        int idx = 0;
        while(idx < n){
            if(citations[idx] >= n-idx)//要求的条件
                return n-idx;
            idx++;
```

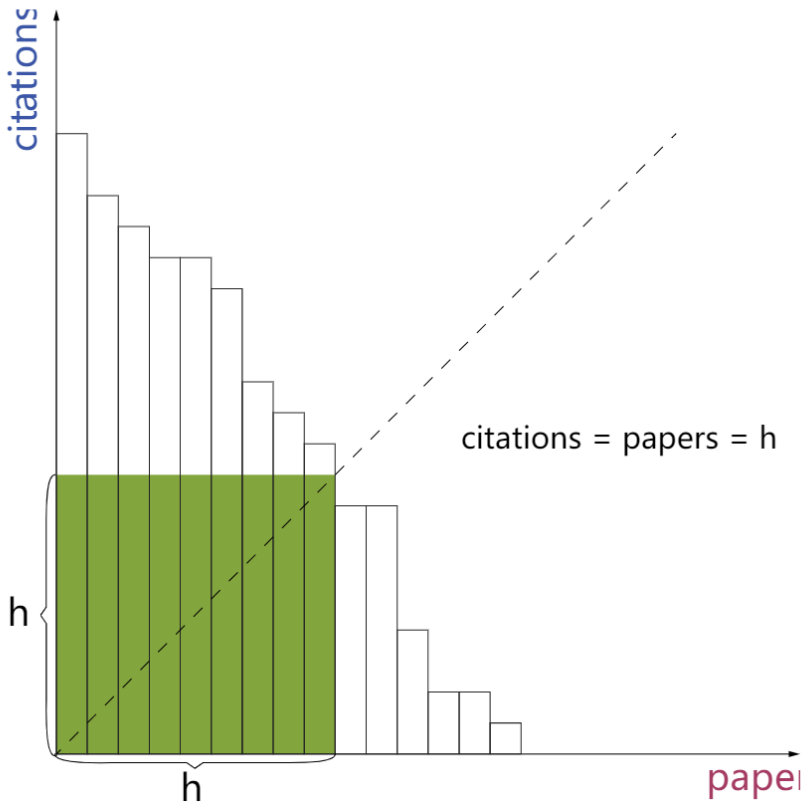
```
            }
            return 0;
        }
    };
```

3. 计数法

想要O(n)时间复杂度，需要接用O(n)空间计数

官方题解对于上一个解法的解释：

我们想象一个直方图，其中 x 轴表示文章，y 轴表示每篇文章的引用次数。如果将这些文章按照引用次数*降序*排序并在直方图上进行表示，那么直方图上的最大的正方形的边长 h 就是我们所要求的 h。



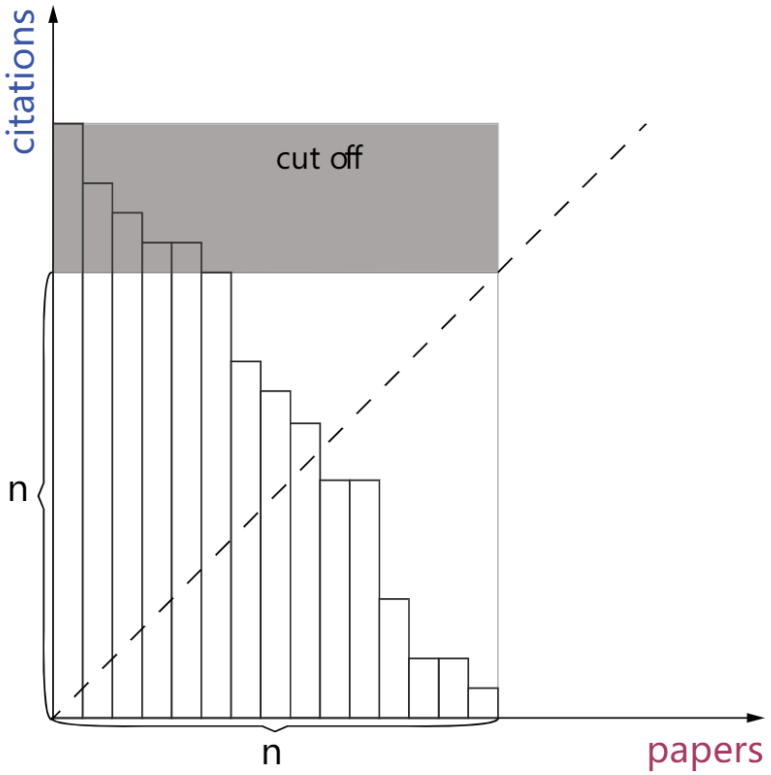
本解法：

方法一中，我们通过降序排序得到了 h 指数，然而，所有基于比较的排序算法，例如堆排序，合并排序和快速排序，都存在时间复杂度下界 $O(n \log n)$。要得到时间复杂度更低的算法. 可以考虑最常用的不基于比较的排序，计数排序。

然而，论文的引用次数可能会非常多，这个数值很可能会超过论文的总数 n，因此使用计数排序是非常不合算的（会超出空间限制）。在这道题中，我们可以通过一个不难发现的结论来让计数排序变得有用，即：

> 如果一篇文章的引用次数超过论文的总数 n，那么将它的引用次数降低为 n 也不会改变 h 指数的值。

由于 h 指数一定小于等于 n，因此这样做是正确的。在直方图中，将所有超过 y 轴值大于 n 的变为 n 等价于去掉 $y > n$ 的整个区域。



从直方图中可以更明显地看出结论的正确性，将 $y > n$ 的区域去除，并不会影响到最大的正方形，也就不会影响到 h 指数。

我们用一个例子来说明如何使用计数排序得到 h 指数。首先，引用次数如下所示：

$$\text{citations} = [1, 3, 2, 3, 100]$$

将所有大于 $n = 5$ 的引用次数变为 n，得到：

$$\text{citations} = [1, 3, 2, 3, 5]$$

计数排序得到的结果如下：

k	0	1	2	3	4	5
count	0	1	1	2	0	1
s_k	5	5	4	3	1	1

其中 s_k 表示至少有 k 次引用的论文数量，在表中即为在它之后的列（包括本身）的 count 一行的和。根据定义，最大的满足 $k \leq s_k$ 的 k 即为所求的 h。在表中，这个 k 为 3，因此 h 指数为 3。

Java

```java
public class Solution {
    public int hIndex(int[] citations) {
        int n = citations.length;
        int[] papers = new int[n + 1];
        // 计数
        for (int c: citations)
            papers[Math.min(n, c)]++;
        // 找出最大的 k
        int k = n;
        for (int s = papers[n]; k > s; s += papers[k])
            k--;
        return k;
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$。在计数时，我们仅需要遍历 citations 数组一次，因此时间复杂度为 $O(n)$。在找出最大的 k 时，我们最多需要遍历计数的数组一次，而计数的数组的长度为 $O(n)$，因此这一步的时间复杂度为 $O(n)$，即总的时间复杂度为 $O(n)$。
- 空间复杂度：$O(n)$。我们需要使用 $O(n)$ 的空间来存放计数的结果。

思考

> 可能会出现多个不同的 h 指数吗?

答案是 **否**。从直方图中可以看出，由于 y 轴已经降序排序，因此直线 $y = x$ 有且仅有穿过直方图一次。同时，也可以直接通过 h 指数的定义证明出 h 指数的唯一性。

417. 太平洋大西洋水流问题（中等）

1. 题目描述

给定一个 m x n 的非负整数矩阵来表示一片大陆上各个单元格的高度。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。

规定水流只能按照上、下、左、右四个方向流动，且只能从高到低或者在同等高度上流动。

请找出那些水流既可以流动到“太平洋”，又能流动到“大西洋”的陆地单元的坐标。

提示：

- 输出坐标的顺序不重要
- m 和 n 都小于150

```
示例:
给定下面的 5x5 矩阵:
  太平洋 ~   ~   ~   ~   ~
       ~  1   2   2   3  (5) *
       ~  3   2   3  (4) (4) *
       ~  2   4  (5)  3   1  *
       ~ (6) (7)  1   4   5  *
       ~ (5)  1   1   2   4  *
          *   *   *   *   * 大西洋

返回:[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (上图中带括号的单元).
```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> dirs = {{0,1}, {0,-1}, {-1,0}, {1,0}};
    vector<vector<int>> pacificAtlantic(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if(m <= 0) return {};
        int n = matrix[0].size();
        vector<vector<vector<bool>>> flag(m, vector<vector<bool>>(n, vector<bool>
(2, 0)));
        queue<pair<int, int>> q;
        //太平洋
        for(int j = 0; j < n; j++){
            q.push(make_pair(0, j));
            flag[0][j][0] = true;
        }
        for(int i = 1; i < m; i++){
            q.push(make_pair(i, 0));
            flag[i][0][0] = true;
        }
        while(!q.empty()){
            int x = q.front().first;
            int y = q.front().second;
            q.pop();
            for(int i = 0; i < 4; i++){
                int xx = x + dirs[i][0];
                int yy = y + dirs[i][1];
                if(xx >= 0 && xx < m && yy >= 0 && yy < n && matrix[xx][yy] >=
matrix[x][y] && flag[xx][yy][0] == false){
                    flag[xx][yy][0] = true;
                    q.push(make_pair(xx, yy));
                }
            }
        }
        //大西洋
        for(int j = 0; j < n; j++){
            q.push(make_pair(m-1, j));
            flag[m-1][j][1] = true;
        }
```

```
        for(int i = 0; i < m-1; i++){
            q.push(make_pair(i, n-1));
            flag[i][n-1][1] = true;
        }
        vector<vector<int>> ans;
        while(!q.empty()){
            int x = q.front().first;
            int y = q.front().second;
            q.pop();
            if(flag[x][y][0] == true && flag[x][y][1] == true)
                ans.push_back({x, y});
            for(int i = 0; i < 4; i++){
                int xx = x + dirs[i][0];
                int yy = y + dirs[i][1];
                if(xx >= 0 && xx < m && yy >= 0 && yy < n && matrix[xx][yy] >=
matrix[x][y] && flag[xx][yy][1] == false){
                    flag[xx][yy][1] = true;
                    q.push(make_pair(xx, yy));
                }
            }
        }
        return ans;
    }
};
```

169. 多数元素（简单）

1. 题目描述

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

```
示例 1:
输入: [3,2,3]
输出: 3

示例 2:
输入: [2,2,1,1,1,2,2]
输出: 2
```

2. 简单实现

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int ans = nums[0];
        int cnt = 1;
        for(int i = 1; i < nums.size(); i++){
            if(ans == nums[i])
                cnt++;
            else if(cnt > 0)
```

```
                cnt--;
            else{
                ans = nums[i];
                cnt = 1;
            }
        }
        return ans;
    }
};
```

543. 二叉树的直径（简单）

1. 题目描述

   给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

   示例：

```
给定二叉树
      1
     / \
    2   3
   / \
  4   5
返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。
```

   注意：两结点之间的路径长度是以它们之间边的数目表示。

2. 简单实现

```
class Solution {
public:
    int ans = 0;
    int helper(TreeNode* root){
        if(!root) return 0;
        int l = helper(root->left);
        int r = helper(root->right);
        ans = max(ans, l+r);
        return max(l, r) + 1;
    }
    int diameterOfBinaryTree(TreeNode* root) {
        helper(root);
        return ans;
    }
};
```

247. 中心对称数II（中等）

要会员

392. 判断子序列（简单）

1. 题目描述

   给定字符串 s 和 t ，判断 s 是否为 t 的子序列。

   你可以认为 s 和 t 中仅包含英文小写字母。字符串 t 可能会很长（长度 ~= 500,000），而 s 是个短字符串（长度 <=100）。

   字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

   ```
   示例 1:
   s = "abc", t = "ahbgdc"
   返回 true.

   示例 2:
   s = "axc", t = "ahbgdc"
   返回 false.
   ```

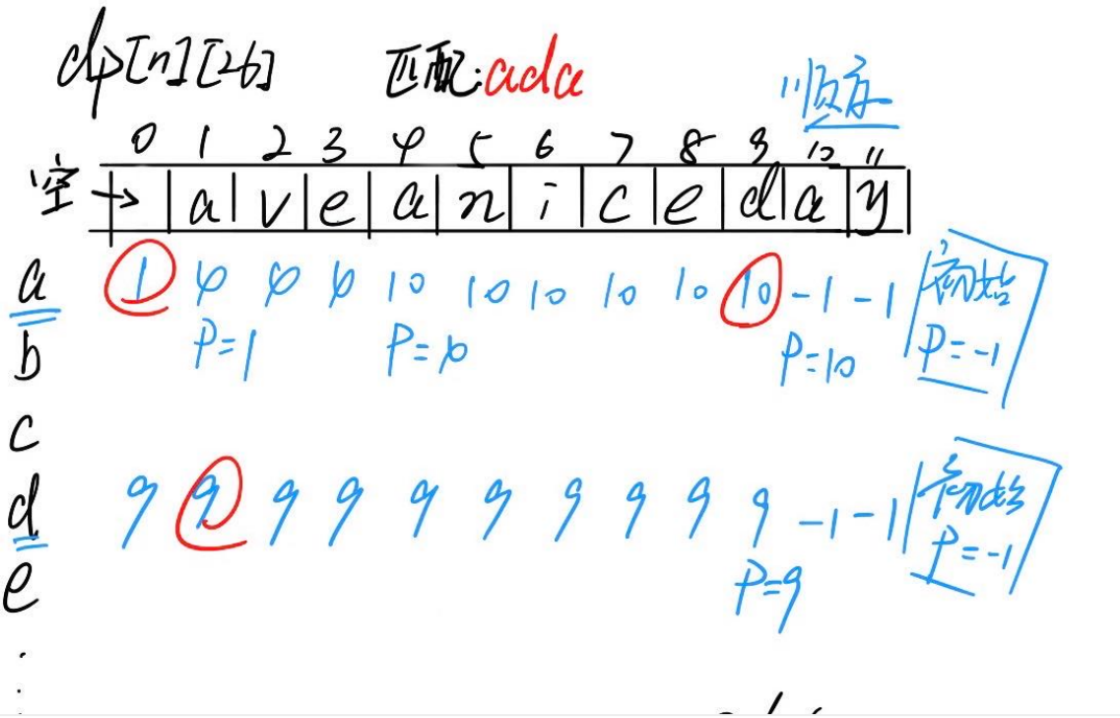
   后续挑战 : 如果有大量输入的 S，称作S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码?

2. 简单实现

   ```cpp
   class Solution {
   public:
       bool isSubsequence(string s, string t) {
           int idx = 0;
           int len1 = s.size();
           int len2 = t.size();
           if(len1 == 0) return true;
           for(int i = 0; i < len2 && idx < len1; i++){
               if(t[i] == s[idx])
                   idx++;
           }
           return idx == len1;
       }
   };
   ```

匹配一串字符需要 $$O(n)$$ ， n 为 **t** 的长度。如果有大量输入的 **S**，称作 **S1** , **S2** , ... , **Sk** 其中 k >= 10 亿，你需要依次检查它们是否为 **T** 的子序列，这时候处理每一个子串都需要扫描一遍 **T** 是很费时的。

在这种情况下，我们需要在匹配前对 **T** 做预处理，利用一个二维数组记录每个位置的下一个要匹配的字符的位置，这里的字符是 `'a' ~ 'z'` ，所以这个数组的大小是 `dp[n][26]` ，n 为 **T** 的长度。那么每处理一个子串只需要扫描一遍 **Si** 即可，因为在数组的帮助下我们对 **T** 是“跳跃”扫描的。比如下面匹配 "ada" 的例子，只需要“跳跃”三次。



64. 最小路径和（中等）

1. 题目描述

给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

```
示例:
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 7
```

解释: 因为路径 1→3→1→1→1 的总和最小。

2. 简单实现

第一反应是DFS，仔细一想果然可以dp，犯懒直接在原地修改了，也可以一维动规

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        if(m <= 0) return 0;
        int n = grid[0].size();
        for(int j = 1; j < n; j++)
            grid[0][j] += grid[0][j-1];
        for(int i = 1; i < m; i++){
            grid[i][0] += grid[i-1][0];
            for(int j = 1; j < n; j++){
                grid[i][j] += min(grid[i-1][j], grid[i][j-1]);
            }
        }
        return grid[m-1][n-1];
    }
};
```

351. 安卓系统手势解锁（中等）

要会员

463. 岛屿的周长（简单）

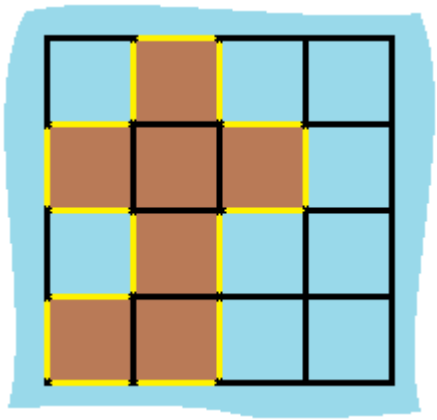
1. 题目描述

给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地 0 表示水域。

网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（或者说，一个或多个表示陆地的格子相连组成的岛屿）。

岛屿中没有“湖”（“湖” 指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。网格为长方形，且宽度和高度均不超过 100 。计算这个岛屿的周长。

```
示例 ：
输入:
[[0,1,0,0],
 [1,1,1,0],
 [0,1,0,0],
 [1,1,0,0]]
输出: 16
解释: 它的周长是下面图片中的 16 个黄色的边:
```



2. 简单实现

   遍历数边

```
class Solution {
public:
    vector<vector<int>> dirs = {{0,1}, {0,-1}, {-1,0}, {1,0}};
    int islandPerimeter(vector<vector<int>>& grid) {
        int ans = 0;
        int m = grid.size();
        if(m == 0) return ans;
        int n = grid[0].size();
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(grid[i][j] == 1){
                    for(int k = 0; k < 4; k++){
                        int xx = i + dirs[k][0];
                        int yy = j + dirs[k][1];
                        if(xx >= 0 && xx < m && yy >= 0 && yy < n){
                            if(grid[xx][yy] == 0)
                                ans++;
                        }
                        else
                            ans++;
                    }
                }
            }
        }
        return ans;
    }
};
```

737. 句子相似性（中等）

要会员

69. x的平方根（简单）

1. 题目描述

实现 int sqrt(int x) 函数，计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

```
示例 1:
输入: 4
输出: 2

示例 2:
输入: 8
输出: 2
说明: 8 的平方根是 2.82842...,  由于返回类型是整数，小数部分将被舍去。
```

2. 简单实现

```
class Solution {
public:
    int mySqrt(int x) {
        if(x <= 1) return x;
        long l = 1, r = x;//long防止越界
        while(l <= r){
            long mid = l + (r - l) / 2;
            long cur = mid * mid;
            if(cur == x)
                return mid;
            else if(cur < x)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return r;
    }
};
```

3. 简易计算器法

首先，了解一下袖珍计算器算法。

通常，袖珍计算器通过对数表或其他方式计算指数函数和自然对数。那么考虑将求平方根的运算转换为指数运算和对数运算：

$$\sqrt{x} = e^{\frac{1}{2}\log x}$$

此方法中使用到了对数表与其他策略，省略了一部分计算。但实际上对数函数和指数函数本身就是这样计算的。

```java
class Solution {
  public int mySqrt(int x) {
    if (x < 2) return x;
    int left = (int)Math.pow(Math.E, 0.5 * Math.log(x));
    int right = left + 1;
    return (long)right * right > x ? left : right;
  }
}
```

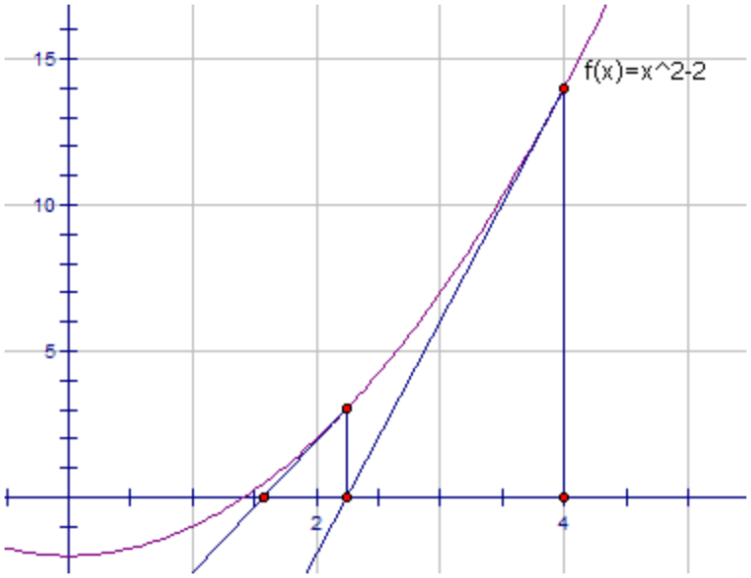
4. 牛顿法

这题的解法用暴力解法是非常简单的。主要的麻烦在于如何解的更好，答案就是用牛顿迭代法。

下面这种方法可以很有效地求出根号 a 的近似值：首先随便猜一个近似值 x，然后不断令 x 等于 x 和 a/x 的平均数，迭代个六七次后 x 的值就已经相当精确了。

例如，我想求根号 2 等于多少。假如我猜测的结果为 4，虽然错的离谱，但你可以看到使用牛顿迭代法后这个值很快就趋近于根号 2 了：

> (4 + 2/ 4) / 2 = 2.25
>
> (2.25 + 2/ 2.25) / 2 = 1.56944..
>
> (1.56944..+ 2/1.56944..) / 2 = 1.42189..
>
> (1.42189..+ 2/1.42189..) / 2 = 1.41423..
>
> ....



这种算法的原理很简单，我们仅仅是不断用 $(x, f(x))$ 的切线来逼近方程 $x^2 - a = 0$ 的根。根号 a 实际上就是 $x^2 - a = 0$ 的一个正实根，这个函数的导数是 $2x$。也就是说，函数上任一点 $(x, f(x))$ 处的切线斜率是 $2x$。那么，$x - f(x)/(2x)$ 就是一个比 x 更接近的近似值。代入 $f(x) = x^2 - a$ 得到 $x - (x^2 - a)/(2x)$，也就是 $(x + a/x)/2$。

同样的方法可以用在其它的近似值计算中。Quake III 的源码中有一段非常牛B的开方取倒函数。

知道方程实现就非常简单了。

150. 逆波兰表达式求值（中等）

1. 题目描述

根据逆波兰表示法，求表达式的值。

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

说明：

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

```
示例 1:
输入: ["2", "1", "+", "3", "*"]
输出: 9
解释: ((2 + 1) * 3) = 9

示例 2:
输入: ["4", "13", "5", "/", "+"]
输出: 6
解释: (4 + (13 / 5)) = 6

示例 3:
输入: ["10", "6", "9", "3", "+", "-11", "", "/", "", "17", "+", "5", "+"]
输出: 22
```

```
解释:
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

2. 简单实现

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> s;
        for(int i = 0; i < tokens.size(); i++){
            if(tokens[i] == "+"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a+b);
            }
            else if(tokens[i] == "-"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a-b);
            }
            else if(tokens[i] == "*"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a*b);
            }
            else if(tokens[i] == "/"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a/b);
            }
            else{
                s.push(stoi(tokens[i]));
            }
        }
        return s.top();
    }
};
```

334. 递增的三元子序列（中等）

1. 题目描述

   给定一个未排序的数组，判断这个数组中是否存在长度为 3 的递增子序列。

   数学表达式: 如果存在这样的 i, j, k, 且满足 0 ≤ i < j < k ≤ n-1，使得 arr[i] < arr[j] < arr[k]，返回 true ; 否则返回 false 。 说明: 要求算法的时间复杂度为 O(n)，空间复杂度为 O(1) 。

```
示例 1:
输入: [1,2,3,4,5]
输出: true

示例 2:
输入: [5,4,3,2,1]
输出: false
```

2. 简单实现

   之前做过的找最长上升子序列长度的简易版

```
class Solution {
public:
    bool increasingTriplet(vector<int>& nums) {
        int size = nums.size();
        if(size < 3) return false;
        vector<int> v = {nums[0]};
        for(int i = 1; i < size; i++){
            if(nums[i] > v.back()){
                if(v.size() == 2)
                    return true;
                v.push_back(nums[i]);
            }
            else{
                int idx = v.size() - 1;
                while(idx >= 0 && v[idx] >= nums[i])
                    idx--;
                v[idx+1] = nums[i];
            }
        }
        return false;
    }
};
```

   代码简化

```
class Solution {
public:
    const long INF = 1e10;
    bool increasingTriplet(vector<int>& nums) {
        long n1 = INF;
        long n2 = INF;
```

```
        long n3 = INF;
        for (auto x : nums) {
            if (x < n1) {
                n1 = x;
            } else if (x != n1 && x < n2) {
                n2 = x;
            } else if (x != n1 && x != n2 && x < n3) {
                n3 = x;
            }
            if (n1 < n2 && n2 < n3 && n3 < INF) {
                return true;
            }
        }
        return false;
    }
};
```

846. 一手顺子（中等）

1. 题目描述

   爱丽丝有一手（hand）由整数数组给定的牌。 现在她想把牌重新排列成组，使得每个组的大小都是 W，且由 W 张连续的牌组成。

   如果她可以完成分组就返回 true，否则返回 false。

```
示例 1：
输入：hand = [1,2,3,6,2,3,4,7,8], W = 3
输出：true
解释：爱丽丝的手牌可以被重新排列为 [1,2,3]，[2,3,4]，[6,7,8]。

示例 2：
输入：hand = [1,2,3,4,5], W = 4
输出：false
解释：爱丽丝的手牌无法被重新排列成几个大小为 4 的组。
```

   提示：

   - 1 <= hand.length <= 10000
   - 0 <= hand[i] <= 10^9
   - 1 <= W <= hand.length

2. 简单实现

   用map统计各个数字出现的次数，每次从小到大找W个连续的，看能否找到

```
class Solution {
public:
    bool isNStraightHand(vector<int>& hand, int W) {
        int size = hand.size();
        if(size % W != 0) return false;//不能整除，肯定不行
        map<int, int> cnt;
```

```
        for(int i = 0; i < size; i++)
            cnt[hand[i]]++;
        auto l = cnt.begin();//当前分组中连续数字的起点
        auto r = l;
        while(size > 0){
            l->second--;//第一个数字放入数组
            size--;
            r++;
            for(int i = 1; i < w; i++){
                if(r->first == l->first + i && r->second > 0){//连续
                    r->second--;
                    r++;
                    size--;
                }
                else//断掉了
                    return false;
            }
            if(size == 0) return true;
            while(l->second == 0)//寻找新的起点
                l++;
            r = l;
        }
        return true;
    }
};
```

94. 二叉树的中序遍历（中等）

1. 题目描述

   给定一个二叉树，返回它的中序 遍历。

   示例:

   输入: [1,null,2,3] 1 \ 2 / 3

   输出: [1,3,2] 进阶: 递归算法很简单，你可以通过迭代算法完成吗?

2. 简单实现

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        if(!root) return vector<int>();
        stack<TreeNode*> s;
        vector<int> ans;
        while(root){
            s.push(root);
            root = root->left;
        }
        while(!s.empty()){
            TreeNode* cur = s.top();
            s.pop();
            ans.push_back(cur->val);
```

```
            if(cur->right){
                cur = cur->right;
                while(cur){
                    s.push(cur);
                    cur = cur->left;
                }
            }
        }
        return ans;
    }
};
```

341. 扁平化嵌套列表迭代器（中等）

1. 题目描述

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

```
示例 1:
输入: [[1,1],2,[1,1]]
输出: [1,1,2,1,1]
解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,1,2,1,1]。

示例 2:
输入: [1,[4,[6]]]
输出: [1,4,6]
解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,4,6]。
```

2. 简单实现

```
/**
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * class NestedInteger {
 *   public:
 *     // Return true if this NestedInteger holds a single integer, rather than a
nested list.
 *     bool isInteger() const;
 *
 *     // Return the single integer that this NestedInteger holds, if it holds a
single integer
 *     // The result is undefined if this NestedInteger holds a nested list
 *     int getInteger() const;
 *
 *     // Return the nested list that this NestedInteger holds, if it holds a
nested list
 *     // The result is undefined if this NestedInteger holds a single integer
 *     const vector<NestedInteger> &getList() const;
 * };
 */
```

```
class NestedIterator {
public:
    vector<int> data;
    vector<int>::iterator it;
    void flat(const NestedInteger &n){
        if(n.isInteger())
            data.push_back(n.getInteger());
        else{
            const vector<NestedInteger>& list = n.getList();
            for(int i = 0; i < list.size(); i++)
                flat(list[i]);
        }
    }
    NestedIterator(vector<NestedInteger> &nestedList) {
        for(int i = 0; i < nestedList.size(); i++)
                flat(nestedList[i]);
        it = data.begin();
    }

    int next() {
        return *it++;
    }

    bool hasNext() {
        return it != data.end();
    }
};

/**
 * Your NestedIterator object will be instantiated and called as such:
 * NestedIterator i(nestedList);
 * while (i.hasNext()) cout << i.next();
 */
```

207. 课程表（中等）

1. 题目描述

   你这个学期必须选修 numCourse 门课程，记为 0 到 numCourse-1 。

   在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：[0,1]

   给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习?

```
示例 1:
输入: 2, [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。

示例 2:
输入: 2, [[1,0],[0,1]]
输出: false
解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。
```

提示：

- 输入的先决条件是由 边缘列表 表示的图形，而不是 邻接矩阵 。详情请参见图的表示法。
- 你可以假定输入的先决条件中没有重复的边。
- 1 <= numCourses <= 10^5

2. 简单实现

```cpp
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        unordered_map<int, vector<int>> m;//构造邻接表法表示的图
        vector<int> incnt = vector<int>(numCourses, 0);//记录每个节点的入度
        //初始化
        for(int i = 0; i < prerequisites.size(); i++){
            incnt[prerequisites[i][0]]++;
            m[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }
        //拓扑排序
        queue<int> q;
        int done = 0;//记录可以修的课程数
        for(int i = 0; i < numCourses; i++)
            if(incnt[i] == 0){//0入度的进队列
                q.push(i);
                done++;
            }
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                int cur = q.front();
                q.pop();
                for(int j = 0; j < m[cur].size(); j++){
                    if(--incnt[m[cur][j]] == 0){//所有下一个节点入度-1后为0则入队列
                        q.push(m[cur][j]);
                        done++;
                    }
                }
            }
        }
        if(done == numCourses)
            return true;
        else
```

```
                return false;
        }
};
```

403. 青蛙过河（困难）

1. 题目描述

一只青蛙想要过河。 假定河流被等分为 x 个单元格，并且在每一个单元格内都有可能放有一石子（也有可能没有）。 青蛙可以跳上石头，但是不可以跳入水中。

给定石子的位置列表（用单元格序号升序表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一个石子上）。 开始时， 青蛙默认已站在第一个石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格1跳至单元格2）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

请注意：

- 石子的数量 ≥ 2 且 < 1100；
- 每一个石子的位置序号都是一个非负整数，且其 < 231；
- 第一个石子的位置永远是0。

```
示例 1:
[0,1,3,5,6,8,12,17]
总共有8个石子。
第一个石子处于序号为0的单元格的位置，第二个石子处于序号为1的单元格的位置，
第三个石子在序号为3的单元格的位置， 以此定义整个数组...
最后一个石子处于序号为17的单元格的位置。
返回 true。即青蛙可以成功过河，按照如下方案跳跃：
跳1个单位到第2块石子，然后跳2个单位到第3块石子，接着
跳2个单位到第4块石子，然后跳3个单位到第6块石子，
跳4个单位到第7块石子，最后，跳5个单位到第8个石子（即最后一块石子）。

示例 2:
[0,1,2,3,4,8,9,11]
返回 false。青蛙没有办法过河。
这是因为第5和第6个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。
```

2. 简单实现

dp[i]表示如果能跳到第i个石子，它的上一步可能跳的所有步数的可能，则转移方程为：

`dp[i] = {stones[i]-stones[j] if (stones[i]-stones[j] 或其+-1 在dp[j]中)}`

```
class Solution {
public:
    bool canCross(vector<int>& stones) {
        int size = stones.size();
        if(stones[1] != 1) return false;
        vector<set<int>> dp(size);//dp[i]表示如果能跳到第i个石子，它的上一步可能跳的所有步
数的可能
        dp[1].insert(1);
```

```
        for(int i = 2; i < size; i++){
            //寻找j的起始点，规则在于，不可能从小于3的石子跳到6
            int j = lower_bound(stones.begin(), stones.end(), stones[i] / 2) -
stones.begin();
            for(; j < i; j++){
                if(dp[j].size() > 0){
                    int aim = stones[i] - stones[j];
                    auto it = dp[j].lower_bound(aim-1);
                    if(it != dp[j].end() && (*it == aim-1 || *it == aim || *it ==
aim+1))
                        dp[i].insert(aim);
                }
            }
        }
        return dp[size-1].size() > 0;
    }
};
```

3. 最优解法

```
class Solution {
public:
    bool canCross(vector<int>& stones) {
        int n = stones.size();
        if(*stones.rbegin() > (n - 1) * n / 2)
            return false;
        unordered_set<int> st(stones.begin(), stones.end());
        return dfs(st, 0, 0, *stones.rbegin());
    }
    //上一步跳了step跳到了pos
    bool dfs(unordered_set<int>& st, int pos, int step, int target){
        if(pos == target)
            return true;
        for(int s = step + 1; s > step - 2; s--){
            if(s + pos <= pos)
                continue;
            if(st.count(s + pos) && dfs(st, pos + s, s, target))
                return true;
        }
        return false;
    }
};
```