# 贪心

## 455 分发饼干

class Solution {

public:

    int findContentChildren(vector<int>& g, vector<int>& s) {

        sort(g.begin(), g.end());

        sort(s.begin(), s.end());

        int id\_g = 0, id\_s = 0;

        while(id\_g < g.size() && id\_s < s.size()) {

            if(s[id\_s] >= g[id\_g]) {

                ++id\_s;

                ++id\_g;

            }

            else {

                ++id\_s;

            }

        }

        return id\_g;

    }

};

## 135 分发糖果

class Solution {

public:

    int candy(vector<int>& ratings) {

        int n = ratings.size();

        vector<int> candy(n, 1);

        for(int i=1; i<n; ++i) {

            if(ratings[i] > ratings[i-1]) {

                candy[i] = candy[i-1] + 1;

            }

        }

        for(int i=n-2; i>=0; --i) {

            if(ratings[i] > ratings[i+1]) {

                candy[i] = max(candy[i], candy[i+1] + 1);

            }

        }

        return accumulate(candy.begin(), candy.end(), 0);

    }

};

## 605 种花问题

这个题虽然简单，但是如果思考放花的条件，还是比较麻烦的。这时候想起了在验证二叉搜索树里面的一个小技巧。我们可以考虑问题的反面，什么时候不能放花呢？排除掉不能放的情况，当然就能放啦。代码如下：

class Solution {

public:

    bool canPlaceFlowers(vector<int>& flowerbed, int n) {

        int num = 0;

        for(int i=0; i<flowerbed.size(); ++i) {

            if(flowerbed[i] == 0) {

                if(i-1 >=0 && flowerbed[i-1] == 1) continue;

                if(i+1 < flowerbed.size() && flowerbed[i+1] == 1) continue;

                flowerbed[i] = 1;

                ++num;

            }

        }

        return num >= n;

    }

};

## 763 划分字母区间

这道题类似跳跃游戏，也是双指针类型的贪心问题

class Solution {

public:

    vector<int> partitionLabels(string S) {

        vector<int> res;

        unordered\_map<char, int> char\_to\_last;

        int n = S.size();

        for(int i=0; i<n; ++i) {

            char\_to\_last[S[i]] = i;

        }

        int start = 0, end = 0;

        for(int i=0; i<n; ++i) {

            end = max(end, char\_to\_last[S[i]]);

            if( i == end ) {

                res.push\_back(end-start+1);

                start = end + 1;

                end = end + 1;

            }

        }

        return res;

    }

};

## 406 根据身高重建队列

class Solution {

public:

    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {

        int n = people.size();

        sort(people.begin(), people.end(), [](const vector<int>& p1, const vector<int>& p2) {

            return p1[0] > p2[0] || (p1[0] == p2[0] && p1[1] < p2[1]);

        });

        vector<vector<int>> res;

        for(int i=0; i<n; ++i) {

            res.insert(res.begin() + people[i][1], people[i]);

        }

        return res;

    }

};

这个题理解起来还是有难度的。首先要明确插入顺序为先高后矮，因为矮个子的插入，不会影响前面高个子的属性。

用deque转一下，会更好

class Solution {

public:

    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {

        int n = people.size();

        sort(people.begin(), people.end(), [](const vector<int>& p1, const vector<int>& p2) {

            return p1[0] > p2[0] || (p1[0] == p2[0] && p1[1] < p2[1]);

        });

        deque<vector<int>> dq;

        for(int i=0; i<n; ++i) {

            dq.insert(dq.begin() + people[i][1], people[i]);

        }

        return vector<vector<int>>(dq.begin(), dq.end());

    }

};

## 665 非递减数列

一定要学会计算机的思考方式：寻找在当前状态下的局部最优处理

class Solution {

public:

    bool checkPossibility(vector<int>& nums) {

        int cnt = 0;

        for(int i=1; i<nums.size() && cnt < 2; ++i) {

            if(nums[i] >= nums[i-1]) continue;

            ++cnt;

            if(i-2 >= 0 && nums[i-2] > nums[i]) {

                nums[i] = nums[i-1];

            }

            else {

                nums[i-1] = nums[i];

            }

        }

        return cnt <= 1;

    }

};

# 双指针

## 88 合并两个有序数组

class Solution {

public:

    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

        int i = m-1, j=n-1, k=m+n-1;

        while(i>=0 && j>=0) nums1[k--] = nums1[i] > nums2[j] ? nums1[i--] : nums2[j--];

        while(i>=0) nums1[k--] = nums1[i--];

        while(j>=0) nums1[k--] = nums2[j--];

    }

};

## 633 平方数之和

class Solution {

public:

    bool judgeSquareSum(int c) {

        long long i = 0, j = sqrt(c);

        while(i <= j) {

            long long pow\_sum = i \* i + j \* j;

            if(pow\_sum == c) return true;

            else if(pow\_sum > c) --j;

            else ++i;

        }

        return false;

    }

};

类似两数之和的解法

## 680 验证回文字符串 Ⅱ

class Solution {

public:

    bool validPalindrome(string s) {

        int i=0, j=s.size()-1;

        while(i<j) {

            if(s[i] != s[j]) {

                return isValid(s, i+1, j) || isValid(s, i, j-1);

            }

            else {

                ++i; --j;

            }

        }

        return true;

    }

    bool isValid(const string& s, int i, int j) {

        while(i<j) {

            if(s[i++] != s[j--]) return false;

        }

        return true;

    }

};

双指针验证回文的一个变种版本。

## 524通过删除字母匹配到字典里最长单词

这道题正好可以用到二分查找来高效判定子序列

[https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247484479&idx=1&sn=31a3fc4aebab315e01ea510e482b186a&chksm=9bd7fa37aca0732103ca82e6f2cc23f475cf771696958456fc17d7662abb6b0879e8dfbaf7a1&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzAxODQxMDM0Mw==&mid=2247484479&idx=1&sn=31a3fc4aebab315e01ea510e482b186a&chksm=9bd7fa37aca0732103ca82e6f2cc23f475cf771696958456fc17d7662abb6b0879e8dfbaf7a1&scene=21" \l "wechat_redirect)

这道题其实就是子序列

class Solution {

    bool isSubsequence(string s, string target){

        int i = 0, j = 0;

        int m = s.size(), n = target.size();

        while(i<m && j<n) {

            if(s[i] == target[j]) ++j;

            ++i;

        }

        return j==n;

    }

public:

    string findLongestWord(string s, vector<string>& d) {

        string res = "";

        for(const auto& target : d) {

            if(target.length() < res.length()) continue;

            if(target.length() == res.length() && res <= target) continue;

            if(isSubsequence(s, target)) {

                res = target;

            }

        }

        return res;

    }

};

二分查找版本：

class Solution {

    int left\_bound(vector<int>& nums, int tgt) {

        int lo = 0, hi = nums.size(), mid;

        while(lo < hi) {

            mid = lo + (hi-lo) / 2;

            if(nums[mid] < tgt) lo = mid + 1;

            else hi = mid;

        }

        return lo;

    }

    bool isSubsequence(string s, string target, vector<vector<int>>& char\_to\_index){

        int j = 0;

        for(int i=0; i<target.length(); ++i) {

            char c = target[i];

            if(char\_to\_index[c-'a'].empty()) return false;

            int pos = left\_bound(char\_to\_index[c-'a'], j);

            if(pos == char\_to\_index[c-'a'].size()) return false;

            j = char\_to\_index[c-'a'][pos] + 1;

        }

        return true;

    }

public:

    string findLongestWord(string s, vector<string>& d) {

        string res = "";

        // 预处理s

        vector<vector<int>> char\_to\_index(26);

        for(int i=0; i<s.length(); ++i) {

            char c = s[i];

            char\_to\_index[c-'a'].push\_back(i);

        }

        // 循环

        for(const auto& target : d) {

            if(res.length() > target.length()) continue;

            if(res.length() == target.length() && res <= target) continue;

            // 判断每个target是不是子序列，如果是更新结果

            if(isSubsequence(s, target, char\_to\_index)) res = target;

        }

        return res;

    }

};

挺考验代码能力的，需要慢慢写

从leetcode实际来看，确实是节省了时间。

## 340 至多包含 K 个不同字符的最长子串

典型的滑动窗口问题，套用模板即可

class Solution {

public:

    int lengthOfLongestSubstringKDistinct(string s, int k) {

        int left = 0, right = 0;

        unordered\_map<char, int> window;

        int res = 0;

        while(right < s.length()) {

            char c = s[right];

            ++right;

            ++window[c];

            while(window.size() > k) {

                char d = s[left];

                ++left;

                --window[d];

                if(window[d] == 0) {

                    window.erase(d);

                }

            }

            res = max(res, right - left);

        }

        return res;

    }

};

# 排序

## 347 前k个高频元素

桶排序解法：

class Solution {

public:

    vector<int> topKFrequent(vector<int>& nums, int k) {

        unordered\_map<int, int> counter;

        int max\_count;

        for(auto num : nums) {

            ++counter[num];

            max\_count = max(max\_count, counter[num]);

        }

        vector<vector<int>> buckets(max\_count + 1);

        for(const auto& p : counter) {

            buckets[p.second].push\_back(p.first);

        }

        vector<int> ans;

        for(int i=max\_count; i>=0 && ans.size() < k; --i) {

            for(auto num : buckets[i]) {

                ans.push\_back(num);

                if(ans.size() == k) break;

            }

        }

        return ans;

    }

};

优先级队列解法：

与上面差不多，但是使用小顶堆，把太小的从堆顶扔出去，堆里面剩下的就是大的

class Solution {

public:

    static bool comp(const pair<int,int>& a, const pair<int,int>& b) {

        return a.second > b.second;

    }

    vector<int> topKFrequent(vector<int>& nums, int k) {

        unordered\_map<int, int> counter;

        for(auto num : nums) {

            ++counter[num];

        }

        priority\_queue<pair<int, int>, vector<pair<int,int>>, decltype(&comp)> pq(comp);

        for(const auto& p : counter) {

            if(pq.size() == k) {

                if(pq.top().second < p.second) {

                    pq.pop();

                    pq.push(p);

                }

            }

            else {

                pq.emplace(p.first, p.second);

            }

        }

        vector<int> ans;

        while(!pq.empty()) {

            ans.push\_back(pq.top().first);

            pq.pop();

        }

        return ans;

    }

};

注意这里优先级队列的定义方式

## 451 根据字符出现频率排序

这里也可以用桶排序和优先级队列两种解法

优先级队列：

class Solution {

    static bool comp(const pair<char,int>& a, const pair<char,int>& b) {

        return a.second < b.second;

    }

public:

    string frequencySort(string s) {

        unordered\_map<char, int> counter;

        for(auto c : s) ++counter[c];

        priority\_queue<pair<char,int>, vector<pair<char,int>>, decltype(&comp)> pq(comp);

        for(const auto& p : counter) {

            pq.push(p);

        }

        string ans;

        while(!pq.empty()) {

            auto [ch, num] = pq.top();

            pq.pop();

            for(int i=0; i<num; ++i) {

                ans.push\_back(ch);

            }

        }

        return ans;

    }

};

## 75 颜色分类（荷兰国旗问题）

<https://leetcode-cn.com/problems/sort-colors/solution/yan-se-fen-lei-by-leetcode-solution/>

// 单指针两次循环解法，第一次换0，第二次换1

class Solution {

public:

    void sortColors(vector<int>& nums) {

        int n = nums.size();

        int ptr = 0;

        for(int i=0; i<n; ++i) {

            if(nums[i]==0) {

                swap(nums[i], nums[ptr]);

                ++ptr;

            }

        }

        for(int i=ptr; i<n; ++i) {

            if(nums[i]==1) {

                swap(nums[i], nums[ptr]);

                ++ptr;

            }

        }

    }

};

// 双指针，一个找0， 一个找1。只需要一次循环

// 重点是要理解nums[i] == 0的情形，有疑惑去找官网

class Solution {

public:

    void sortColors(vector<int>& nums) {

        int n = nums.size();

        int p0 = 0, p1 = 0;

        for(int i=0; i<n; ++i) {

            if(nums[i] == 1) {

                swap(nums[i], nums[p1]);

                ++p1;

            }

            if(nums[i] == 0) {

                swap(nums[i], nums[p0]);

                if(p0 < p1) swap(nums[p1], nums[i]);

                ++p0; ++p1;

            }

        }

    }

};

// 双指针，一个找0， 一个找2

class Solution {

public:

    void sortColors(vector<int>& nums) {

        int n = nums.size();

        int p0 = 0, p2 = n-1;

        for(int i=0; i<=p2; ++i) {

            while(i<=p2 && nums[i] == 2) {

                swap(nums[p2], nums[i]);

                --p2;

            }

            if(nums[i] == 0) {

                swap(nums[p0], nums[i]);

                ++p0;

            }

        }

    }

};

# 搜索

## 695 岛屿的最大面积

栈

vector<int> dir{-1, 0, 1, 0, -1};

class Solution {

public:

    int maxAreaOfIsland(vector<vector<int>>& grid) {

        int m = grid.size(), n = m ? grid[0].size() : 0;

        int x, y;

        int local\_area = 0, global\_area = 0;

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                if(grid[i][j] == 1) {

                    local\_area = 1;

                    grid[i][j] = 0;

                    stack<pair<int, int>> island;

                    island.push({i, j});

                    while(!island.empty()) {

                        auto [r, c] = island.top();

                        island.pop();

                        for(int i=0; i<4; ++i) {

                            x = r + dir[i];

                            y = c + dir[i+1];

                            if(x>=0 && x<m && y>=0 && y<n && grid[x][y]==1) {

                                grid[x][y] = 0;

                                local\_area++;

                                island.push({x,y});

                            }

                        }

                    }

                    global\_area = max(global\_area, local\_area);

                }

            }

        }

        return global\_area;

    }

};

递归一：

vector<int> dir{-1, 0, 1, 0, -1};

class Solution {

public:

    int maxAreaOfIsland(vector<vector<int>>& grid) {

        int m = grid.size(), n = m ? grid[0].size() : 0;

        int x, y;

        int local\_area = 0, max\_area = 0;

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                if(grid[i][j] == 1) {

                    local\_area = dfs(grid, i, j);

                    max\_area = max(max\_area, local\_area);

                }

            }

        }

        return max\_area;

    }

    int dfs(vector<vector<int>>& grid, int i, int j) {

        if(grid[i][j] == 0) return 0;

        grid[i][j] = 0;

        int x, y, local\_area = 1;

        for(int k=0; k<4; ++k) {

            x = i + dir[k];

            y = j + dir[k+1];

            if(x>=0 && x<grid.size() && y>=0 && y<grid[0].size()) {

                local\_area += dfs(grid, x, y);

            }

        }

        return local\_area;

    }

};

递归二

vector<int> dir{-1, 0, 1, 0, -1};

class Solution {

public:

    int maxAreaOfIsland(vector<vector<int>>& grid) {

        int m = grid.size(), n = m ? grid[0].size() : 0;

        int x, y;

        int local\_area = 0, max\_area = 0;

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                local\_area = dfs(grid, i, j);

                max\_area = max(max\_area, local\_area);

            }

        }

        return max\_area;

    }

    int dfs(vector<vector<int>>& grid, int i, int j) {

        if(i<0 || i>=grid.size() || j<0 || j>=grid[0].size()) return 0;

        if(grid[i][j] == 0) return 0;

        grid[i][j] = 0;

        int x, y, local\_area = 1;

        for(int k=0; k<4; ++k) {

            x = i + dir[k];

            y = j + dir[k+1];

            local\_area += dfs(grid, x, y);

        }

        return local\_area;

    }

};

## 547 省份数量

class Solution {public:

int findCircleNum(vector<vector<int>>& isConnected) {

int n = isConnected.size();

int count = 0;

vector<bool> visited(n, false);

for(int i=0; i<n; ++i) {

if(!visited[i]) {

dfs(isConnected, i, visited);

++count;

}

}

return count;

}

void dfs(vector<vector<int>>& isConnected, int i, vector<bool>& visited) {

if(visited[i]) return;

visited[i] = true;

for(int j=0; j<isConnected.size(); ++j) {

if(!visited[j] && isConnected[i][j] == 1) {

dfs(isConnected, j, visited);

}

}

}

};

/\*

这个题实际上是一个树的遍历，树的节点数为n，每一个节点有n-1个相邻节点。只不过在遍历相邻节点的时候，要注意两方面的剪枝：

1、序号比i小的要剪掉（这个后面复盘发现，并不能剪掉）

2、已经visit的要剪掉

还有，dfs的定义要白话一下：

将与节点i相连的，且未标记为visited（即未分组）的节点都标记为visited。

\*/

接下来是并查集解法：

class UF {

private:

    int count\_;

    vector<int> parent\_;

    vector<int> size\_;

    int find(int x) {

        while(parent\_[x] != x) {

            parent\_[x] = parent\_[parent\_[x]];

            x = parent\_[x];

        }

        return x;

    }

public:

    UF(int n) {

        count\_ = n;

        parent\_.resize(n);

        size\_.resize(n);

        for(int i=0; i<n; ++i) {

            parent\_[i] = i;

            size\_[i] = 1;

        }

    }

    void nodeUnion(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        if(root\_p == root\_q) return;

        if(size\_[root\_p] < size\_[root\_q]) {

            parent\_[root\_p] = root\_q;

            size\_[root\_q] += size\_[root\_p];

        }

        else {

            parent\_[root\_q] = root\_p;

            size\_[root\_p] += size\_[root\_q];

        }

        --count\_;

    }

    bool isConnected(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        return root\_p == root\_q;

    }

    int count() {return count\_;}

};

class Solution {

public:

    int findCircleNum(vector<vector<int>>& isConnected) {

        int n = isConnected.size();

        UF uf(n);

        for(int i=0; i<n; ++i) {

            for(int j=i+1; j<n; ++j) {

                if(isConnected[i][j] == 1)

                    uf.nodeUnion(i, j);

            }

        }

        return uf.count();

    }

};

## 417 太平洋大西洋水流问题

vector<int> dir{-1, 0, 1, 0, -1};

class Solution {

public:

    vector<vector<int>> pacificAtlantic(vector<vector<int>>& matrix) {

        int m = matrix.size(), n = m ? matrix[0].size() : 0;

        vector<vector<bool>> can\_reach\_p(m, vector<bool>(n, false));

        vector<vector<bool>> can\_reach\_a(m, vector<bool>(n, false));

        for(int i=0; i<m; ++i) {

            dfs(matrix, can\_reach\_p, i, 0);

            dfs(matrix, can\_reach\_a, i, n-1);

        }

        for(int j=0; j<n; ++j) {

            dfs(matrix, can\_reach\_p, 0, j);

            dfs(matrix, can\_reach\_a, m-1, j);

        }

        vector<vector<int>> ans;

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                if(can\_reach\_a[i][j] && can\_reach\_p[i][j]) {

                    ans.push\_back({i,j});

                }

            }

        }

        return ans;

    }

    void dfs(vector<vector<int>>& matrix, vector<vector<bool>>& can\_reach, int r, int c) {

        if(can\_reach[r][c] == true) return;

        can\_reach[r][c] = true;

        int x, y;

        for(int i=0; i<4; ++i) {

            x = r + dir[i];

            y = c + dir[i+1];

            if(x>=0 && x<matrix.size() && y>=0 && y<matrix[0].size() && matrix[x][y] >= matrix[r][c]) {

                dfs(matrix, can\_reach, x, y);

            }

        }

    }

};

这个题看似简单，其实也不是特别好做对。要搞清几个概念：

1. 节点是什么？节点是每一个格子，其子节点是什么呢？是它的四个邻居
2. 怎么剪枝？用visited来标记，这里就是can\_reach

## 79 单词搜索

vector<int> dir{-1,0,1,0,-1};

class Solution {

public:

    bool exist(vector<vector<char>>& board, string word) {

        bool find = false;

        int m = board.size(), n = m ? board[0].size() : 0;

        vector<vector<bool>> visited(m, vector<bool>(n, false));

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                backtrack(board, i, j, word, 0, visited, find);

            }

        }

        return find;

    }

    void backtrack(vector<vector<char>>& board, int i, int j, const string& word, int pos, vector<vector<bool>>& visited, bool& find) {

        if(i<0 || i>=board.size() || j<0 || j>=board[0].size()) return;

        if(find) return;

        if(visited[i][j]) return;

        if(board[i][j] != word[pos]) return;

        if(pos == word.size() - 1) {

            find = true;

            return;

        }

        visited[i][j] = true;

        int x, y;

        for(int k=0; k<4; ++k) {

            x = i + dir[k];

            y = j + dir[k+1];

            backtrack(board, x, y, word, pos+1, visited, find);

        }

        visited[i][j] = false;

    }

};

/\*

首先要认识到，计算机解决这种问题就是穷举

怎么解决这个问题呢？要站在一个节点上，思考如何解决问题

假设目前走到的节点是(i,j)，已走过的状态记录在visited，是否找到的状态记录在find，word的位置为pos

\*/

## 934 最短的桥

vector<int> dir{-1, 0, 1, 0, -1};

class Solution {

public:

    int shortestBridge(vector<vector<int>>& A) {

        int m = A.size(), n = m ? A[0].size() : 0;

        queue<pair<int,int>> points;

        bool first\_island\_found = false;

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                if(A[i][j] == 1) {

                    dfs(A, i, j, points);

                    first\_island\_found = true;

                    break;

                }

            }

            if(first\_island\_found) break;

        }

        int level = 0;

        while(!points.empty()) {

            ++level;

            int size = points.size();

            for(int i=0; i<size; ++i) {

                auto [r, c] = points.front();

                points.pop();

                int x, y;

                for(int k=0; k<4; ++k) {

                    x = dir[k] + r;

                    y = dir[k+1] + c;

                    if(x<0 || x>=m || y<0 || y>=n) continue;

                    if(A[x][y] == 1) return level;

                    if(A[x][y] == 2) continue;

                    if(A[x][y] == 0) {

                        points.push({x,y});

                        A[x][y] = 2;

                    }

                }

            }

        }

        return level;

    }

    // 深度优先搜索遍历

    // 把遍历过的岛屿赋值为2，同时把接触到的0存到队列points里

    void dfs(vector<vector<int>>& A, int i, int j, queue<pair<int,int>>& points) {

        if(A[i][j] == 2) return;

        if(A[i][j] == 0) {

            points.push({i, j});

            return;

        }

        int m = A.size(), n = m ? A[0].size() : 0;

        A[i][j] = 2;

        int x, y;

        for(int k=0; k<4; ++k) {

            x = i + dir[k];

            y = j + dir[k+1];

            if(x>=0 && x<m && y>=0 && y<n) {

                dfs(A, x, y, points);

            }

        }

    }

};

/\*

这个题的技巧在于，通过深度优先搜索第一个岛屿的同时，把第一个岛屿外围的一圈也存到队列里，这样第二步的广度优先就能同时展开

还有一个易错的地方是：一定要注意遍历过的地方要打上记号，本题的记号是A[x][y] = 2;

\*/

## 126 单词接龙 II

class Solution {

public:

    vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {

        vector<vector<string>> ans;

        unordered\_set<string> dict;

        for(const auto& word : wordList) dict.insert(word);

        if(!dict.count(endWord)) return ans;

        queue<string> q;

        q.push(beginWord);

        dict.erase(beginWord);

        int distance = 0;

        unordered\_map<string, unordered\_set<string>> next;

        bool found = false;

        while(!q.empty()) {

            int sz = q.size();

            unordered\_set<string> visited;

            for(int i=0; i<sz; ++i) {

                string cur = q.front();

                if(cur == endWord) {

                    found = true;

                    break;

                }

                q.pop();

                dict.erase(cur);

                for(int j=0; j<cur.size(); ++j) {

                    for(int k=0; k<26; ++k) {

                        string temp = cur;

                        temp[j] = 'a' + k;

                        if(dict.count(temp)) {

                            q.push(temp);

                            next[cur].insert(temp);

                            visited.insert(temp);

                        }

                    }

                }

            }

            for(auto elem : visited) {

                dict.erase(elem);

            }

            if(found) break;

            ++distance;

        }

        cout << distance << endl;

        for(auto elem : next) {

            cout << elem.first << " : " <<  endl;

            for(auto n : elem.second) cout << n << " ";

            cout << endl;

        }

        if(found) {

            vector<string> path{beginWord};

            backtrack(beginWord, endWord, next, path, ans);

        }

        return ans;

    }

    void backtrack(const string& src, const string& dst, unordered\_map<string, unordered\_set<string>>& next,

                    vector<string>& path, vector<vector<string>>& ans) {

        if(src == dst) {

            ans.push\_back(path);

            return;

        }

        for(const auto& s : next[src]) {

            path.push\_back(s);

            backtrack(s, dst, next, path, ans);

            path.pop\_back();

        }

    }

};

/\*

这道题起点和终点均为已知，可以使用双向的BFS。

我要写一遍单向，写一遍双向，做下对比。

\*/

/\*

这个题坑也比较多，属于捣鼓了半天也没解决利索的

几个要注意的点：

1、防重复的话，就要删字典，但是删字典又比较讲究，比如while循环里面的一圈只能同时删

2、也要注意dfs的写法

这道题属于找到所有路径的BFS题型，这种类型要注意的点就是visited要在一圈遍历结束时同时标记，不能分开标记

\*/

下面是双向BFS的优化解法

class Solution {

public:

    vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {

        vector<vector<string>> ans;

        unordered\_set<string> dict;

        for(const auto& word : wordList) dict.insert(word);

        if(!dict.count(endWord)) return ans;

        unordered\_set<string> q1{beginWord}, q2{endWord};

        dict.erase(beginWord);

        dict.erase(endWord);

        unordered\_map<string, unordered\_set<string>> next;

        bool found = false;

        bool reversed = false;

        while(!q1.empty()) {

            unordered\_set<string> q;

            for(const auto& w1 : q1) {

                string s = w1;

                for(int i=0; i<s.size(); ++i) {

                    char ch = s[i];

                    for(int j=0; j<26; ++j) {

                        s[i] = 'a' + j;

                        if(q2.count(s)) {

                            reversed ? (next[s].insert(w1)) : (next[w1].insert(s));

                            found = true;

                        }

                        if(dict.count(s)) {

                            reversed ? (next[s].insert(w1)) : (next[w1].insert(s));

                            q.insert(s);

                        }

                    }

                    s[i] = ch;

                }

            }

            if(found) break;

            for(const auto& w : q) dict.erase(w);

            if(q.size() <= q2.size()) {

                q1 = q;

            }

            else {

                q1 = q2;

                q2 = q;

                reversed = !reversed;

            }

        }

        if(found) {

            vector<string> path{beginWord};

            backtrack(beginWord, endWord, next, path, ans);

        }

        return ans;

    }

    void backtrack(const string& src, const string& dst, unordered\_map<string, unordered\_set<string>>& next,

                    vector<string>& path, vector<vector<string>>& ans) {

        if(src == dst) {

            ans.push\_back(path);

            return;

        }

        for(const auto& s : next[src]) {

            path.push\_back(s);

            backtrack(s, dst, next, path, ans);

            path.pop\_back();

        }

    }

};

/\*

起点和终点分别开始往中间找的BFS类型，设置了reversed的bool变量，在结尾根据q1和q的大小进行BFS的优化，很棒！

\*/

## 257 二叉树的所有路径

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode() : val(0), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

 \* };

 \*/

class Solution {

    string vecToString(vector<int>& vec) {

        string ret;

        if(vec.size() == 0) {

            ret = "";

        }

        else if(vec.size() == 1) {

            ret = to\_string(vec[0]);

        }

        else {

            ret += to\_string(vec[0]);

            for(int i=1; i<vec.size(); ++i) {

                ret.push\_back('-');

                ret.push\_back('>');

                ret += to\_string(vec[i]);

            }

        }

        return ret;

    }

public:

    vector<string> binaryTreePaths(TreeNode\* root) {

        vector<int> path;

        vector<vector<int>> ans;

        vector<string> ret;

        if(!root) return ret;

        path.push\_back(root->val);

        dfs(root, path, ans);

        for(auto vec : ans) {

            // for(auto elem : vec) cout << elem << endl;

            ret.push\_back(vecToString(vec));

        }

        return ret;

    }

    void dfs(TreeNode\* root, vector<int>& path, vector<vector<int>>& ans) {

        if(!root) return;

        if(!root->left && !root->right) {

            ans.push\_back(path);

            return;

        }

        if(root->left) {

            path.push\_back(root->left->val);

            dfs(root->left, path, ans);

            path.pop\_back();

        }

        if(root->right) {

            path.push\_back(root->right->val);

            dfs(root->right, path, ans);

            path.pop\_back();

        }

    }

};

Dfs，回溯，没啥太大难度

## 310 最小高度树

struct MyNode {

    unordered\_set<int> neighbors;

    bool isLeaf() {return neighbors.size() == 1;}

};

class Solution {

public:

    vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {

        if(edges.empty()) return {0};

        if(n==1) return {0};

        vector<MyNode> tree(n);

        for(auto edge : edges) {

            tree[edge[0]].neighbors.insert(edge[1]);

            tree[edge[1]].neighbors.insert(edge[0]);

        }

        vector<int> q, new\_q;

        for(int i=0; i<n; ++i) {

            if(tree[i].isLeaf()) q.push\_back(i);

        }

        while(true) {

            for(auto leaf : q) {

                for(auto nb : tree[leaf].neighbors) {

                    tree[nb].neighbors.erase(leaf);

                    if(tree[nb].isLeaf()) new\_q.push\_back(nb);

                }

            }

            if(new\_q.empty()) return q;

            swap(q, new\_q);

            new\_q.clear();

        }

    }

};

/\*

这个题的意思是从外层的叶子节点出发，用BFS的思路层层剥皮，直到找到最后的根节点。

意思大概就是如果用叶子节点做根节点，那么结果一定不是高度最小的。

具体原理也不是太清楚，链接如下：

https://github.com/CR7dongfeng/notes/blob/master/leetcode/leetcode-301-350.md

本题留待以后再深究。

\*/

# 动态规划

## 70 爬楼梯

class Solution {

public:

    int climbStairs(int n) {

        if(n==1) return 1;

        if(n==2) return 2;

        vector<int> dp(n+1, 0);

        dp[1] = 1; dp[2] = 2;

        for(int i=3; i<=n; ++i) {

            dp[i] = dp[i-1] + dp[i-2];

        }

        return dp[n];

    }

};

没啥难度，不用看第二遍了

## 413 等差数列划分

class Solution {

public:

    int numberOfArithmeticSlices(vector<int>& A) {

        int n = A.size();

        if(n<3) return 0;

        vector<int> dp(n, 0);

        for(int i=2; i<n; ++i) {

            if(A[i]-A[i-1] == A[i-1]-A[i-2]) {

                dp[i] = dp[i-1] + 1;

            }

        }

        return accumulate(dp.begin(), dp.end(), 0);

    }

};

// 这个题很简单，dp的定义是以i结尾的子数组的性质。但是也值得一看

## 542 01矩阵

class Solution {

public:

    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {

        if(matrix.empty()) return {};

        int m = matrix.size(), n = m ? matrix[0].size() : 0;

        vector<vector<int>> dp(m, vector<int>(n, INT\_MAX-1));

        for(int i=0; i<m; ++i) {

            for(int j=0; j<n; ++j) {

                if(matrix[i][j] == 0) {

                    dp[i][j] = 0;

                }

                else {

                    if(j>0) {

                        dp[i][j] = min(dp[i][j], dp[i][j-1] + 1);

                    }

                    if(i>0) {

                        dp[i][j] = min(dp[i][j], dp[i-1][j] + 1);

                    }

                }

            }

        }

        for(int i=m-1; i>=0; --i) {

            for(int j=n-1; j>=0; --j) {

                if(matrix[i][j] != 0) {

                    if(j<n-1) {

                        dp[i][j] = min(dp[i][j], dp[i][j+1] + 1);

                    }

                    if(i<m-1) {

                        dp[i][j] = min(dp[i][j], dp[i+1][j] + 1);

                    }

                }

            }

        }

        return dp;

    }

};

/\*

dp[i][j] = min(dp[i][j], dp[i-1][j] + 1, dp[i][j+1] + 1, dp[i+1][j] + 1, dp[i][j-1] + 1)

四方向搜索，每次遍历两个方向，很有意思，这个题并不好想。

\*/

## 221 最大正方形

class Solution {

public:

    int maximalSquare(vector<vector<char>>& matrix) {

        int m = matrix.size(), n = m ? matrix[0].size() : 0;

        int max\_side = 0;

        vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

        for(int i=1; i<=m; ++i) {

            for(int j=1; j<=n; ++j) {

                if(matrix[i-1][j-1] == '1') {

                    dp[i][j] = min(dp[i-1][j-1], min(dp[i][j-1], dp[i-1][j])) + 1;

                    max\_side = max(max\_side, dp[i][j]);

                }

            }

        }

        return max\_side \* max\_side;

    }

};

/\*

增广类型的dp数组

\*/

## 279 完全平方数

class Solution {

public:

    int numSquares(int n) {

        vector<int> dp(n+1, INT\_MAX);

        dp[0] = 0;

        for(int i=1; i<=n; ++i) {

            for(int k=1; k\*k<=i; ++k) {

                dp[i] = min(dp[i], dp[i-k\*k] + 1);

            }

        }

        return dp[n];

    }

};

/\*

dp[i]的定义，得到数字i所需要的完全平方数的数量

所以dp[0] = 0 得到数字0不需要完全平方数

这道题的关键是dp的递推公式。

for(int k=1; k\*k <= n; ++k) {

    dp[n] = min(dp[n], dp[n-k\*k] + 1)

}

\*/

## 91 解码方法

class Solution {

public:

    int numDecodings(string s) {

        int n = s.size();

        vector<int> memo(n+1, -1);

        return dp(s, 0, memo);

    }

    int dp(const string& s, int i, vector<int>& memo) {

        int n = s.size();

        if(i == n) return 1;

        if(i == n-1) {

            if(s[i] == '0') {

                return 0;

            }

            else {

                return 1;

            }

        }

        if(s[i] == '0') {

            return 0;

        }

        if(memo[i] != -1) return memo[i];

        int ret = 0;

        if((s[i] == '1' && s[i + 1] != '0') || (s[i] == '2' && s[i+1] > '0' && s[i+1] <= '6')) {

            ret = dp(s, i+2, memo) + dp(s, i+1, memo);

        }

        else if(s[i] <= '2' && s[i+1] == '0'){

            ret = dp(s, i+2, memo);

        }

        else {

            ret = dp(s, i+1, memo);

        }

        memo[i] = ret;

        return ret;

    }

};

/\*

这是一道自己经过千辛万苦的改进，用dp函数 + memo数组的形式搞定的一道题

小错误犯了很多

要注意两点：

1、题目的dp函数怎么转化

2、要理解i==n这种base case,base case 下返回1，而不是返回0

\*/

这道题的方法选择上，有两个维度

1. 用dp函数 + memo，还是直接上dp数组
2. dp[i]表示0到i，还是i到结尾。这个题用后者表示要方便一些。

class Solution {

public:

    int numDecodings(string s) {

        int n = s.size();

        if(n==0) return 0;

        if(n==1) return s[0] == '0' ? 0 : 1;

        vector<int> dp(n+1, 1);

        dp[n] = 1;

        dp[n-1] = s[n-1] == '0' ? 0 : 1;

        for(int i=n-2; i>=0; --i) {

            if(s[i] == '0') {

                dp[i] = 0;

            }

            else if((s[i] == '1' && s[i+1] != '0') || (s[i] == '2' && s[i+1] > '0' && s[i+1] < '7')) {

                dp[i] = dp[i+1] + dp[i+2];

            }

            else if(s[i] <= '2' && s[i+1] == '0') {

                dp[i] = dp[i+2];

            }

            else {

                dp[i] = dp[i+1];

            }

        }

        return dp[0];

    }

};

/\*

dp数组解法

\*/

## 139 单词拆分

class Solution {

public:

    bool wordBreak(string s, vector<string>& wordDict) {

        int m = wordDict.size();

        int n = s.size();

        // dp[i][j] - 用字典前i个单词，能否构造s前j个字符？

        vector<vector<bool>> dp(m+1, vector<bool>(n+1, false));

        // base case : 用字典可以构造空串，但是用空字典不能构造非空串！

        for(int i=0; i<=m; ++i) {

            dp[i][0] = true;

        }

        for(int i=1; i<=m; ++i) {

            for(int j=1; j<=n; ++j) {

                // 这里面的处理不同于一般的完全背包了

                dp[i][j] = dp[i-1][j];

                for(int k=0; k<i; ++k) {

                    int len = wordDict[k].size();

                    if(j>=len && (wordDict[k] == s.substr(j-len, len))) {

                        dp[i][j] = dp[i-1][j] || dp[i][j-len];

                    }

                }

            }

        }

        return dp[m][n];

    }

};

/\*

啥叫背包问题

给你一个最大容纳重量为W的背包，以及若干个参数为(v,w)的物品，问你怎么装这个背包，可以使背包里的总价值最大

如果物品只能使用一次，则成为0-1背包问题；如果物品可以使用多次，则为完全背包问题

双状态 ： v = dp[i][w]   最多只用前i个物品，去装容量为w的背包，能得到的最大价值为v

递推公式：

0-1背包：dp[i][w] = max(dp[i-1][w], dp[i-1][w-w[i]] + v[i])

完全背包： dp[i][w] = max(dp[i-1][w], dp[i][w-w[i]] + v[i])

如果压缩空间的话，注意0-1背包逆向遍历

这个题有时候不归到完全背包问题里面，因为字符串拼接不等同于重量相加问题，里面还有一个顺序问题要考量

\*/

以上方法是基本的方法，下面是压缩空间方法

class Solution {

public:

    bool wordBreak(string s, vector<string>& wordDict) {

        int m = wordDict.size();

        int n = s.size();

        vector<bool> dp(n+1, false);

        dp[0] = true;

        for(int i=1; i<=n; ++i) {

            for(const auto& word : wordDict) {

                int len = word.size();

                if(i>=len && word == s.substr(i-len, len)) {

                    dp[i] = dp[i] || dp[i-len];

                }

            }

        }

        return dp[n];

    }

};

## 474 一和零

class Solution {

public:

    int findMaxForm(vector<string>& strs, int m, int n) {

        vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

        for(const auto& str : strs) {

            auto [count0, count1] = count(str);

            for(int i=m; i>=count0; --i) {

                for(int j=n; j>=count1; --j) {

                    dp[i][j] = max(dp[i][j], 1 + dp[i-count0][j-count1]);

                }

            }

        }

        return dp[m][n];

    }

    pair<int, int> count(const string& s) {

    int count0 = s.length(), count1 = 0;

    for(auto c : s) {

        if(c == '1') {

            ++count1;

            --count0;

        }

    }

    return make\_pair(count0, count1);

}

};

/\*

背包问题分析：

双背包 m 和 n。二维背包经典题目

\*/

## 343 整数拆分

class Solution {

public:

    int integerBreak(int n) {

        vector<int> dp(n+1, 1);

        for(int i=2; i<=n; ++i) {

            for(int j=1; j<=i/2; ++j) {

                // cout << "dp" << i << " = " << "dp" << j << " \* " << "dp" << i-j << endl;

                dp[i] = max( dp[i], max(j, dp[j]) \* max(i-j, dp[i-j]));

            }

        }

        // cout << dp[n] << endl;

        return dp[n];

    }

};

// 这个解法是自己想的，感觉还可以，dp公式有点绕

// 据说还有数学解法，以后再说吧

int integerBreak(int n) {

if (n == 2) return 1;

if (n == 3) return 2;

if (n == 4) return 4;

if (n == 5) return 6;

if (n == 6) return 9;

return 3 \* integerBreak(n - 3);

}

## 646 最长数对链

无重叠区间的翻版问题，先掌握一个贪心算法再说吧

class Solution {

public:

    int findLongestChain(vector<vector<int>>& pairs) {

        int n = pairs.size();

        if(n==0) return 0;

        sort(pairs.begin(), pairs.end(), [](const vector<int>& a, const vector<int>& b) {

            return a[1] < b[1] || (a[1]==b[1] && a[0] < b[0]);

        });

        int count = 1;

        int end = pairs[0][1];

        for(int i=1; i<n; ++i) {

            if(pairs[i][0] > end) {

                ++count;

                end = pairs[i][1];

            }

        }

        return count;

    }

};

贴一个二分的解法：跟最长递增子序列很像很像

class Solution {

public:

    int findLongestChain(vector<vector<int>>& pairs) {

        int n = pairs.size();

        if(n==0) return 0;

        sort(pairs.begin(), pairs.end(), [](const vector<int>& a, const vector<int>& b) {

            return (a[0]<b[0])||(a[0]==b[0]&&a[1]<b[1]);

        });

        vector<vector<int>> dp;

        for(const auto& p : pairs) {

            int lo = 0, hi = dp.size(), mid;

            while(lo < hi) {

                mid = lo + (hi-lo) / 2;

                if(p[0] < dp[mid][1]) {

                    hi = mid;

                }

                else if(p[0] > dp[mid][1]) {

                    lo = mid + 1;

                }

                else {

                    hi = mid;

                }

            }

            if(lo >= dp.size()) dp.emplace\_back(p);

            else if(dp[lo][1] > p[1]) dp[lo] = p;

        }

        return dp.size();

    }

};

/\*

二分的解法

注意跟贪心法的排序方式不同

\*/

## 376 摆动序列

class Solution {

public:

    int wiggleMaxLength(vector<int>& nums) {

        int n = nums.size();

        if(n<=1) return n;

        vector<pair<int, int>> dp(n, {1,1});

        int max\_length = 1;

        for(int i=1; i<n; ++i) {

            for(int j=0; j<i; ++j) {

                if(nums[i] > nums[j]) {

                    dp[i].first = max(dp[i].first, dp[j].second + 1);

                    max\_length = max(max\_length, dp[i].first);

                }

                else if(nums[i] < nums[j]) {

                    dp[i].second = max(dp[i].second, dp[j].first + 1);

                    max\_length = max(max\_length, dp[i].second);

                }

            }

        }

        return max\_length;

    }

};

/\*

完全自己想的算法，居然一遍就过了

思路类似300 最长递增子序列

dp[i]包含两个数，分别表示以i结尾的结尾升序摆动序列和以i结尾的结尾降序摆动序列

\*/

class Solution {

public:

    int wiggleMaxLength(vector<int>& nums) {

        int n = nums.size();

        if(n<=1) return n;

        vector<int> up(n), down(n);

        up[0] = down[0] = 1;

        for(int i=1; i<n; ++i) {

            if(nums[i] == nums[i-1]) {

                up[i] = up[i-1];

                down[i] = down[i-1];

            }

            else if(nums[i] > nums[i-1]) {

                up[i] = max(down[i-1] + 1, up[i-1]);

                down[i] = down[i-1];

            }

            else if(nums[i] < nums[i-1]) {

                up[i] = up[i-1];

                down[i] = max(up[i-1] + 1, down[i-1]);

            }

        }

        return max(up[n-1], down[n-1]);

    }

};

/\*

来自官网的思路，一个好的递推公式，节省了算法复杂度

\*/

# 分治法

## 932 漂亮数组

class Solution {

public:

    vector<int> beautifulArray(int N) {

        unordered\_map<int, vector<int>> memo;

        return f(N, memo);

    }

    vector<int> f(int N, unordered\_map<int, vector<int>>& memo) {

        if(N==1) return {1};

        if(memo.count(N)) return memo[N];

        vector<int> ans;

        for(auto x : f((N+1)/2, memo)) {

            ans.push\_back(2\*x - 1);

        }

        for(auto x : f((N)/2, memo)) {

            ans.push\_back(2\*x);

        }

        memo[N] = ans;

        return ans;

    }

};

/\*

白话说一下思路：

这道题是把N分为奇偶两部分，(N+1)/2个奇数在前，N/2个偶数在后，这样变成两个子问题

原问题和子问题对应的数字存在映射，奇数是2N-1，偶数是2N。

以上是代码的思路

但是，本题代码思路不简单，更难的确是推导的难度，为什么可以通过分奇偶来解决？问什么映射不影响子问题？

你要好好想明白哦！

\*/

下面是自底向上的写法：

class Solution {

public:

    vector<int> beautifulArray(int N) {

        if(N==1) return {1};

        vector<vector<int>> dp(N+1);

        dp[1] = {1};

        for(int i=2; i<=N; ++i) {

            for(auto x : dp[(i+1)/2]) {

                dp[i].push\_back(2\*x-1);

            }

            for(auto x : dp[i/2]) {

                dp[i].push\_back(2\*x);

            }

        }

        return dp[N];

    }

};

/\*

这样反而效率低了

\*/

# 数学问题

## 公因数公倍数

int gcd(int a, int b) {  
 return b == 0 ? a : (b, a%b);  
}  
  
int lcm(int a, int b) {  
 return a \* b / gcd(a, b);  
}

## 204 计数质数

class Solution {

public:

    int countPrimes(int n) {

        vector<bool> is\_prime(n, true);

        for(int i=2; i\*i<n; ++i) {

            if(is\_prime[i]) {

                for(int j=i\*i; j<n; j+=i) {

                    is\_prime[j] = false;

                }

            }

        }

        int count = 0;

        for(int i=2; i<n; ++i) {

            if(is\_prime[i]) ++count;

        }

        return count;

    }

};

/\*

高效查找素数需要注意的几点：

1、判断一个数n是不是素数，在遍历时只需要遍历到sqrt(n)

2、内层循环从i的平方开始算。

\*/

## 504 七进制数

class Solution {

public:

    string convertToBase7(int num) {

        if(num == 0) return "0";

        bool is\_negative = num < 0;

        if(is\_negative) num = -num;

        string ans;

        while(num) {

            int a = num / 7, b = num % 7;

            ans = to\_string(b) + ans;

            num = a;

        }

        return is\_negative ? "-" + ans : ans;

    }

};

## 415 字符串相加

class Solution {

public:

    string addStrings(string num1, string num2) {

        reverse(num1.begin(), num1.end());

        reverse(num2.begin(), num2.end());

        int onelen = num1.length(), twolen = num2.length();

        if(onelen < twolen) {

            swap(num1, num2);

            swap(onelen, twolen);

        }

        int addbit = 0;

        string output = "";

        for(int i=0; i<twolen; ++i) {

            int cur\_sum = (num1[i] - '0') + (num2[i] - '0') + addbit;

            output += to\_string((cur\_sum) % 10);

            addbit = cur\_sum >= 10 ? 1 : 0;

        }

        for(int i=twolen; i<onelen; ++i) {

            int cur\_sum = (num1[i] - '0') + addbit;

            output += to\_string((cur\_sum) % 10);

            addbit = cur\_sum >= 10 ? 1 : 0;

        }

        if(addbit) {

            output += "1";

        }

        reverse(output.begin(), output.end());

        return output;

    }

};

## 326 3的幂

class Solution {

public:

    bool isPowerOfThree(int n) {

        return fmod(log10(n) / log10(3), 1) == 0;

    }

};

## 384 打乱数组

class Solution {

    vector<int> origin\_;

public:

    Solution(vector<int>& nums) {

        origin\_ = std::move(nums);

    }

    /\*\* Resets the array to its original configuration and return it. \*/

    vector<int> reset() {

        return origin\_;

    }

    /\*\* Returns a random shuffling of the array. \*/

    vector<int> shuffle() {

        vector<int> shuffled(origin\_);

        int n = origin\_.size();

        // 正向反向洗牌都可以

        // for(int i=n-1; i>=0; --i) {

        //     swap(shuffled[i], shuffled[rand() % (i+1)]);

        // }

        for(int i=0; i<n; ++i) {

            swap(shuffled[i], shuffled[i + rand() % (n-i)]);

        }

        return shuffled;

    }

};

/\*\*

 \* Your Solution object will be instantiated and called as such:

 \* Solution\* obj = new Solution(nums);

 \* vector<int> param\_1 = obj->reset();

 \* vector<int> param\_2 = obj->shuffle();

 \*/

## 168 Excel表列名称

class Solution {

public:

    string convertToTitle(int n) {

        if(n == 1) return "A";

        int num = n;

        string output = "";

        while(num) {

            --num;

            int a = num / 26;

            int b = num % 26;

            output.push\_back('A' + b);

            num = a;

        }

        reverse(output.begin(), output.end());

        return output;

    }

};

递归解法：

class Solution {

public:

    string convertToTitle(int n) {

        if(n==0) return "";

        --n;

        return convertToTitle(n/26) + (char) (n%26 + 'A');

    }

};

注意这个题跟递归类似，但有所区别。

## 67 二进制求和

class Solution {

public:

    string addBinary(string a, string b) {

        reverse(a.begin(), a.end());

        reverse(b.begin(), b.end());

        string output = "";

        int alen = a.length(), blen = b.length();

        if(alen < blen) {

            swap(a, b);

            swap(alen, blen);

        }

        int addbit = 0;

        for(int i=0; i<blen; ++i) {

            int cur\_sum = (a[i] - '0') + (b[i] - '0') + addbit;

            output += to\_string(cur\_sum % 2);

            addbit = cur\_sum >= 2 ? 1 : 0;

        }

        for(int i=blen; i<alen; ++i) {

            int cur\_sum = (a[i] - '0') + addbit;

            output += to\_string(cur\_sum % 2);

            addbit = cur\_sum >= 2 ? 1 : 0;

        }

        if(addbit) output += "1";

        reverse(output.begin(), output.end());

        return output;

    }

};

## 238 除自身以外数组的乘积

class Solution {

public:

    vector<int> productExceptSelf(vector<int>& nums) {

        int n = nums.size();

        vector<int> left\_product(n, 1);

        vector<int> right\_product(n, 1);

        vector<int> res(n, 1);

        for(int i=1; i<n; ++i) {

            left\_product[i] = left\_product[i-1] \* nums[i-1];

        }

        for(int i=n-2; i>=0; --i) {

            right\_product[i] = right\_product[i+1] \* nums[i+1];

        }

        for(int i=0; i<n; ++i) {

            res[i] = left\_product[i] \* right\_product[i];

        }

        return res;

    }

};

这道题是按照提示写出来的，跟贪心算法里面的发糖果是一个类型的。

## 462 最少移动次数使数组元素相等 II

class Solution {

public:

    int minMoves2(vector<int>& nums) {

        int n = nums.size();

        if(n<2) return 0;

        if(n==2) return abs(nums[0]-nums[1]);

        int median = findMedian(nums);

        int ret = 0;

        for(auto num : nums) {

            ret += abs( num - median );

        }

        return ret;

    }

    int findMedian(vector<int>& nums) {

        int n = nums.size();

        int lo = 0, hi = n-1, target = (lo+hi)/2, mid;

        while(lo < hi) {// 注意，这里不是lo==hi，因为我们必须要一个结果，lo==hi意味着结束

            mid = partition(nums, lo, hi); // 注意mid产生的方式是本题不一样的地方

            if(mid == target) {

                return nums[mid];

            }

            else if(mid < target) {

                lo = mid + 1;

            }

            else {

                hi = mid - 1;

            }

        }

        return nums[lo];

    }

    int partition(vector<int>& nums, int lo, int hi) {

        if(lo == hi) return lo;

        int pivot = nums[lo];

        int i = lo, j = hi + 1;

        while(true) {

            while(nums[++i] < pivot) {

                if(i==hi) break;

            }

            while(nums[--j] > pivot) {

                if(j==lo) break;

            }

            if(i>=j) break;

            swap(nums[i], nums[j]);

        }

        swap(nums[lo], nums[j]);

        return j;

    }

};

/\*

最小移动的元素，就是中位数！！！！用快速选择算法来找中位数，非常合适！！

\*/

## 169 多数元素

class Solution {

public:

    int majorityElement(vector<int>& nums) {

        int count = 0, elem = 0;

        for(int i=0; i<nums.size(); ++i) {

            if(count == 0) {

                elem = nums[i];

                count = 1;

            }

            else {

                if(nums[i] == elem) ++count;

                else --count;

            }

        }

        return elem;

    }

};

/// 这个算法比较独特，也没有详细理解，学习了，暂时略过

## 202 快乐数

class Solution {

public:

    bool isHappy(int n) {

        auto digitsum = [](int a) -> int {

            int sum = 0;

            while(a) {

                sum += (a%10) \* (a%10);

                a /= 10;

            }

            return sum;

        };

        int fast = n, slow = n;

        do {

            slow = digitsum(slow);

            fast = digitsum(digitsum(fast));

        } while(slow != fast);

        return slow == 1;

    }

};

快慢指针判圈法

## 470 用rand7()实现rand10()

// The rand7() API is already defined for you.

// int rand7();

// @return a random integer in the range 1 to 7

class Solution {

public:

    int rand10() {

        int row, col, idx;

        do {

            row = rand7();

            col = rand7();

            idx = (row - 1) \* 7 + col;

        } while(idx > 40);

        return idx % 10 + 1;

    }

};

# 位运算

## 461 汉明距离

class Solution {

public:

    int hammingDistance(int x, int y) {

        int diff = x ^ y;

        int ans = 0;

        while(diff) {

            ans += (diff & 1);

            diff >>= 1;

        }

        return ans;

    }

};

// 亦或 + 统计

## 190 颠倒二进制位

class Solution {

public:

    uint32\_t reverseBits(uint32\_t n) {

        uint32\_t ans = 0;

        for(int i=0; i<32; ++i) {

            ans <<= 1;

            ans += n & 1; // 取最后一位的写法

            n >>= 1;

        }

        return ans;

    }

};

## 136 只出现一次的数字

class Solution {

public:

    int singleNumber(vector<int>& nums) {

        int ans = 0;

        for(auto num : nums) {

            ans ^= num;

        }

        return ans;

    }

};

## 318 最大单词长度乘积

class Solution {

public:

    int maxProduct(vector<string>& words) {

        int ans = 0;

        unordered\_map<int, int> hash;

        for(auto word : words) {

            int size = word.size();

            int mask = 0;

            for(auto c : word) {

                mask |= 1 << (c-'a');

            }

            hash[mask] = max(hash[mask], size); // hash里面存的是含有相同字母的字符串里面的最大值

            for(auto [h\_mask, h\_len] : hash) {

                if((mask & h\_mask) == 0)

                    ans = max(ans, h\_len \* size);

            }

        }

        return ans;

    }

};

## 338 比特位计数

class Solution {

public:

    vector<int> countBits(int num) {

        vector<int> dp(num+1, 0);

        for(int i=1; i<=num; ++i) {

            if((i & 1) == 1) {

                dp[i] = dp[i-1] + 1;

            }

            else {

                dp[i] = dp[i>>1];

            }

        }

        return dp;

    }

};

## 693 交替位二进制数

class Solution {

public:

    bool hasAlternatingBits(int n) {

        long long a = n ^ (n >> 1); // 1010 ^ 101 = 1111

        return (a & (a + 1)) == 0;  // 1111 & 10000 = 0

    }

};

## 476 数字的补数

class Solution {

public:

    int findComplement(int num) {

        int res = 0;

        int pos = 0;

        while(num) {

            res += (num % 2 == 0) ? (1 << pos) : 0;

            num >>= 1;

            ++pos;

        }

        return res;

    }

};

## 260 只出现一次的数字III

// 这个题非常非常巧妙，

class Solution {

public:

    vector<int> singleNumber(vector<int>& nums) {

        long diff = accumulate(nums.begin(), nums.end(), 0, bit\_xor<int>());

        diff &= -diff;  // 保留最后一个1； n & (n-1)是去掉最后一个1

        int num\_a = 0, num\_b = 0;

        for(auto num : nums) {

            if(num & diff) num\_a ^= num;

            else num\_b ^= num;

        }

        return vector<int> {num\_a, num\_b};

    }

};

# 妙用数据结构

## 448 找到所有数组中消失的数字

class Solution {

public:

    vector<int> findDisappearedNumbers(vector<int>& nums) {

        for(auto num : nums) {

            int pos = abs(num) - 1;

            if(nums[pos] > 0) {

                nums[pos] = -nums[pos];

            }

        }

        vector<int> ans;

        for(int i=0; i<nums.size(); ++i) {

            if(nums[i] > 0) {

                ans.push\_back(i+1);

            }

        }

        return ans;

    }

};

数组映射法，没啥难度了

## 48 旋转图像

class Solution {

public:

    void rotate(vector<vector<int>>& matrix) {

        int n = matrix.size() - 1;

        for(int i=0; i<=n/2; ++i) {

            for(int j=i; j<n-i; ++j) {

                int temp = matrix[i][j];

                matrix[i][j] = matrix[n-j][i];

                matrix[n-j][i] = matrix[n-i][n-j];

                matrix[n-i][n-j] = matrix[j][n-i];

                matrix[j][n-i] = temp;

            }

        }

    }

};

// 要掌握这种从外到内给矩阵建立index的思路

n = matrix.size() - 1，这样定义n可以方便下标操作

i记录的是从外到内的圈数，i范围是闭区间[0, n/2]

j的取值范围也要注意讲究哦！

## 240 搜索二维矩阵II

class Solution {

public:

    bool searchMatrix(vector<vector<int>>& matrix, int target) {

        int m = matrix.size(), n = m ? matrix[0].size() : 0;

        if(m==0 || n==0) return false;

        int i=0, j=n-1;

        while(i<m && j>=0) {

            if(matrix[i][j] == target) return true;

            if(matrix[i][j] < target) ++i;

            else --j;

        }

        return false;

    }

};

从右上角开始的双指针搜索，很简单的

## 769 最多能完成排序的块

class Solution {

public:

    int maxChunksToSorted(vector<int>& arr) {

        int max\_val = 0;

        int part = 0;

        for(int i=0; i<arr.size(); ++i) {

            max\_val = max(max\_val, arr[i]);

            if(max\_val == i) ++part;

        }

        return part;

    }

};

// 这道题比较独特，记住这个解法就行

## 155 最小栈

写法一：

class MinStack {

    stack<int> s\_, min\_s\_;

public:

    /\*\* initialize your data structure here. \*/

    MinStack() {

    }

    void push(int x) {

        s\_.push(x);

        if(min\_s\_.empty() || min\_s\_.top() >= x) {

            min\_s\_.push(x);

        }

    }

    void pop() {

        if(!min\_s\_.empty() && (min\_s\_.top() == s\_.top())) {

            min\_s\_.pop();

        }

        s\_.pop();

    }

    int top() {

        return s\_.top();

    }

    int getMin() {

        return min\_s\_.top();

    }

};

/\*\*

 \* Your MinStack object will be instantiated and called as such:

 \* MinStack\* obj = new MinStack();

 \* obj->push(x);

 \* obj->pop();

 \* int param\_3 = obj->top();

 \* int param\_4 = obj->getMin();

 \*/

写法二：

一个写起来更简单但是时间复杂度略高的方法是，我们每次插入原栈时，都向新栈插入一次

原栈里所有值的最小值（新栈栈顶和待插入值中小的那一个）；每次从原栈里取出数字时，同样

取出新栈的栈顶。这样可以避免判断，但是每次都要插入和取出。

后面有时间再写吧

## 23 合并K个有序链表

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

struct Comp {

    bool operator() (ListNode\* l1, ListNode\* l2) {

        return l1->val > l2->val;

    }

};

class Solution {

public:

    ListNode\* mergeKLists(vector<ListNode\*>& lists) {

        if(lists.empty()) return nullptr;

        priority\_queue<ListNode\*, vector<ListNode\*>, Comp> pq;

        for(auto list : lists) {

            if(list) {

                pq.push(list);

            }

        }

        ListNode\* dummy = new ListNode(0), \*cur = dummy;

        while(!pq.empty()) {

            cur->next = pq.top();

            pq.pop();

            cur = cur->next;

            if(cur->next) pq.push(cur->next);

        }

        return dummy->next;

    }

};

/\*

建一个ListNode\*类型的小根堆，把每个链表都扔进去

每次循环把堆顶吐出来，同时把堆的后继节点推到堆里去

\*/

## 218 天际线问题

class Solution {

public:

    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {

        int n = buildings.size();

        int i = 0;

        priority\_queue<pair<int, int>> max\_heap;

        int cur\_x = 0, cur\_h = 0;

        vector<vector<int>> ans;

        while(i < n || !max\_heap.empty()) {

            // 确定cur\_x

            if(max\_heap.empty() || (i<n && buildings[i][0] <= max\_heap.top().second)) {

                cur\_x = buildings[i][0];

                while( i<n && cur\_x == buildings[i][0]) {

                    max\_heap.emplace(buildings[i][2], buildings[i][1]);

                    ++i;

                }

            }

            else {

                cur\_x = max\_heap.top().second;

                while(!max\_heap.empty() && cur\_x >= max\_heap.top().second) {

                    max\_heap.pop();

                }

            }

            // cur\_x更新完毕，下面是cur\_h

            cur\_h = max\_heap.empty() ? 0 : max\_heap.top().first;

            // 更新ans

            if(ans.empty() || ans.back()[1] != cur\_h) {

                ans.push\_back({cur\_x, cur\_h});

            }

        }

        return ans;

    }

};

/\*

天际线问题，最重要的不是代码，而是梳理思路

问题有两个维度，x轴的工作点和y轴的高度，思路围绕上面两者展开。

具体如下：

我们把最大高度交给优先级队列来自动生成，我们只管往里扔建筑物

我们关注的是cur\_x，cur\_h两个量，即当前工作点和对应天际线高度

对于一个新来的buildings[i],如果它有可能改变当前的cur\_x，则记录cur\_x,把它扔到优先级队列

如果它不能改变当前cur\_x，则堆顶的右端充当cur\_x

上面两行是确定cur\_x，下面则要确定cur\_h:

cur\_h一定是取自堆顶！！！如果堆为空，则等于0

最后，更新ans

有可能遍历完了，但堆还没清空完。要考虑到

\*/

## 128 最长连续序列

解法一、哈希表解法，来自101

class Solution {

public:

    int longestConsecutive(vector<int>& nums) {

        int ans = 0;

        unordered\_set<int> hash;

        for(auto num : nums) {

            hash.insert(num);

        }

        while(!hash.empty()) {

            int cur = \*hash.begin();

            hash.erase(cur);

            int prev = cur-1, next = cur+1;

            while(hash.count(prev)) {

                hash.erase(prev);

                --prev;

            }

            while(hash.count(next)) {

                hash.erase(next);

                ++next;

            }

            ans = max(ans, next - prev - 1);

        }

        return ans;

    }

};

解法二、并查集解法？不是很方便 放弃

## 149 直线上最多的点数

class Solution {

public:

    int maxPoints(vector<vector<int>>& points) {

        int n = points.size();

        if (n==0) return 0;

        int max\_points = 0;

        for(int i=0; i<n; ++i) {

            int same = 1, same\_x = 1;

            unordered\_map<long double, int> hash;

            for(int j=i+1; j<n; ++j) {

                if(points[i][0] == points[j][0]) {

                    ++same\_x;

                    if(points[i][1] == points[j][1]) {

                        ++same;

                    }

                }

                else {

                    long double dy = points[j][1] - points[i][1];

                    long double dx = points[j][0] - points[i][0];

                    ++hash[dy/dx];

                }

            }

            max\_points = max(max\_points, same\_x);

            for(auto num : hash) {

                max\_points = max(max\_points, num.second + same);

            }

        }

        return max\_points;

    }

};

/\*

思路：借助哈希表，思路很朴素，注意用long double

\*/

## 332 重新安排行程

最初自己写的一个解法，问题很大，两天搞不过。主要有以下问题

1. multiset<string>其实不好用，改用map<string, int>增加一个计数的int，避免了增删和迭代器错误
2. Dfs函数没有设置合理的停止条件
3. Dfs没有设置合理的返回值bool
4. 没有很好地利用vector<string>& ans，而只是把它当做记录器来用了。

下面是九牛二虎之后的正确写法：

1. 回溯写法：

class Solution {

public:

    vector<string> findItinerary(vector<vector<string>>& tickets) {

        unordered\_map<string, map<string, int>> hash;

        for(const auto& ticket : tickets) {

            ++hash[ticket[0]][ticket[1]];

        }

        vector<string> ans;

        ans.push\_back("JFK");

        dfs(hash, tickets.size(), ans);

        return ans;

    }

    bool dfs(unordered\_map<string, map<string, int>>& hash, int ticket\_num, vector<string>& ans) {

        if(ans.size() == ticket\_num + 1) {

            return true;

        }

        string start = ans.back();

        for(auto& [target, count] : hash[start]) {

            if(count > 0) {

                ans.push\_back(target);

                --count;

                if(dfs(hash, ticket\_num, ans)) return true;

                ++count;

                ans.pop\_back();

            }

        }

        return false;

    }

};

1. 迭代写法（栈）

class Solution {

public:

    vector<string> findItinerary(vector<vector<string>>& tickets) {

        unordered\_map<string, multiset<string>> hash;

        for(const auto& ticket : tickets) {

            hash[ticket[0]].insert(ticket[1]);

        }

        stack<string> s;

        s.push("JFK");

        vector<string> ans;

        while(!s.empty()) {

            string cur = s.top();

            if(hash[cur].empty()) {

                ans.push\_back(cur);

                s.pop();

            }

            else {

                s.push(\*hash[cur].begin());

                hash[cur].erase(hash[cur].begin());

            }

        }

        reverse(ans.begin(), ans.end());

        return ans;

    }

};

## 566 重塑矩阵

class Solution {

public:

    vector<vector<int>> matrixReshape(vector<vector<int>>& nums, int r, int c) {

        int m = nums.size(), n = m ? nums[0].size() : 0;

        if(m\*n != r\*c) return nums;

        vector<vector<int>> ans(r, vector<int>(c, 0));

        for(int k=0; k<m\*n; ++k) {

            ans[k/c][k%c] = nums[k/n][k%n];

        }

        return ans;

    }

};

/\*

二维映射到一维：

行列数为m,n; 坐标为i,j; 则对应的一维坐标x = i \* n + j

一维换算成二维：

i = x / n;

y = x % n;

完毕

\*/

## 217 存在重复元素

class Solution {

public:

    bool containsDuplicate(vector<int>& nums) {

        unordered\_set<int> hash;

        for(int num : nums) {

            if(hash.count(num))

                return true;

            else {

                hash.insert(num);

            }

        }

        return false;

    }

};

比较简单，循环一次，配合哈希表，搞定

## 697 数组的度

class Solution {

public:

    int findShortestSubArray(vector<int>& nums) {

        unordered\_map<int, int> counter, max\_idx, min\_idx;

        int degree = 0;

        int ret = nums.size();

        for(int i=0; i<nums.size(); ++i) {

            int num = nums[i];

            ++counter[num];

            degree = max(degree, counter[num]);

            max\_idx[num] = i;

            if(!min\_idx.count(num)) {

                min\_idx[num] = i;

            }

        }

        for(const auto& [num, cnt] : counter) {

            if(cnt != degree) continue;

            ret = min(ret, max\_idx[num] - min\_idx[num] + 1);

        }

        return ret;

    }

};

// 用两次次循环，三个哈希表，搞定

## 594 最长和谐子序列

哈希表，统计一下cnt即可

class Solution {

public:

    int findLHS(vector<int>& nums) {

        unordered\_map<int, int> hash;

        for(auto num : nums) ++hash[num];

        int ret = 0;

        for(const auto& [num, cnt] : hash) {

            if(hash.count(num+1)) {

                ret = max(ret, hash[num+1] + hash[num]);

            }

        }

        return ret;

    }

};

## 287 寻找重复数

标负法：

class Solution {

public:

    int findDuplicate(vector<int>& nums) {

        for(auto num : nums) {

            num = abs(num);

            if(nums[num-1] < 0) {

                return num;

            }

            nums[num-1] = -nums[num-1];

        }

        return 0;

    }

};

双指针法：

<https://leetcode-cn.com/problems/find-the-duplicate-number/solution/287xun-zhao-zhong-fu-shu-by-kirsche/>

## 313 超级丑数

class Solution {

public:

    int nthSuperUglyNumber(int n, vector<int>& primes) {

        vector<int> uglies(n);

        uglies[0] = 1;

        priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> min\_heap;

        int k = primes.size();

        vector<int> last\_factor(n);

        vector<int> idx(k);

        for(int i=0; i<k; ++i) min\_heap.emplace(primes[i], i);

        for(int i = 1; i<n; ++i) {

            tie(uglies[i], last\_factor[i]) = min\_heap.top();

            min\_heap.pop();

            int j = last\_factor[i];

            do ++idx[j]; while(last\_factor[idx[j]] > j);

            min\_heap.emplace(primes[j]\*uglies[idx[j]], j);

        }

        return uglies.back();

    }

};

// 这个题真是一言难尽，log打印如下：

----------- i = 1-------------

uglies[i] = 2

last\_factor[i] = 0

j = 0

idx[j] = 1, last\_factor[idx[j]] = 0, j = 0

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 2 \* 2

4, 0

----------- i = 2-------------

uglies[i] = 4

last\_factor[i] = 0

j = 0

idx[j] = 2, last\_factor[idx[j]] = 0, j = 0

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 2 \* 4

8, 0

----------- i = 3-------------

uglies[i] = 7

last\_factor[i] = 1

j = 1

idx[j] = 1, last\_factor[idx[j]] = 0, j = 1

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 7 \* 2

14, 1

----------- i = 4-------------

uglies[i] = 8

last\_factor[i] = 0

j = 0

idx[j] = 3, last\_factor[idx[j]] = 1, j = 0

idx[j] = 4, last\_factor[idx[j]] = 0, j = 0

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 2 \* 8

16, 0

----------- i = 5-------------

uglies[i] = 13

last\_factor[i] = 2

j = 2

idx[j] = 1, last\_factor[idx[j]] = 0, j = 2

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 13 \* 2

26, 2

----------- i = 6-------------

uglies[i] = 14

last\_factor[i] = 1

j = 1

idx[j] = 2, last\_factor[idx[j]] = 0, j = 1

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 7 \* 4

28, 1

----------- i = 7-------------

uglies[i] = 16

last\_factor[i] = 0

j = 0

idx[j] = 5, last\_factor[idx[j]] = 2, j = 0

idx[j] = 6, last\_factor[idx[j]] = 1, j = 0

idx[j] = 7, last\_factor[idx[j]] = 0, j = 0

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 2 \* 16

32, 0

----------- i = 8-------------

uglies[i] = 19

last\_factor[i] = 3

j = 3

idx[j] = 1, last\_factor[idx[j]] = 0, j = 3

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 19 \* 2

38, 3

----------- i = 9-------------

uglies[i] = 26

last\_factor[i] = 2

j = 2

idx[j] = 2, last\_factor[idx[j]] = 0, j = 2

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 13 \* 4

52, 2

----------- i = 10-------------

uglies[i] = 28

last\_factor[i] = 1

j = 1

idx[j] = 3, last\_factor[idx[j]] = 1, j = 1

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 7 \* 7

49, 1

----------- i = 11-------------

uglies[i] = 32

last\_factor[i] = 0

j = 0

idx[j] = 8, last\_factor[idx[j]] = 3, j = 0

idx[j] = 9, last\_factor[idx[j]] = 2, j = 0

idx[j] = 10, last\_factor[idx[j]] = 1, j = 0

idx[j] = 11, last\_factor[idx[j]] = 0, j = 0

min\_heap.emplace : primes[j] \* uglies[idx[j]] : 2 \* 32

64, 0

解释说明：

Last\_factor是丑数index到质数index的映射，以下简称lf

Idx是质数index到丑数index的映射。

开始的时候，把质数和对应的last\_factor存入优先级队列

在每个循环中，从栈顶取出丑数和对应的质数。再从栈底推进去一个元素：

这个元素对应的质数同取出的栈顶相同，关键是推入的丑数怎么计算：

算了，不想写了……

## 870 优势洗牌

class Solution {

public:

    vector<int> advantageCount(vector<int>& A, vector<int>& B) {

        multiset<int> s(A.begin(), A.end());

        vector<int> res;

        // 对于B中的每个元素，在A中要么有upper\_bound，要么找不到对应的，只能对应最小值

        for(int i=0; i<B.size(); ++i) {

            auto p = \*s.rbegin() <= B[i] ? s.begin() : s.upper\_bound(B[i]);

            res.push\_back(\*p);

            s.erase(p);

        }

        return res;

    }

};

## 307 区域和检索 - 数组可修改

线段树类型 - 可以暂时略过了

# 字符串

## 242 有效的字母异位词

class Solution {

public:

    bool isAnagram(string s, string t) {

        if(s.length() != t.length()) return false;

        vector<int> cnt(26, 0);

        for(int i = 0; i<s.size(); ++i) {

            ++cnt[s[i] - 'a'];

            --cnt[t[i] - 'a'];

        }

        for(int i=0; i<26; ++i) {

            if(cnt[i] != 0) return false;

        }

        return true;

    }

};

## 205 同构字符串

class Solution {

public:

    bool isIsomorphic(string s, string t) {

        vector<int> s\_prev\_index(256, 0), t\_prev\_index(256, 0);

        for(int i=0; i<s.size(); ++i) {

            if(s\_prev\_index[s[i]] != t\_prev\_index[t[i]]) return false;

            s\_prev\_index[s[i]] = t\_prev\_index[t[i]] = i+1;

        }

        return true;

    }

};

## 647 回文子串

class Solution {

public:

    int countSubstrings(string s) {

        int num = 0;

        for(int i=0; i<s.size(); ++i) {

            num += extendSubStrings(s, i, i);

            num += extendSubStrings(s, i, i+1);

        }

        return num;

    }

    int extendSubStrings(const string& s, int l, int r) {

        int count = 0;

        while(l>=0 && r<s.length() && s[l] == s[r]) {

            --l;

            ++r;

            ++count;

        }

        return count;

    }

};

## 696 计数二进制子串

自己写的，从中间往外扩散：

class Solution {

public:

    int countBinarySubstrings(string s) {

        int num = 0;

        for(int i=1; i<s.size(); ++i) {

            if(s[i] != s[i-1])

                num += helper(s, i-1, i);

        }

        return num;

    }

    int helper(const string& s, int l, int r) {

        int cnt = 0;

        do{

            --l;

            ++r;

            ++cnt;

        } while(l>=0 && r<s.size() && s[l] == s[l+1] && s[r] == s[r-1]);

        return cnt;

    }

};

101的写法：效率高，一次循环搞定

class Solution {

public:

    int countBinarySubstrings(string s) {

        int prev = 0, cur = 1, count = 0;

        for(int i = 1; i<s.size(); ++i) {

            if(s[i] == s[i-1]) ++cur;

            else {

                prev = cur;

                cur = 1;

            }

            if(prev >= cur) ++count;

        }

        return count;

    }

};

## 28 strStr

经典的子串查找算法KMP，暂时略过

## 409 最长回文串

class Solution {

public:

    int longestPalindrome(string s) {

        vector<bool> used(52, false);

        int len = 0;

        int pos;

        for(auto c : s) {

            if(c >= 'A' && c <= 'Z') {

                pos = c-'A'+26;

            }

            else {

                pos = c-'a';

            }

            if(used[pos])  len += 2;

            used[pos] = !used[pos];

        }

        for(auto u : used) if(u) return len+1;

        return len;

    }

};

/\*

思路先行：

哈希表的思路：vector<bool> 用于统计某个字符是否处于待配对状态

遍历auto c : s

如果该字符带配对中，则长度加2

配对状态要反置

遍历结束

结尾的时候，如果仍有待配对的字符，则可放在中间

\*/

# 链表

## 24 两两交换链表中的节点

class Solution {

public:

    ListNode\* swapPairs(ListNode\* head) {

        ListNode \*p = head, \*s;

        if(p && p->next) {

            s = p->next;

            p->next = s->next;

            s->next = p;

            head = s;

            while(p->next && p->next->next) {

                s = p->next->next;

                p->next->next = s->next;

                s->next = p->next;

                p->next = s;

                p = s->next;

            }

        }

        return head;

    }

};

/\*

迭代解法：

 因为要返回队头，所以头两个元素的处理单独拿出来

 仔细即可

\*/

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

class Solution {

    // 反转左闭右开区间

    ListNode\* reverse(ListNode\* a, ListNode\* b) {

        ListNode \*prev = nullptr, \*cur = a, \*next;

        while(cur != b) {

            next = cur->next;

            cur->next = prev;

            prev = cur;

            cur = next;

        }

        return prev;

    }

public:

    ListNode\* swapPairs(ListNode\* head) {

        if(!head || !head->next) return head;

        ListNode\* a = head, \*b = head->next->next;

        ListNode\* new\_head = reverse(a, b);

        a->next = swapPairs(b);

        return new\_head;

    }

};

/\*

递归解法：

\*/

## 21 合并两个有序链表

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

class Solution {

public:

    ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

        ListNode\* dummy = new ListNode(0), \*node = dummy;

        while(l1 && l2) {

            if(l1->val < l2->val) {

                node->next = l1;

                l1 = l1->next;

            }

            else {

                node->next = l2;

                l2 = l2->next;

            }

            node = node->next;

        }

        if(l1) node->next = l1;

        if(l2) node->next = l2;

        return dummy->next;

    }

};

/\*

迭代解法：

\*/

class Solution {

public:

    ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

        if(!l1) return l2;

        if(!l2) return l1;

        if(l1->val < l2->val) {

            l1->next = mergeTwoLists(l1->next, l2);

            return l1;

        }

        l2->next = mergeTwoLists(l1, l2->next);

        return l2;

    }

};

/\*

递归解法：

\*/

## 328 奇偶链表

自己写的代码，总感觉不是很利索

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

class Solution {

public:

    ListNode\* oddEvenList(ListNode\* head) {

        if(!head || !head->next) return head;

        ListNode\* even\_start = head->next;

        ListNode\* cur = head, \*next = head;

        bool is\_cur\_odd = true;

        while(cur->next && cur->next->next) {

            next = cur->next;

            cur->next = cur->next->next;

            cur = next;

            is\_cur\_odd = !is\_cur\_odd;

        }

        if(is\_cur\_odd) cur->next = even\_start;

        else {

            cur->next->next = even\_start;

            cur->next = nullptr;

        }

        return head;

    }

};

更好的方法是：对这种成对的题，两个放一起循环，写出的效果更好。

class Solution {

public:

    ListNode\* oddEvenList(ListNode\* head) {

        if(!head) return head;

        ListNode \*odd = head, \*even = head->next, \*even\_head = even;

        while(even && even->next) {

            odd->next = odd->next->next;

            even->next = even->next->next;

            odd = odd->next;

            even = even->next;

        }

        odd->next = even\_head;

        return head;

    }

};

## 19 删除链表的倒数第 N 个结点

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

class Solution {

public:

    ListNode\* removeNthFromEnd(ListNode\* head, int n) {

        ListNode\* dummy = new ListNode(0);

        dummy->next = head;

        ListNode \*slow = dummy, \*fast = dummy;

        for(int i=0; i<=n; ++i) fast = fast->next;

        while(fast) {

            fast = fast->next;

            slow = slow->next;

        }

        ListNode\* temp = slow->next;

        slow->next = slow->next->next;

        delete temp;

        return dummy->next;

    }

};

这里要注意的是在前面加一个dummy，防止head被删掉导致的错误

## 148 排序链表

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

class Solution {

    ListNode\* merge(ListNode\* l1, ListNode\* l2) {

        ListNode\* dummy = new ListNode(0), \*node = dummy;

        while(l1 && l2) {

            if(l1->val < l2->val) {

                node->next = l1;

                l1 = l1->next;

            }

            else {

                node->next = l2;

                l2 = l2->next;

            }

            node = node->next;

        }

        node->next = l1 ? l1 : l2;

        return dummy->next;

    }

public:

    ListNode\* sortList(ListNode\* head) {

        if(!head || !head->next) return head;

        ListNode \*fast = head->next, \*slow = head;

        while(fast && fast->next) {

            fast = fast->next->next;

            slow = slow->next;

        }

        fast = slow->next;

        slow->next = nullptr;

        return merge(sortList(head), sortList(fast));

    }

};

这个题有个语句很容易出错，就是初始化的时候,fast = head->next，而不是head。这个以后要研究一下

# 树

## 110 平衡二叉树

class Solution {

public:

    bool isBalanced(TreeNode\* root) {

        return helper(root) != -1;

    }

    int helper(TreeNode\* root) {

        if(!root) return 0;

        int left = helper(root->left), right = helper(root->right);

        if(left == -1 || right == -1 || abs(left-right) > 1) {

            return -1;

        }

        return 1 + max(left, right);

    }

};

值得一看，也没那么easy

## 543 二叉树的直径

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

class Solution {

public:

    int diameterOfBinaryTree(TreeNode\* root) {

        int diameter = 0;

        helper(root, diameter);

        return diameter;

    }

    // helper的定义很重要：

    // 对于以root为根节点的树，记录其最长半径为diameter，返回单侧最大节点数（供父节点使用）

    // 这里可以看出，引用传参适合持续维护更新的变量，返回值适合一锤子买卖

    int helper(TreeNode\* root, int& diameter) {

        if(!root) return 0;

        int left = helper(root->left, diameter);

        int right = helper(root->right, diameter);

        diameter = max(diameter, left + right);

        return max(left, right) + 1;

    }

};

这个题不简单

## 113 路径总和II

自己写的写法：

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode() : val(0), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

 \* };

 \*/

class Solution {

public:

    vector<vector<int>> pathSum(TreeNode\* root, int targetSum) {

        vector<int> path;

        if(root) path.push\_back(root->val);

        vector<vector<int>> ans;

        helper(root, targetSum, path, ans);

        return ans;

    }

    void helper(TreeNode\* root, int targetSum, vector<int>& path, vector<vector<int>>& ans) {

        if(!root) return;

        if(!root->left && !root->right) {

            if(root->val == targetSum) {

                ans.push\_back(path);

            }

        }

        if(root->left) {

            path.push\_back(root->left->val);

            helper(root->left, targetSum-root->val, path, ans);

            path.pop\_back();

        }

        if(root->right) {

            path.push\_back(root->right->val);

            helper(root->right, targetSum-root->val, path, ans);

            path.pop\_back();

        }

    }

};

另一种写法，遍历时在left和right的外面进行当前节点的push和pop：

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode() : val(0), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

 \* };

 \*/

class Solution {

public:

    vector<vector<int>> pathSum(TreeNode\* root, int targetSum) {

        vector<int> path;

        vector<vector<int>> ans;

        if(!root) return ans;

        helper(root, targetSum, path, ans);

        return ans;

    }

    void helper(TreeNode\* root, int targetSum, vector<int>& path, vector<vector<int>>& ans) {

        path.push\_back(root->val);

        if(!root->left && !root->right) {

            if(root->val == targetSum) {

                ans.push\_back(path);

            }

        }

        if(root->left) {

            helper(root->left, targetSum-root->val, path, ans);

        }

        if(root->right) {

            helper(root->right, targetSum-root->val, path, ans);

        }

        path.pop\_back();

    }

};

## 437 路径总和III

这其实是一个非常非常经典的递归题目，非常好

pathSum的定义：给定一个根节点和目标，求路径数目；

它等于1、左右节点的pathSum，2、再加上以根节点为节点的路径数

1是pathSum的子问题  2、是一个可以拿来单独递归的问题

\*/

class Solution {

public:

    int pathSum(TreeNode\* root, int sum) {

        // TODO base case

        if(!root) return 0;

        return pathSum(root->left, sum) + pathSum(root->right, sum) + pathSumStartWithRoot(root, sum);

    }

    int pathSumStartWithRoot(TreeNode\* root, int sum) {

        if(!root) return 0;

        int count = root->val == sum ? 1 : 0;

        count += pathSumStartWithRoot(root->left, sum-root->val);

        count += pathSumStartWithRoot(root->right, sum-root->val);

        return count;

    }

};

## 1110 删点成林

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

class Solution {

public:

    vector<TreeNode\*> delNodes(TreeNode\* root, vector<int>& to\_delete) {

        vector<TreeNode\*> forest;

        if(!root) return forest;

        unordered\_set<int> dict(to\_delete.begin(), to\_delete.end());

        root = helper(root, dict, forest);

        if(root) forest.push\_back(root);

        return forest;

    }

    TreeNode\* helper(TreeNode\* root, unordered\_set<int>& dict, vector<TreeNode\*>& forest) {

        if(!root) return nullptr;

        root->left = helper(root->left, dict, forest);

        root->right = helper(root->right, dict, forest);

        if(dict.count(root->val)) {

            if(root->left) {

                forest.push\_back(root->left);

            }

            if(root->right) {

                forest.push\_back(root->right);

            }

            // delete(root);

            root = nullptr;

        }

        return root;

    }

};

/\*

实战化解题：

把视角放到每一个节点

不难看出，很关键的一点是：这个题只能从叶子节点往上来删。所以，果断采用后续遍历

后续遍历就是不管三七二十一，先调用子节点的函数，最后处理根节点

搞定

注意：

1、有一个点还要清楚，如果最后总的根节点不删的话，它也要算作森林里的一颗树，放到结果里面

2、这种修剪或改动树的情况下，最好在返回树的节点

\*/

## 637 二叉树的层序遍历

class Solution {

public:

    vector<double> averageOfLevels(TreeNode\* root) {

        vector<double> ans;

        if(!root) return ans;

        queue<TreeNode\*> q;

        q.push(root);

        while(!q.empty()) {

            int sz = q.size();

            double sum = 0;

            for(int i=0; i<sz; ++i) {

                TreeNode\* temp = q.front();

                q.pop();

                sum += temp->val;

                if(temp->left) q.push(temp->left);

                if(temp->right) q.push(temp->right);

            }

            ans.push\_back(sum/sz);

        }

        return ans;

    }

};

一遍就写出来了，是真的easy

## 99 恢复二叉搜索树

class Solution {

public:

    void recoverTree(TreeNode\* root) {

        TreeNode \*mistake1 = nullptr, \*mistake2 = nullptr, \*prev = nullptr;

        inorder(root, mistake1, mistake2, prev);

        if(mistake1 && mistake2) {

            int temp = mistake1->val;

            mistake1->val = mistake2->val;

            mistake2->val = temp;

        }

    }

    void inorder (TreeNode\* root, TreeNode\*& mistake1, TreeNode\*& mistake2, TreeNode\*& prev) {

        if(!root) return;

        inorder(root->left, mistake1, mistake2, prev);

        if(prev && prev->val > root->val) {

            if(!mistake1) {

                mistake1 = prev;

                mistake2 = root;

            }

            else {

                mistake2 = root;

            }

        }

        prev = root;

        inorder(root->right, mistake1, mistake2, prev);

    }

};

/\*

中序遍历，技巧是：

 1 用prev记录之前节点

 2 如果只出现一次逆序，则是相邻节点错误；如果出现两次逆序，则是交换非相邻节点

\*/

这道题不算简单，一定要好好看

## 669 修剪二叉搜索树

class Solution {

public:

    TreeNode\* trimBST(TreeNode\* root, int low, int high) {

        if(!root) return nullptr;

        if(root->val < low) {

            return trimBST(root->right, low, high);

        }

        if(root->val > high) {

            return trimBST(root->left, low, high);

        }

        root->left = trimBST(root->left, low, high);

        root->right = trimBST(root->right, low, high);

        return root;

    }

};

/\*

这个题不用想的太复杂，还是把视线转移到当前的一个根节点上来。

 如果根节点的值太小，小于low，那么根节点和它的左子树都要剪掉，递归右子树并返回，即可

 如果根节点的值太大，同理，递归调用左子树并返回。

 如果根节点值符合要求，那么就要开始构造它的左右子树，也是递归调用，最后返回root

 完事

\*/

## 208 实现Trie（字典树、前缀树）

class TrieNode {

public:

    TrieNode\* child[26];

    bool is\_vaild;

    TrieNode() : is\_vaild(false) {

        for(int i=0; i<26; ++i) {

            child[i] = nullptr;

        }

    }

};

class Trie {

private:

    TrieNode\* root;

public:

    /\*\* Initialize your data structure here. \*/

    Trie() : root(new TrieNode) {

    }

    /\*\* Inserts a word into the trie. \*/

    void insert(string word) {

        TrieNode\* temp = root;

        for(auto c : word) {

            if(!temp->child[c-'a']) {

                temp->child[c-'a'] = new TrieNode;

            }

            temp = temp->child[c-'a'];

        }

        temp->is\_vaild = true;

    }

    /\*\* Returns if the word is in the trie. \*/

    bool search(string word) {

        TrieNode\* temp = root;

        for(auto c : word) {

            if(!temp) break;

            temp = temp->child[c-'a'];

        }

        return temp ? temp->is\_vaild : false;

    }

    /\*\* Returns if there is any word in the trie that starts with the given prefix. \*/

    bool startsWith(string prefix) {

        TrieNode\* temp = root;

        for(auto c : prefix) {

            if(!temp) break;

            temp = temp->child[c-'a'];

        }

        return temp;

    }

};

/\*\*

 \* Your Trie object will be instantiated and called as such:

 \* Trie\* obj = new Trie();

 \* obj->insert(word);

 \* bool param\_2 = obj->search(word);

 \* bool param\_3 = obj->startsWith(prefix);

 \*/

基本是抄的答案，哈哈，不难，都是一些细节。

## 617 合并二叉树

用于练手的简单题

class Solution {

public:

    TreeNode\* mergeTrees(TreeNode\* root1, TreeNode\* root2) {

        if(!root1 && !root2) return nullptr;

        if(!root1) return root2;

        if(!root2) return root1;

        root1->val += root2->val;

        root1->left = mergeTrees(root1->left, root2->left);

        root1->right = mergeTrees(root1->right, root2->right);

        return root1;

    }

};

## 572 另一棵树的子树

这是一个很好的题，真的很好，而且并不简单！

class Solution {

    bool compare(TreeNode\* a, TreeNode\* b) {

        if(!a && !b) return true;

        if(!a || !b) return false;

        return a->val == b->val && compare(a->left, b->left) && compare(a->right, b->right);

    }

public:

    bool isSubtree(TreeNode\* s, TreeNode\* t) {

        if(!s) return !t;

        return isSubtree(s->left, t) || isSubtree(s->right, t) || compare(s,t);

    }

};

思路是定义一个辅助函数compare！！

## 404 左叶子之和

class Solution {

public:

    int sumOfLeftLeaves(TreeNode\* root) {

        int sum = 0;

        traverse(root, false, sum);

        return sum;

    }

    void traverse(TreeNode\* root, bool is\_left\_subtree, int& sum) {

        if(!root) return;

        if(!root->left && !root->right && is\_left\_subtree) {

            sum += root->val;

        }

        traverse(root->left, true, sum);

        traverse(root->right, false, sum);

    }

};

这个题自己一遍就写对了，就是一个遍历加上状态传参和维护传参

## 513 找数左下角的值

两种方法：1层次遍历广度优先 2深度优先

先来层次遍历：

class Solution {

public:

    int findBottomLeftValue(TreeNode\* root) {

        queue<TreeNode\*> q;

        q.push(root);

        int ret = root->val;

        while(!q.empty()) {

            int sz = q.size();

            for(int i=0; i<sz; ++i) {

                TreeNode\* node = q.front();

                q.pop();

                if(i == 0) ret = node->val;

                if(node->left) q.push(node->left);

                if(node->right) q.push(node->right);

            }

        }

        return ret;

    }

};

或者换一种从右往左的层次遍历，遍历到最后就是想要的值：

class Solution {

public:

    int findBottomLeftValue(TreeNode\* root) {

        queue<TreeNode\*> q;

        q.push(root);

        while (!q.empty()) {

            root = q.front();

            q.pop();

            if (root->right) q.push(root->right);

            if (root->left) q.push(root->left);

        }

        return root->val;

    }

};

深度优先写法：

class Solution {

public:

    int findBottomLeftValue(TreeNode\* root) {

        int max\_depth = 0;

        int leftdown = root->val;

        traverse(root, 1, max\_depth, leftdown);

        return leftdown;

    }

    void traverse(TreeNode\* root, int depth, int& max\_depth, int& leftdown) {

        if(!root) return;

        if(depth > max\_depth) {

            max\_depth = depth;

            if(!root->left && !root->right) {

                leftdown = root->val;

                return;

            }

        }

        traverse(root->left, depth+1, max\_depth, leftdown);

        traverse(root->right, depth+1, max\_depth, leftdown);

    }

};

## 235 二叉搜索树的最近公共祖先

class Solution {

public:

    TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

        if(!root) return nullptr;

        if(max(p->val, q->val) < root->val) {

            return lowestCommonAncestor(root->left, p, q);

        }

        if(min(p->val, q->val) > root->val) {

            return lowestCommonAncestor(root->right, p, q);

        }

        return root;

    }

};

看似简单，但是如果没有递归思维的话，想不出来的

## 530 二叉搜索树的最小绝对差

class Solution {

public:

    int getMinimumDifference(TreeNode\* root) {

        int min\_diff = INT\_MAX;

        TreeNode\* prev = nullptr;

        inorder(root, prev, min\_diff);

        return min\_diff;

    }

    void inorder(TreeNode\* root, TreeNode\*& prev, int& min\_diff) {

        if(!root) return;

        inorder(root->left, prev, min\_diff);

        if(prev && root->val - prev->val < min\_diff) {

            min\_diff = root->val - prev->val;

        }

        prev = root;

        inorder(root->right, prev, min\_diff);

    }

};

## 889 根据前序遍历和后序遍历构造二叉树

先上自己写的答案：

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

class Solution {

public:

    TreeNode\* constructFromPrePost(vector<int>& pre, vector<int>& post) {

        int n = pre.size();

        return build(pre, 0, n-1, post, 0, n-1);

    }

    TreeNode\* build(vector<int>& pre, int pre\_l, int pre\_r,

               vector<int>& post, int post\_l, int post\_r) {

        if(pre\_l > pre\_r || post\_l > post\_r) return nullptr;

        TreeNode\* root = new TreeNode(pre[pre\_l]);

        if(pre\_l == pre\_r || post\_l == post\_r) return root;

        int idx\_left\_child = find(post.begin(), post.end(), pre[pre\_l+1]) - post.begin();

        int left\_size = idx\_left\_child - post\_l + 1;

        int right\_size = pre.size() - 1 - left\_size;

        root->left = build(pre, pre\_l+1, pre\_l + 1 + left\_size - 1, post, post\_l, post\_l + left\_size - 1);

        root->right = build(pre, pre\_l + 1 + left\_size, pre\_r, post, post\_l + left\_size, post\_r-1);

        return root;

    }

};

## 109 有序链表转二叉搜索树

class Solution {

public:

    TreeNode\* sortedListToBST(ListNode\* head) {

        if(!head) return nullptr;

        if(!head->next) return new TreeNode(head->val);

        ListNode \*slow = head, \*fast = head->next, \*preslow = head;

        while(fast && fast->next) {

            fast = fast->next->next;

            preslow = slow;

            slow = slow ->next;

        }

        TreeNode\* root = new TreeNode(slow->val);

        if(slow != preslow) {

            root->right = sortedListToBST(slow->next);

            preslow->next = nullptr;

            root->left = sortedListToBST(head);

        }

        else {

            root->right = sortedListToBST(slow->next);

        }

        return root;

    }

};

/\*

这个题需要考虑的细节是链表中点的停止条件，以及preslow的使用

\*/

## 897 递增顺序查找树

class Solution {

public:

    TreeNode\* increasingBST(TreeNode\* root) {

        TreeNode\* head = new TreeNode(-1), \*prev = head;

        head->right = root;

        inorder(root, prev);

        return head->right;

    }

    void inorder(TreeNode\* root, TreeNode\*& prev) {

        if(!root) return;

        inorder(root->left, prev);

        prev->right = root;

        prev = prev->right;

        prev->left = nullptr;

        inorder(root->right, prev);

    }

};

注意inorder里面的顺序。这个题我自己写，思路对了，但细节没对。

这个有点像设置了dummy的链表

## 653 两数之和 IV - 输入 BST

class Solution {

public:

    bool findTarget(TreeNode\* root, int k) {

        vector<int> nums;

        inorder(root, nums);

        int i=0, j=nums.size()-1;

        while(i<j) {

            int sum = nums[i] + nums[j];

            if(sum == k) return true;

            if(sum < k) ++i;

            else --j;

        }

        return false;

    }

    void inorder(TreeNode\* root, vector<int>& nums) {

        if(!root) return;

        inorder(root->left, nums);

        nums.push\_back(root->val);

        inorder(root->right, nums);

    }

};

这个题有点没意思了

## 450 删除二叉搜索树中的节点

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode() : val(0), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

 \* };

 \*/

class Solution {

public:

    // 定义先行：对于以root为根节点的树，删除值为key的节点，返回修改后的树

    TreeNode\* deleteNode(TreeNode\* root, int key) {

        if(!root) return nullptr;

        if(root->val > key) root->left = deleteNode(root->left, key);

        else if(root->val < key) root->right = deleteNode(root->right, key);

        else {

            if(!root->left && !root->right) {

                delete(root);

                root = nullptr;

            }

            else if(!root->left || !root->right) {

                TreeNode\* new\_root = root->left ? root->left : root->right;

                delete(root);

                root = new\_root;

            }

            else if(root->left && root->right) {

                TreeNode\* tmp = root->right;

                while(tmp->left) {

                    tmp = tmp->left;

                }

                root->val = tmp->val;

                root->right = deleteNode(root->right, tmp->val);

            }

        }

        return root;

    }

};

这个题很经典很经典，关于如何删除二叉搜索树中的节点！！

# 图

## 1135 最低成本联通所有城市

接下来要掌握一下leetcode的1135题

1. kruskal + 并查集解法：自己写的

class UF {

private:

    int count\_;

    vector<int> parent\_;

    vector<int> size\_;

    int find(int x) {

        while(x != parent\_[x]) {

            parent\_[x] = parent\_[parent\_[x]];

            x = parent\_[x];

        }

        return x;

    }

public:

    UF(int n) {

        count\_ = n;

        parent\_.resize(n);

        size\_.resize(n);

        for(int i=0; i<n; ++i) {

            parent\_[i] = i;

            size\_[i] = 1;

        }

    }

    int count() {return count\_;}

    void nodeUnion(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        if(root\_p == root\_q) return;

        if(size\_[p] > size\_[q]) {

            parent\_[root\_q] = root\_p;

            size\_[p] += size\_[q];

        }

        else {

            parent\_[root\_p] = root\_q;

            size\_[q] += size\_[p];

        }

        --count\_;

    }

    bool isConnected(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        return root\_p == root\_q;

    }

};

class Solution {

public:

    int minimumCost(int N, vector<vector<int>>& connections) {

        vector<vector<int>> ret\_edges;

        UF uf(N+1);

        sort(connections.begin(), connections.end(),

            [](const vector<int>& a, const vector<int>& b) {

            return a[2] < b[2];

        });

        int n = connections.size();

        for(int i=0; i<n && ret\_edges.size() < N-1; ++i) {

            int b = connections[i][0];

            int e = connections[i][1];

            if(!uf.isConnected(b, e)) {

                ret\_edges.push\_back(connections[i]);

                uf.nodeUnion(b, e);

            }

        }

        if(ret\_edges.size() != N-1) return -1;

        int ret = 0;

        for(const auto& edge : ret\_edges) {

            ret += edge[2];

        }

        return ret;

    }

};

1. prim写法， 来自101，优先级队列，但是不能输出路径

class Solution {

public:

    int minimumCost(int N, vector<vector<int>>& connections) {

        // 基本数据结构pair<int,int> - 权重， 后继， 基于这个结构建立查询表graph和优先级队列

        vector<vector<pair<int, int>>> graph(N, vector<pair<int,int>>());

        for(const auto& conn : connections) {

            graph[conn[0]-1].push\_back(make\_pair(conn[2], conn[1]-1));

            graph[conn[1]-1].push\_back(make\_pair(conn[2], conn[0]-1));

        }

        priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

        unordered\_set<int> visited;

        int costs = 0;

        // 小技巧， 在进入循环以前，优先级队列先推入{0, 0}，这样就可以把所有操作统一放到循环里面

        pq.emplace(0,0);

        while(!pq.empty()) {

            auto [cost, city2] = pq.top(); pq.pop();

            if(!visited.count(city2)) {

                costs += cost;

                visited.insert(city2);

                for(const auto& c : graph[city2]) pq.push(c);

            }

        }

        return N == visited.size() ? costs : -1;

    }

};

## 882 细分图中的可到达节点

/\*

 这道hard题的整体思路就浮出水面了

 它是利用Dijkstra算法遍历以求得可到达节点的最短路径，同时在这个过程中记录更新used数组。

 这种就是在遍历框架中加些小操作的问题

跟普通的Dijkstra算法的区别在于：1、没有设置V表和S表；2、遍历中统计相邻节点关系

 \*/

class Solution {

public:

    using pii = pair<int,int>;

    // maxMoves - 移动距离上限     n - 节点数

    int reachableNodes(vector<vector<int>>& edges, int maxMoves, int n) {

        int ans = 0;

        // 后继表示法表示的graph, pii-first后继节点 pii-second:权重

        vector<vector<pii>> graph(n);

        for(auto& edge : edges) {

            int u = edge[0], v = edge[1], w = edge[2];

            graph[u].push\_back({v, w});

            graph[v].push\_back({u, w});

        }

        // dist数组 : 从起点0到起点i的最短距离

        vector<int> dist(n, maxMoves+1);// 最多走M步，所以M+1有无穷大的意义

        dist[0] = 0;

        // used数组 : used[{i,j}] = u  表示从起点i到终点j经历了u个小节点

        map<pii, int> used;

        // 优先级队列 待选择列表 first-权重 second-该边对应的节点

        priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

        pq.emplace(0,0);

        while(!pq.empty()) {

            pii top = pq.top(); pq.pop();

            int d = top.first, node = top.second;

            if(dist[node] < d) continue; // 疑点1：为什么这样做？为什么等于不continue

            dist[node] = d;

            ++ans;

            // 和邻居的关系只能在下面这段代码来体现

            for(auto pair : graph[node]) {

                int nei = pair.first;

                int weight = pair.second;

                used[{node, nei}] = min(weight, maxMoves-d);

                int d2 = d + 1 + weight;

                if(d2 < dist[nei]) {

                    dist[nei] = d2;

                    pq.emplace(d2, nei);

                }

            }

        }

        for(auto edge : edges) {

            int u = edge[0], v = edge[1], w = edge[2];

            ans += min(used[{u,v}] + used[{v,u}], w);

        }

        return ans;

    }

};

## 785 判断二分图

广度优先搜索

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        vector<int> color(n);

        queue<int> q;

        for(int i=0; i<n; ++i) {

            if(color[i] == 0) {

                color[i] = 1;

                q.push(i);

                while(!q.empty()) {

                    int node = q.front(); q.pop();

                    for(auto adj : graph[node]) {

                        if(color[adj] == 0) {

                            color[adj] = color[node] == 2 ? 1 : 2;

                            q.push(adj);

                        }

                        else {

                            if(color[adj] == color[node]) {

                                return false;

                            }

                        }

                    }

                }

            }

        }

        return true;

    }

};

/\*

思路：遍历所有节点，对每一个未染色的节点进行广度优先搜索，搜索未染色节点进行染色，并在邻居处判断是否出现错误。

也就是：一遍遍历涂色，一遍检查错误

\*/

下面是深度优先解法：

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        vector<int> color(n,0);

        for(int i=0; i<n; ++i) {

            if(color[i] == 0) {

                color[i] = 1;

                if(!dfs(graph, i, color)) return false;

            }

        }

        return true;

    }

    bool dfs(vector<vector<int>>& graph, int node, vector<int>& color) {

        for(auto adj : graph[node]) {

            if(color[adj] == 0) {

                color[adj] = color[node] == 2 ? 1 : 2;

                if(!dfs(graph, adj, color)) return false;

            }

            else {

                if(color[adj] == color[node]) {

                    return false;

                }

            }

        }

        return true;

    }

};

/\*

思路：遍历所有节点，对每一个未染色的节点进行深度优先搜索，搜索未染色节点进行染色，并在邻居处判断是否出现错误。

也就是：一遍遍历涂色，一遍检查错误

\*/

上面用的是先赋值再递归的方法，把判断放在当前节点和邻居节点的关系上；其实还有另一种方法，就是直接递归，把判断放在当前节点的值是否会出现冲突上——就是说，我给你设置一个值作为参数，看你会不会自相矛盾。链接：<https://leetcode-cn.com/problems/is-graph-bipartite/solution/bfs-dfs-bing-cha-ji-san-chong-fang-fa-pan-duan-er-/>

代码如下：

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        vector<int> color(n,0);

        for(int i=0; i<n; ++i) {

            if(color[i] == 0) {

                if(!dfs(graph, i, 1, color)) return false;

            }

        }

        return true;

    }

    bool dfs(vector<vector<int>>& graph, int node, int given\_color, vector<int>& color) {

        if(color[node] != 0) {

            return given\_color == color[node];

        }

        color[node] = given\_color;

        for(auto adj : graph[node]) {

            if(!dfs(graph, adj, -given\_color, color)) return false;

        }

        return true;

    }

};

还有第三种方法：并查集

class UF {

private:

    int count\_;

    vector<int> parent\_;

    vector<int> size\_;

    int find(int x) {

        while(parent\_[x] != x) {

            parent\_[x] = parent\_[parent\_[x]];

            x = parent\_[x];

        }

        return x;

    }

public:

    UF(int n) {

        count\_ = n;

        parent\_.resize(n);

        size\_.resize(n);

        for(int i=0; i<n; ++i) {

            parent\_[i] = i;

            size\_[i] = 1;

        }

    }

    void nodeUnion(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        if(root\_p == root\_q) return;

        if(size\_[root\_p] > size\_[root\_q]) {

            parent\_[root\_q] = root\_p;

            size\_[root\_p] += size\_[root\_q];

        }

        else {

            parent\_[root\_p] = root\_q;

            size\_[root\_q] += size\_[root\_p];

        }

        --count\_;

    }

    bool isConnected(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        return (root\_p == root\_q);

    }

    int count() {

        return count\_;

    }

};

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        UF uf(n);

        for(int i=0; i<n; ++i) {

            auto& adjs = graph[i];

            for(auto adj : adjs) {

                if(uf.isConnected(i, adj)) {

                    return false;   // 如果当前节点已经跟邻居一个集合（颜色）了，则错误

                }

                uf.nodeUnion(adjs[0], adj);

            }

        }

        return true;

    }

};

/\*

并查集写法，把每个节点的后继节点都联通起来，并且判断当前节点是否已经和邻居“勾结“

\*/

## 210 课程表II

拓扑排序类问题，步骤如下：

1. 建立邻接表，统计入度
2. 广度优先入度为0的节点。注意在找邻居的时候入度--。表示该节点已删除。

class Solution {

public:

    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {

        vector<vector<int>> graph(numCourses);

        vector<int> indgree(numCourses, 0);

        vector<int> res;

        for(const auto& prerequisite : prerequisites) {

            graph[prerequisite[1]].push\_back(prerequisite[0]);

            ++indgree[prerequisite[0]];

        }

        queue<int> q;

        for(int i=0; i<numCourses; ++i) {

            if(indgree[i] == 0) {

                q.push(i);

            }

        }

        while(!q.empty()) {

            int u = q.front();

            q.pop();

            res.push\_back(u);

            for(auto v : graph[u]) {

                --indgree[v];

                if(indgree[v] == 0) {

                    q.push(v);

                }

            }

        }

        for(auto x : indgree) {

            if(x!=0) return vector<int>();

        }

        return res;

    }

};

## 1059 从始点到终点的所有路径

class Solution {

public:

    bool leadsToDestination(int n, vector<vector<int>>& edges, int source, int destination) {

        vector<vector<int>> graph(n);

        for(auto edge : edges) {

            graph[edge[0]].push\_back(edge[1]);

        }

        vector<bool> visited(n, false);

        return dfs(graph, source, destination, visited);

    }

    // 从start节点出发能否所有终点都到达target

    bool dfs(vector<vector<int>>& graph, int start, int destination, vector<bool>& visited) {

        if(visited[start]) return false;

        if(graph[start].empty() && start != destination) return false;

        visited[start] = true;

        for(auto adj : graph[start]) {

            if(!dfs(graph, adj, destination, visited)) return false;

        }

        visited[start] = false; // 神奇，加了这句话就对了

        return true;

    }

};

/\*

这个题我写的很好

难点在于对环路的处理，对于这种bool类型的递归或者回溯，一个比较好用的办法就是找true的情形或者false的情形进行排除

我就是挑出了所有false的情形。

同时回溯必不可少，因为要寻找多条路，所以visited既要拿来判断环路，又可以用来回溯。

\*/

# 更复杂数据结构

class UF {

private:

    int count\_;

    vector<int> parent\_;

    vector<int> size\_;

    int find(int x) {

        while(x != parent\_[x]) {

            parent\_[x] = parent\_[parent\_[x]];

            x = parent\_[x];

        }

        return x;

    }

public:

    UF(int n) {

        count\_ = n;

        parent\_.resize(n);

        size\_.resize(n);

        for(int i=0; i<n; ++i) {

            size\_[i] = 1;

            parent\_[i] = i;

        }

    }

    void nodeUnion(int p, int q) {

        int root\_p = find(p), root\_q = find(q);

        if(root\_p == root\_q) return;

        if(size\_[root\_p] > size\_[root\_q]) {

            parent\_[root\_q] = root\_p;

            size\_[root\_p] += size\_[root\_q];

        }

        else {

            parent\_[root\_p] = root\_q;

            size\_[root\_q] += size\_[root\_p];

        }

        --count\_;

    }

    bool isConnected(int p, int q) {

        int root\_p = find(p), root\_q = find(q);

        return (root\_p == root\_q);

    }

    int count() {return count\_;}

};

class Solution {

public:

    vector<int> findRedundantConnection(vector<vector<int>>& edges) {

        int n = edges.size();

        UF uf(n+1);

        for(auto& edge : edges) {

            if(uf.isConnected(edge[0], edge[1])) {

                return vector<int>{edge[0], edge[1]};

            }

            uf.nodeUnion(edge[0], edge[1]);

        }

        return vector<int>();

    }

};

## 432 全 O(1) 的数据结构

class AllOne {

private:

    struct Node {

        int val;

        unordered\_set<string> container;

        Node(int v) : val(v) {}

    };

    unordered\_map<string, list<Node>::iterator> kv;

    list<Node> mylist;

public:

    /\*\* Initialize your data structure here. \*/

    AllOne() {

    }

    /\*\* Inserts a new key <Key> with value 1. Or increments an existing key by 1. \*/

    void inc(string key) {

        if(kv.count(key)) {

            auto old\_node = kv[key];

            auto new\_node = next(old\_node, 1);

            if(new\_node == mylist.end() || new\_node->val > old\_node->val + 1) {

                new\_node = mylist.insert(new\_node, Node(old\_node->val + 1)); // 这里出错啦！！！没有写返回值

            }

            new\_node->container.insert(key);

            old\_node->container.erase(key);

            if(old\_node->container.empty()) {

                mylist.erase(old\_node);

            }

            kv[key] = new\_node;

        }

        else {

            if(mylist.empty() || mylist.front().val > 1) {  // 这里也出过错！！！

                mylist.push\_front(Node(1));

            }

            mylist.front().container.insert(key);

            kv[key] = mylist.begin();

        }

    }

    /\*\* Decrements an existing key by 1. If Key's value is 1, remove it from the data structure. \*/

    void dec(string key) {

        if(!kv.count(key)) return;

        auto old\_node = kv[key];

        if(old\_node->val == 1) {

            kv.erase(key);

        }

        else {

            auto new\_node = next(old\_node, -1);

            if(old\_node == mylist.begin() || new\_node->val < old\_node->val -1) {

                new\_node = mylist.insert(old\_node, Node(old\_node->val - 1));

            }

            new\_node->container.insert(key);

            kv[key] = new\_node;

        }

        old\_node->container.erase(key);

        if(old\_node->container.empty()) {

            mylist.erase(old\_node);

        }

    }

    /\*\* Returns one of the keys with maximal value. \*/

    string getMaxKey() {

        if(mylist.empty()) return "";

        return \*mylist.rbegin()->container.begin();

    }

    /\*\* Returns one of the keys with Minimal value. \*/

    string getMinKey() {

        if(mylist.empty()) return "";

        return \*mylist.begin()->container.begin();

    }

};

/\*\*

 \* Your AllOne object will be instantiated and called as such:

 \* AllOne\* obj = new AllOne();

 \* obj->inc(key);

 \* obj->dec(key);

 \* string param\_3 = obj->getMaxKey();

 \* string param\_4 = obj->getMinKey();

 \*/

 /\*

 这个题非常非常好，一定要自己多手写几遍。跟LRU和LFU一样，属于必须要掌握的设计类题目

 https://leetcode-cn.com/problems/all-oone-data-structure/solution/c-ha-xi-kvshuang-xiang-lian-biao-by-lwpyr/

 \*/

## 716 最大栈

class MaxStack {

    list<int> stack\_;

    map<int, vector<list<int>::iterator>> hash\_;

public:

    /\*\* initialize your data structure here. \*/

    MaxStack() {

    }

    void push(int x) {

        stack\_.push\_front(x);

        hash\_[x].push\_back(stack\_.begin());

    }

    int pop() {

        int x = stack\_.front();

        hash\_[x].pop\_back();

        if(hash\_[x].empty()) {

            hash\_.erase(x);

        }

        stack\_.pop\_front();

        return x;

    }

    int top() {

        return stack\_.front();

    }

    int peekMax() {

        return hash\_.rbegin()->first;

    }

    int popMax() {

        int x = hash\_.rbegin()->first;

        auto iter = hash\_[x].back();

        stack\_.erase(iter);

        hash\_[x].pop\_back();

        if(hash\_[x].empty()) hash\_.erase(x);

        return x;

    }

};

/\*\*

 \* Your MaxStack object will be instantiated and called as such:

 \* MaxStack\* obj = new MaxStack();

 \* obj->push(x);

 \* int param\_2 = obj->pop();

 \* int param\_3 = obj->top();

 \* int param\_4 = obj->peekMax();

 \* int param\_5 = obj->popMax();

 \*/