# 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

# 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

\*/