回溯算法

131.分割回文串（中等）

1. 题目描述

   给定一个字符串 *s*，将 *s* 分割成一些子串，使每个子串都是回文串。

   返回 *s* 所有可能的分割方案。

   示例:

```
输入: "aab"
输出:
[
  ["aa","b"],
  ["a","a","b"]
]
```

2. 简单实现

   剪枝回溯，割就完事了

```
class Solution {
public:
    vector<vector<string>> ans;
    int len;//待分割字符串长度
    bool judge(const string& s){//判断是否为回文串
        int l = 0;
        int r = s.size()-1;
        while(l < r){
            if(s[l] != s[r])
                return false;
            l++;
            r--;
        }
        return true;
    }
    //s[0...idx-1]已经分割完存储在temp中，需要对s[idx...len-1]进行分割
    void split(const string& s, int idx, vector<string>& temp){
        if(idx == len){//确定了一种有效的分割方法
            ans.push_back(temp);
            return;
        }
        for(int i = idx; i < len; i++){//从idx开始遍历，因为单个字符自身也是回文串
            if(s[i]==s[idx]){//只有首尾字符相等才有可能是回文串
                string cur = s.substr(idx, i-idx+1);
                if(judge(cur)){//是回文串，可以分割
                    temp.push_back(cur);
```

```
                    split(s, i+1, temp);
                    temp.pop_back();
                }
            }
        }
    }
    vector<vector<string>> partition(string s) {
        len = s.size();
        if(len == 0) return ans;
        if(len == 1) return {{s}};
        vector<string> temp;
        split(s, 0, temp);
        return ans;
    }
};
```

一个优化可能：空间换时间，可能重复对相同字符串判断是否回文，因此可以记忆化存储一下，或者先动态规划算好

301.删除无效的括号（困难）

1. 题目描述

   删除最小数量的无效括号，使得输入的字符串有效，返回所有可能的结果。

   说明: 输入可能包含了除 `(` 和 `)` 以外的字符。

   示例 1:

```
输入: "()())()"
输出: ["()()()", "(())()"]
```

   示例 2:

```
输入: "(a)())()"
输出: ["(a)()()", "(a())()"]
```

   示例 3:

```
输入: ")("
输出: [""]
```

2. 简单实现

   先获取最少要删除的字符数，再使用回溯算法

```
class Solution {
public:
    int num;//最少要删除的字符数
    int len;//输入字符串长度
    vector<string> ans;
    unordered_set<string> cache;//防止重复字符串进入答案
```

```
    void getNum(const string& str){//获取最少要删除的字符数
        stack<char> s;
        for(int i = 0; i < str.size(); i++){
            if(str[i] == '(')
                s.push(str[i]);
            else if(str[i] == ')'){
                if(s.empty())
                    num++;
                else
                    s.pop();
            }
        }
        num += s.size();
    }
    bool judge(const string& str){//判断是否合规
        stack<char> s;
        for(int i = 0; i < str.size(); i++){
            if(str[i] == '(')
                s.push(str[i]);
            else if(str[i] == ')'){
                if(s.empty())
                    return false;
                else
                    s.pop();
            }
        }
        if(s.empty())
            return true;
        else
            return false;
    }
    //temp保存s[0...idx]的处理结果，其中删去cnt个字符，temp含l个'('，r个')'，还要对
s[idx...]进行处理
    void del(const string& s, int idx, string& temp, int cnt, int l, int r){
        if(l < r) return;//左括号数小于右括号数，一定不合规
        if(cnt == num){//达到去除数量，剩下的字符全都保留
            if(idx < len){//还有剩余字符，全部加入
                string cur = temp + s.substr(idx, len-idx);
                if(cache.count(cur) <= 0 && judge(cur)){
                    ans.push_back(cur);
                    cache.insert(cur);
                }
            }
            else if(cache.count(temp) <= 0 && judge(temp)){//无剩余字符
                ans.push_back(temp);
                cache.insert(temp);
            }
        }
        else{//未达到去除数量，继续去除
            if(idx == len) return;
            else{
                if(s[idx] != '(' && s[idx] != ')'){//非括号，必须保留
                    string back = temp;
```

```
                    temp += s[idx];
                    del(s, idx+1, temp, cnt, l, r);
                    temp = back;
                }
                else{//括号
                    del(s, idx+1, temp, cnt+1, l, r);//删除
                    //保留
                    string back = temp;
                    temp += s[idx];
                    if(s[idx] == '(')
                        del(s, idx+1, temp, cnt, l+1, r);
                    else
                        del(s, idx+1, temp, cnt, l, r+1);
                    temp = back;
                }
            }
        }
    }
    vector<string> removeInvalidParentheses(string s) {
        if(s == "") return {""};
        num = 0;
        len = s.size();
        getNum(s);
        if(num > 0){
            if(num == len)
                return {""};
            string temp = "";
            del(s, 0, temp, 0, 0, 0);
            return ans;
        }
        else
            return {s};
    }
};
```

3. BFS

   将给定字符串放入队列，取出队列中元素，如果合法，则加入结果集继续取出队列元素，否则，遍历给定字符串每个字符，如果是左右括号，则去除括号后生成新的字符串，如果新字符串没有在队列中出现过（使用set查看是否出现）则加入队列。

4. 动态规划

解题思路

用动态规划的思路，

1. 对于s，可以把它拆分成left，right两部分
2. 求取left的最长合法子串，求right的最长合法字串，把两个子串连接起来，就是一个s的候选最长子串
3. 这里left，和right可以通过遍历i，0 <= i < s.size() 截取
4. 另外如果s的头尾字符是'(', ')', 我们还需要递归中间部分，也就是说，s的候选字串还有头尾的两个括号加上中间子串的最长这种情况。
5. 最后取这些候选中最长的字符串就是我们的结果了

边界情况或者递归结束条件是s的长度小于1，这时候如果s是括号，返回空字符串，如果不是就直接返回s

代码

```
from functools import lru_cache
class Solution:
    @lru_cache(1000)
    def removeInvalidParentheses(self, s: str) -> List[str]:
        if len(s) < 2:
            return {''} if s in '()' else {s}

        res = set()
        for i in range(1, len(s)):
            l = self.removeInvalidParentheses(s[:i])
            r = self.removeInvalidParentheses(s[i:])
            res |= {s1+s2 for s1 in l for s2 in r}

        if s[0]+s[-1] == '()':
            res |= {'(' + ss + ')' for ss in self.removeInvalidParentheses(s[1:-1])}
        maxlen = max(len(ss) for ss in res)
        res = [ss for ss in res if len(ss) == maxlen]
        return res
```

44.通配符匹配（困难）

1. 题目描述

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

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

两个字符串**完全匹配**才算匹配成功。

说明:

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

示例 1:

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

示例 2:

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

示例 3:

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

示例 4:

```
输入:
s = "adceb"
p = "*a*b"
输出: true
解释: 第一个 '*' 可以匹配空字符串, 第二个 '*' 可以匹配字符串 "dce".
```

示例 5:

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

2. 简单实现——BFS

```
class Solution {
public:
    bool isMatch(string s, string p) {
        //预处理p，去掉连续的*
        string temp = p;
        p = "";
        for(int i = 0; i < temp.size(); i++){
            p += temp[i];
            if(temp[i] == '*'){
                while(i < temp.size() && temp[i] == '*') i++;
                i--;
            }
        }
        if(p == "*") return true;
        if(s == ""){
            if(p == "") return true;
            else return false;
```

```
        }
        int len1 = s.size();
        int len2 = p.size();
        queue<vector<int>> q;//每个状态<i,j>表示当前匹配到s[i],p[j]
        q.push({0, 0});
        vector<vector<int>> visited = vector<vector<int>>(len1+1, vector<int>
(len2+1, false));
        visited[0][0] = true;
        while(!q.empty()){
            int i = q.front()[0];
            int j = q.front()[1];
            q.pop();
            if(i == len1 && j == len2)//正好匹配完两者了
                return true;
            if(p[j] == '*'){
                if(j == len2-1)//p已经到最后一个字符且为*，s剩下字符必然匹配
                    return true;
                if(i+1 <= len1 && !visited[i+1][j]){//s[i]匹配入*，s[i+1]继续匹配*
                    q.push({i+1, j});
                    visited[i+1][j] = true;
                }
                if(i+1<=len1 && j+1<=len2 && !visited[i+1][j+1]){//s[i]匹配入*,
s[i+1]匹配p[j+1]
                    q.push({i+1, j+1});
                    visited[i+1][j+1] = true;
                }
                if(j+1 <= len2 && !visited[i][j+1]){//s[i]不与*匹配，s[i]匹配p[j+1]
                    q.push({i, j+1});
                    visited[i][j+1] = true;
                }
            }
            else if(s[i] == p[j] || p[j] == '?'){//只有相等或p[j]为?才匹配
                if(i+1 <= len1 && j+1 <= len2 && !visited[i+1][j+1]){
                    q.push({i+1, j+1});
                    visited[i+1][j+1] = true;
                }
            }
        }
        return false;
    }
};
```

3. 最简解法——贪心法

本题难点在于处理星号的匹配，用iStar和jStar表示星号在s和p中匹配的位置，初始值为-1，i和j表示当前匹配的位置，匹配过程如下：

- 如果s和p中字符匹配，则分别自增i和j
- 否则如果p中当前字符为星号，则标记iStar和jStar，同时自增j
- 否则如果iStar >= 0，表示之前匹配过星号，因为星号可以匹配任意字符串，所以继续递增i，同时移动j为jStar下一个字符
- 否则返回false

当s中字符匹配完，p中字符不能有除星号以外字符

贪心体现在要尽可能地少将s中的字符匹配入*中

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int i = 0, j = 0, iStar = -1, jStar = -1, m = s.size(), n = p.size();
        while (i < m) {
            if (j < n && (s[i] == p[j] || p[j] == '?'))
                ++i, ++j;
            else if (j < n && p[j] == '*') {
                iStar = i;//记录p中最新的*匹配到s的索引值（istart尚未匹配）
                jStar = j++;//记录p中最新的*的索引值
            }
            else if (iStar >= 0) {
                i = ++iStar;//之前的走法匹配失败，回溯到iStar，并将其匹配入*
                j = jStar + 1;//继续匹配*后下一个字符
            }
            else return false;
        }
        while (j < n && p[j] == '*') ++j;//去除多余星号
        return j == n;
    }
};
```

4. 动态规划

(一) 状态

- f[i][j]表示s1的前i个字符，和s2的前j个字符，能否匹配

(二) 转移方程

1. 如果s1的第 i 个字符和s2的第 j 个字符相同，或者s2的第 j 个字符为 "."
   f[i][j] = f[i - 1][j - 1]
2. 如果s2的第 j 个字符为 *
   1. 若s2的第 j 个字符匹配空串, f[i][j] = f[i][j - 1]
   2. 若s2的第 j 个字符匹配s1的第 i 个字符, f[i][j] = f[i - 1][j]
      - 这里注意不是 f[i - 1][j - 1]， 举个例子就明白了 (abc, a*) f[3][2] = f[2][2]

(三) 初始化

- f[0][i] = f[0][i - 1] && s2[i] == *
- 即s1的前0个字符和s2的前i个字符能否匹配

(四) 结果

- f[m][n]

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size();
        int n = p.size();
        vector<vector<bool>> dp(m+1, vector<bool>(n+1,false));
```

```
        dp[0][0] = true;
        for(int i = 1; i<n+1; i++){
            if(p[i-1] == '*')
                dp[0][i] = true;
            else
                break;
        }
        for(int i = 1; i<m+1; i++)
            for(int j = 1; j<n+1; j++){
                if(p[j-1] == '?' ||s[i-1] == p[j-1])
                    dp[i][j] = dp[i-1][j-1];
                if(p[j-1] == '*')
                    dp[i][j] = dp[i][j-1] || dp[i-1][j];
            }
        return dp[m][n];
    }
};
```

10.正则表达式匹配（困难）

1. 题目描述

给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。

```
'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
```

所谓匹配，是要涵盖 **整个** 字符串 `s` 的，而不是部分字符串。

说明:

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

示例 1:

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

示例 2:

```
输入:
s = "aa"
p = "a*"
输出: true
解释: 因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串
"aa" 可被视为 'a' 重复了一次。
```

示例 3:

```
输入:
s = "ab"
p = ".*"
输出: true
解释: ".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
```

示例 4:

```
输入:
s = "aab"
p = "c*a*b"
输出: true
解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。
```

示例 5:

```
输入:
s = "mississippi"
p = "mis*is*p*."
输出: false
```

2. 简单实现

上题动态规划稍改即可

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int ns = s.length();
        int np = p.length();
        if(p.empty()) return s.empty();
        //dp[i][j]表示s[0...i-1]与p[0...j-1]能否匹配
        vector<vector<bool>> dp(ns+1, vector<bool>(np+1, false));
        dp[0][0] = true;
        for(int i = 1; i <= np; i++)
            if(i-2 >= 0 && p[i-1] == '*')
                dp[0][i] = dp[0][i-2];
        for(int i = 1; i <= ns; i++)
            for(int j = 1; j <= np; j++){
                if(p[j-1] == s[i-1] || p[j-1] == '.')
                    dp[i][j] = dp[i-1][j-1];
                if(p[j-1] == '*'){
                    bool zero, one;
                    if(j-2 >= 0){
                        zero = dp[i][j-2];//零次
                        one = (p[j-2] == s[i-1] || p[j-2] == '.') && dp[i-1][j];//
一次
                        dp[i][j] = zero || one;
                    }
                }
            }
```

```
            return dp[ns][np];
        }
    };
```

排序和搜索

324.摆动排序 II（中等）

1. 题目描述

   给定一个无序的数组 `nums`，将它重新排列成 `nums[0] < nums[1] > nums[2] < nums[3]...` 的顺序。

   示例 1:

   ```
   输入: nums = [1, 5, 1, 1, 6, 4]
   输出: 一个可能的答案是 [1, 4, 1, 5, 1, 6]
   ```

   示例 2:

   ```
   输入: nums = [1, 3, 2, 2, 3, 1]
   输出: 一个可能的答案是 [2, 3, 1, 3, 1, 2]
   ```

   说明:你可以假设所有输入都会得到有效的结果。

   进阶:你能用 O(n) 时间复杂度和 / 或原地 O(1) 额外空间来实现吗?

2. 题解

1. 解法1：排序

首先，我们可以很容易想到一种简单的解法：将数组进行**排序**，然后从中间位置进行**等分**（如果数组长度为奇数，则将中间的元素分到前面），然后将两个数组进行**穿插**。

例如：
对于数组[1, 5, 2, 4, 3]，我们将其排序，得到[1, 2, 3, 4, 5]，然后将其分割为[1, 2, 3]和[4, 5]，对两个数组进行穿插，得到[1, 4, 2, 5, 3]。

但是这一解法有一个问题，例如，对于数组[1, 2, 2, 3]，按照这种做法求得的结果仍为[1, 2, 2, 3]。如果题目不要求各元素严格大于或小于相邻元素，即，只要求nums[0] <= nums[1] >= nums[2] <= nums[3]...，那么这一解法是符合要求的，但题目要求元素相互严格大于或小于，那么需要稍微做一点改进。

为了方便阅读，我们在下文中定义较小的子数组为数组A，较大的子数组为数组B。显然，出现上述现象是因为nums中存在重复元素。实际上，由于穿插之后，相邻元素必来自不同子数组，所以A或B内部出现重复元素是不会出现上述现象的。所以，出现上述情况其实是因为数组A和数组B**出现了相同元素**，我们用r来表示这一元素。而且我们可以很容易发现，如果A和B都存在r，那么r一定是**A的最大值**，**B的最小值**，这意味着r一定出现在A的尾部，B的头部。其实，如果这一数字的个数较少，不会出现这一现象，只有当这一数字个数达到原数组元素总数的一半，才会在穿插后的出现在相邻位置。以下举几个例子进行形象地说明：

例如，对于数组[1,1,2,2,3,3],分割为[1,1,2]和[2,3,3]，虽然A和B都出现了2，但穿插后为[1,**2**,1,3,**2**,3]，满足要求。而如果2的个数再多一些，即[1,1,2,2,**2**,3]，分割为[1,1,2]和[2,2,3]，最终结果为[1,**2**,1,**2**,**2**,3]，来自A的2和来自B的2出现在了相邻位置。

出现这一问题是因为重复数在A和B中的位置决定的，因为r在A尾部，B头部，所以如果r个数太多（**大于等于**(length(nums) + 1)/2），就可能在穿插后相邻。要解决这一问题，我们需要使A的r和B的r在**穿插后尽可能分开**。一种可行的办法是将A和B反序：

例如，对于数组[1,1,2,2,2,3]，分割为[1,1,2]和[2,2,3]，分别反序后得到[2, 1, 1]和[3, 2, 2]，此时2在A头部，B尾部，穿插后就不会发生相邻了。

当然，这只能解决r的个数**等于**(length(nums) + 1)/2的情况，如果r的个数**大于**(length(nums) + 1)/2，还是会出现相邻。但实际上，这种情况是不存在有效解的，也就是说，这种数组对于本题来说是非法的。

此时我们得到了第一个解法，由于需要使用排序，所以时间复杂度为O(NlogN)，由于需要存储A和B，所以空间复杂度为O(N)。

2. 解法2：快速选择 + 3-way-partition

上一解法之所以时间复杂度为O(NlogN)，是因为使用了排序。但回顾解法1，我们发现，我们实际上**并不关心A和B内部的元素顺序**，只需要满足A和B长度相同（或相差1），且A中的元素小于等于B中的元素，且r出现在A的头部和B的尾部即可。实际上，由于A和B长度相同（或相差1），所以r实际上是原数组的中位数，下文改用mid来表示。因此，我们第一步其实不需要进行排序，而**只需要找到中位数即可。**而寻找中位数可以用**快速选择算法**实现，时间复杂度为O(n)。

该算法与快速排序算法类似，在一次递归调用中，首先进行partition过程，即利用一个元素将原数组划分为两个子数组，然后将这一元素放在两个数组之间。两者区别在于快速排序接下来需要对左右两个子数组进行递归，而快速选择只需要对一侧子数组进行递归，所以快速选择的时间复杂度为O(n)。详细原理可以参考有关资料，此处不做赘述。

在C++中，可以用STL的nth_element()函数进行快速选择，这一函数的效果是将数组中第n小的元素放在数组的第n个位置，同时保证其左侧元素不大于自身，右侧元素不小于自身。

找到中位数后，我们需要利用3-way-partition算法将**中位数放在数组中部，同时将小于中位数的数放在左侧，大于中位数的数放在右侧。**该算法与快速排序的partition过程也很类似，只需要在快速排序的partition过程的基础上，添加一个指针k用于定位大数：

```
int i = 0, j = 0, k = nums.size() - 1;
        while(j < k){
            if(nums[j] > mid){
                swap(nums[j], nums[k]);
                --k;
            }
            else if(nums[j] < mid){
                swap(nums[j], nums[i]);
                ++i;
                ++j;
            }
            else{
                ++j;
            }
        }
```

在这一过程中，指针j和k从左右两侧同时出发相向而行，每次要么j移动一步，要么k移动一步，直到相遇为止。这一过程的时间复杂度显然为O(N)。

至此，原数组被分为3个部分，左侧为小于中位数的数，中间为中位数，右侧为大于中位数的数。之后的做法就与解法1相同了：我们只需要将数组从中间等分为2个部分，然后反序，穿插，即可得到最终结果。以下为完整实现：

3. 解法3：快速选择 + 3-way-partition + 虚地址

接下来，我们思考如何简化空间复杂度。上文提到，解法1和2之所以空间复杂度为O(N)，是因为最后一步穿插之前，需要保存A和B。在这里我们使用所谓的虚地址的方法来省略穿插的步骤，或者说将穿插融入之前的步骤，即在3-way-partiton（或排序）的过程中顺便完成穿插，由此来省略保存A和B的步骤。“地址”是一种抽象的概念，在本题中地址就是数组的索引。

BTW，由于虚地址较为抽象，需要读者有一定的数学基础和抽象思维能力，如果实在理解不了没有关系，解法2已经是足够优秀的解法。

如果读者学习过操作系统，可以利用操作系统中的物理地址空间和逻辑地址空间的概念来理解。简单来说，这一方法就是将数组从原本的空间映射到一个虚拟的空间，虚拟空间中的索引和真实空间的索引存在某种映射关系。在本题中，我们需要建立一种映射关系来描述“分割”和“穿插”的过程，建立这一映射关系后，我们可以**利用虚拟地址**访问元素，**在虚拟空间中**对数组进行3-way-partition或排序，使数组**在虚拟空间中**满足某一空间关系。完成后，数组在真实空间中的空间结构就是我们最终需要的空间结构。

在某些场景下，可能映射关系很简洁，有些场景下，映射关系可能很复杂。而如果映射关系太复杂，编程时将会及其繁琐容易出错。在本题中，想建立一个简洁的映射，有必要对前面的3-way-partition进行一定的修改，我们不再将小数排在左边，大数排在右边，而是将**大数排在左边，小数排在右边**，在这种情况下我们可以用一个非常简洁的公式来描述映射关系：`#define A(i) nums[(1+2*(i)) % (n|1)]`，i是虚拟地址，(1+2*(i)) % (n|1)是实际地址。其中n为数组长度，‘|’为按位或，如果n为偶数，(n|1)为n+1，如果n为奇数，(n|1)仍为n。

```
Accessing A(0) actually accesses nums[1].
Accessing A(1) actually accesses nums[3].
Accessing A(2) actually accesses nums[5].
Accessing A(3) actually accesses nums[7].
Accessing A(4) actually accesses nums[9].
Accessing A(5) actually accesses nums[0].
Accessing A(6) actually accesses nums[2].
Accessing A(7) actually accesses nums[4].
Accessing A(8) actually accesses nums[6].
Accessing A(9) actually accesses nums[8].
```

以下为完整代码：

```
class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        int n = nums.size();

        // Find a median.
        auto midptr = nums.begin() + n / 2;
        nth_element(nums.begin(), midptr, nums.end());
        int mid = *midptr;

        // Index-rewiring.
        #define A(i) nums[(1+2*(i)) % (n|1)]

        // 3-way-partition-to-wiggly in O(n) time with O(1) space.
        int i = 0, j = 0, k = n - 1;
        while (j <= k) {
            if (A(j) > mid)
                swap(A(i++), A(j++));
            else if (A(j) < mid)
                swap(A(j), A(k--));
            else
                j++;
        }
    }
};
```

时间复杂度与解法2相同，为O(N)，空间复杂度为O(1)。

当然，也可以在解法1中利用虚地址方法，即利用虚地址对nums进行排序，那么时间复杂度为O(NlogN)，空间复杂度为O(1)。

378.有序矩阵中第K小的元素（中等）

1. 题目描述

   给定一个 *n* x *n* 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第k小的元素。请注意，它是排序后的第k小元素，而不是第k个元素。

   示例:

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

返回 13。
```

   说明: 你可以假设 k 的值永远是有效的, 1 ≤ k ≤ n2 。

2. 简单实现

   用大顶堆维护前k个最小元素

```
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int n = min(int(matrix.size()), k);//最大仅需遍历边长为k的子矩阵即可
        priority_queue<int> q;
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++){
                if(q.size() < k)
                    q.push(matrix[i][j]);
                else if(matrix[i][j] < q.top()){
                    q.pop();
                    q.push(matrix[i][j]);
                }
            }
        return q.top();
    }
};
```

3. 二分法

- 找出二维矩阵中最小的数left，最大的数right，那么第k小的数必定在left~right之间
- mid=(left+right) / 2；在二维矩阵中寻找小于等于mid的元素个数count
- 若这个count小于k，表明第k小的数在右半部分且不包含mid，即left=mid+1, right=right，又保证了第k小的数在left~right之间
- 若这个count大于k，表明第k小的数在左半部分且可能包含mid，即left=left, right=mid，又保证了第k小的数在left~right之间
- 因为每次循环中都保证了第k小的数在left~right之间，当left==right时，第k小的数即被找出，等于right

```
class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        int lo = matrix[0][0], hi = matrix[n-1][n-1];
        while(lo < hi) {
            int mid = lo + (hi - lo) / 2;
            int count = 0,  j = matrix[0].length - 1;
            for(int i = 0; i < matrix.length; i++) {//利用有序矩阵的特性，在线性时间内计
数
                while(j >= 0 && matrix[i][j] > mid)
                    j--;
                count += (j + 1);
            }
            if(count < k)
                lo = mid + 1;
            else //为保证得到的结果在矩阵中，count=k时不能直接返回
                hi = mid;
        }
        return lo;
    }
}
```

动态规划

152.乘积最大子序列（中等）

1. 题目描述

   给定一个整数数组 `nums` ，找出一个序列中乘积最大的连续子序列（该序列至少包含一个数）。

   示例 1:

   ```
   输入: [2,3,-2,4]
   输出: 6
   解释: 子数组 [2,3] 有最大乘积 6。
   ```

   示例 2:

   ```
   输入: [-2,0,-1]
   输出: 0
   解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
   ```

2. 最优解法

   - 遍历数组时计算当前最大值，不断更新
   - 令imax为以当前数结尾的最大值，则以当前数结尾的最大值为 `imax = max(imax * nums[i], nums[i])`
   - 由于存在负数，那么会导致最大的变最小的，最小的变最大的。因此还需要维护以当前数结尾的最小值imin， `imin = min(imin * nums[i], nums[i])`
   - 当负数出现时则imax与imin进行交换再进行下一步计算
   - 时间复杂度：O(n)

```cpp
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int ans = INT_MIN, imax = 1, imin = 1;
        for(int i=0; i<nums.size(); i++){
            if(nums[i] < 0){
              int tmp = imax;
              imax = imin;
              imin = tmp;
            }
            imax = max(imax*nums[i], nums[i]);
            imin = min(imin*nums[i], nums[i]);

            ans = max(ans, imax);
        }
        return ans;
    }
};
```

309. 最佳买卖股票时机含冷冻期（中等）

1. 题目描述

给定一个整数数组，其中第 *i* 个元素代表了第 *i* 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

示例:

```
输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

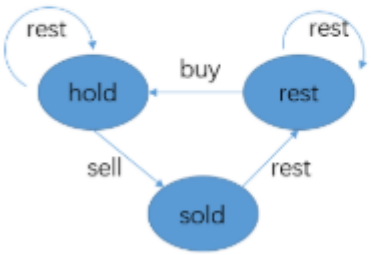
2. 最优解法

思路

每天可能存在三种状态：

- hold：继续持有股票
- sold：卖出股票
- rest：什么都不做

转换关系如下：



sold：

- 前一天hold，当日卖出股票

hold： 可由两个情况转换来

- 前一天hold，当日rest
- 前一天rest，当日买入股票变为hold

rest：

- 前一天sold，当日必须rest
- 前一天rest，当日继续rest

所以

- `sold[i] = hold[i-1] + price[i];`
- `hold[i] = max(hold[i-1], rest[i-1] - price[i])`
- `rest[i] = max(rest[i-1], sold[i-1])`

最后一天最大值情况为要么什么都不做，要么卖出股票，即 `max(sold, rest)`。

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        if(len <= 1) return 0;
        int hold = INT_MIN;//初始化为负的最小值，防止第一天出错（第一天只能买）
        int rest = 0, sold = 0;
        for(int i = 0; i < len; i++){
            int temp = sold;
            sold = hold + prices[i];
            hold = max(hold, rest-prices[i]);
            rest = max(rest, temp);
        }
        return max(sold, rest);
    }
};
```

扩展：动态规划的三板斧

动态规划，我大体上分成这么三板斧。

第一板斧 —— 找思路，看如何自底部向上。

```
F[n]的定义
    有些题目需要做一些变形，直接用题目的定义根本推不出来。
    416，413就是一个很好的例子。
动态规划的公式其实不难推，就那么几种（下面四种）。
如果这几种套上去都搞不定，那么八成F[n]需要重新定义。

定义完毕，就要找F[n]和前面n步的关系了，分成四种，挨个套就行，难度由从低到高
a, F[n]跟前面一两步有关
    这种简单
b, F[n]跟前面n步都有关
    这种找到公式也不难，但要注意时间优化了，不优化往往就n^2了，举例子《最长上升子序列》
c, F[n]需要细分
    这种情况下，你发现看着像，但是跟前面n步的关系整的你头疼。
    举个例子，股票的几个中等难度的题就是这样。
    那就把F[n]分成 F1[n] F2[n]
d, F[n]，一纬数组不够用了
    F[n]的细分其实是这个的简单情况。
    举个例子，股票的状态有出售，买入。那么F[n]再加一纬数组，变成 F[n][]，第二纬只有0，1即可。
    二维比较难的，可以看这道题 分割等和子集

这道题其实介于c和d之间，即可以用c也可以用d。
如果直接F(n)去想和之前的关系，反正我是卡在这里了，情况太复杂。。

要么把问题细分，要么上个二维数组（有些情况下，甚至三维），这个时候公式推起来就快了。
labuladong是把状态分成了出售和保留
但事实上，还有的分成了三种状态，也就是题目中描述的三种状态。
```

第二板斧 —— 确认公式

```
这一步反而基本上没什么难度了
第一，当前持有股票
    a,昨天如果持有股票，那么今天就保留 hold[n - 1]
    b,前天出售个票，今天买入 unhold[n - 2] - prices[n]
    hold[n]=max(a,b)

第二，当前没有股票
    a，昨天如果持有股票，那么今天就卖出 hold[n - 1] + prices[n]
    b，昨天没有持有股票，今天也没有 unhold[n - 1]
    unhold[n] = max(a,b)
最后 max(hold[n], unhold[n])
```

第三板斧 —— 确认需要存储的上1/N步信息是否找对了

```
这一步有三个作用。
第一，double check下公式。
第二，部分情况下，如果只考虑前一步，前两步的话，我们没必要用数组来存储，几个临时变量就OK了
第三，边界条件的确认
    这道题的边界条件，是需要n-2，所以在0，1的初始化需要注意
这三点考虑完，基本上代码很快就写完了
```

279.完全平方数（中等）

1. 题目描述

给定正整数 *n*，找到若干个完全平方数（比如 `1, 4, 9, 16, ...`）使得它们的和等于 *n*。你需要让组成和的完全平方数的个数最少。

示例 1:

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
```

示例 2:

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9.
```

2. 简单实现

```
class Solution {
public:
    int numSquares(int n) {
        if(n == 1) return 1;
        vector<int> dp = vector<int>(n+1, INT_MAX);
        dp[0] = 0;
        dp[1] = 1;
        for(int i = 2; i <= n; i++){
```

```
            dp[i] = i;
            for(int j = 0; j*j <= i; j++)
                dp[i] = min(dp[i], dp[i-j*j] + 1);
        }
        return dp[n];
    }
};
```

3. 数学解法

数学解法是在评论区看到的，也就是有个理论：任何一个数，都可以由小于等于4个的完全平方数相加得到。

然后根据这个理论，有个推论（我也不知道这个推论怎么来的）：

当n满足如下公式时，才只能由4个完全平方数得到, a 和b为某个整数：`n = 4^a (8b + 7)`

否则，就是只能由1-3个完全平方数得到。

```
class Solution {
public:
    int numSquares(int n) {
       int a = sqrt(n);
        if (a * a == n)//验证1
            return 1;
        for (int i = 1; i <= sqrt(n); i++){//验证2
            a = sqrt(n - i * i);
            if (i * i + a * a == n)
                return 2;
        }
        while (n % 4 == 0)
            n /= 4;
        if (n % 8 == 7)//验证4
            return 4;
        return 3;
    }
};
```

139.单词拆分（中等）

1. 题目描述

给定一个**非空**字符串 *s* 和一个包含**非空**单词列表的字典 *wordDict*，判定 *s* 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2:

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
     注意你可以重复使用字典中的单词。
```

示例 3:

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

2. 简单实现

dp[i]表示s[0...i]能否被拆分，状态转移为：

- 如果s[0....i]在字典内，则dp[i]为true
- 如果存在`0<=j<i`使得`dp[j]=true && s.substr(j+1,i-j)) > 0`，则dp[i]为true
- 上一步可以优化为在`j >= max(0, i-max_len)`中搜索即可，其中max_len为字典内最长的字符串长度

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

140.单词拆分 II（困难）

1. 题目描述

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

说明：

- 分隔时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

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

示例 2：

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

示例 3：

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

2. 简单改变
   - 在上一题的基础上简单改变代码就可以实现该功能，即用dp[i]存储s[0...i]所有可能的拆分，但 `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` 的case内存超限
   - 看了题解后发现，要先检查一下能否拆分，再拆分即可

```
class Solution {
public:
    bool judge(vector<string>& wordDict, const string& s){
        for(int i = 0; i < wordDict.size(); i++)
            if(wordDict[i] == s)
```

```
                return true;
            return false;
        }
        vector<string> wordBreak(string s, vector<string>& wordDict) {
            int size = wordDict.size();
            unordered_set<string> dict;
            int max_len = -1;//记录字典内最长的字符串长度
            for(int i = 0; i < size; i++){
                dict.insert(wordDict[i]);
                max_len = max(max_len, int(wordDict[i].size()));
            }
            //检查能否拆分，同上题
            vector<bool> dp1 = vector<bool>(s.size(), false);
            for(int i = 0; i < s.size(); i++){
                if(dict.count(s.substr(0, i+1)) > 0){
                    dp1[i] = true;
                    continue;
                }
                for(int j = i-1; j>=0 && i-j<=max_len; j--){
                    if(dp1[j] == true && dict.count(s.substr(j+1,i-j)) > 0){
                        dp1[i] = true;
                        break;
                    }
                }
            }
            if(!dp1[s.size()-1])
                return {};
            //拆分
            vector<vector<string>> dp = vector<vector<string>>(s.size());
            int len = s.size();
            for(int i = 0; i < len; i++){
                if(dict.count(s.substr(0, i+1)) > 0){
                    if(i == len -1)
                        dp[i].push_back(s.substr(0, i+1));
                    else
                        dp[i].push_back(s.substr(0, i+1) + ' ');
                }
                for(int j = i-1; j>=0 && i-j<=max_len; j--){
                    string cur = s.substr(j+1,i-j);
                    if(dp[j].size()>0 && dict.count(cur) > 0){
                        if(i == len -1)
                            for(int k = 0; k < dp[j].size(); k++)
                                dp[i].push_back(dp[j][k] + cur);
                        else
                            for(int k = 0; k < dp[j].size(); k++)
                                dp[i].push_back(dp[j][k] + cur + ' ');
                    }
                }
            }
            return dp[s.size()-1];
        }
    };
```

312. 戳气球（困难）

1. 题目描述

   有 `n` 个气球，编号为 `0` 到 `n-1`，每个气球上都标有一个数字，这些数字存在数组 `nums` 中。

   现在要求你戳破所有的气球。每当你戳破一个气球 `i` 时，你可以获得 `nums[left] * nums[i] * nums[right]` 个硬币。 这里的 `left` 和 `right` 代表和 `i` 相邻的两个气球的序号。注意当你戳破了气球 `i` 后，气球 `left` 和气球 `right` 就变成了相邻的气球。

   求所能获得硬币的最大数量。

   说明:

   - 你可以假设 `nums[-1] = nums[n] = 1`，但注意它们不是真实存在的所以并不能被戳破。
   - 0 ≤ `n` ≤ 500, 0 ≤ `nums[i]` ≤ 100

   示例:

```
输入: [3,1,5,8]
输出: 167
解释: nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
     coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

2. 题解

   转换思维，不要想先爆谁，想想后爆谁，除去它以后区间内左右部分剩下的爆的先后次序不影响最后的结果

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        nums.insert(nums.begin(),1);
        nums.push_back(1);
        int n=nums.size();
        //dp[i][j]表示第i至第j个元素这个区间能获得的最大硬币数
        vector<vector<int>> dp = vector<vector<int>>(n, vector<int>(n, 0));
        for(int r = 2; r < n; r++)//r为区间长度
            for(int i = 0; i < n - r; i++){//i为左区间
                int j=i+r;//j为右区间
                for(int k = i + 1; k < j; k++)//在该区间内最后戳破k处的气球
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] +
nums[i]*nums[k]*nums[j]);
            }
        return dp[0][n-1];
    }
};
```

设计问题

146.LRU 缓存机制（中等）

1. 题目描述

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 `get` 和 写入数据 `put` 。

获取数据 `get(key)` - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。 写入数据 `put(key, value)` - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 **O(1)** 时间复杂度内完成这两种操作?

示例:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

2. 简单实现

```
class LRUCache {
private:
    int cap;
    // 双链表：装着 (key, value) 元组
    list<pair<int, int>> cache;
    // 哈希表：key 映射到 (key, value) 在 cache 中的位置
    unordered_map<int, list<pair<int, int>>::iterator> map;
public:
    LRUCache(int capacity) {
        this->cap = capacity;
    }

    int get(int key) {
        auto it = map.find(key);
        // 访问的 key 不存在
        if (it == map.end()) return -1;
        // key 存在，把 (k, v) 换到队头
        pair<int, int> kv = *map[key];
        cache.erase(map[key]);
        cache.push_front(kv);
        // 更新 (key, value) 在 cache 中的位置
        map[key] = cache.begin();
        return kv.second; // value
    }
```

```
    void put(int key, int value) {

        /* 要先判断 key 是否已经存在 */
        auto it = map.find(key);
        if (it == map.end()) {
            /* key 不存在，判断 cache 是否已满 */
            if (cache.size() == cap) {
                // cache 已满，删除尾部的键值对腾位置
                // cache 和 map 中的数据都要删除
                auto lastPair = cache.back();
                int lastKey = lastPair.first;
                map.erase(lastKey);
                cache.pop_back();
            }
            // cache 没满，可以直接添加
            cache.push_front(make_pair(key, value));
            map[key] = cache.begin();
        } else {
            /* key 存在，更改 value 并换到队头 */
            cache.erase(map[key]);
            cache.push_front(make_pair(key, value));
            map[key] = cache.begin();
        }
    }
};
```

TIP：c++中list（双向链表）的用法https://blog.csdn.net/yas12345678/article/details/52601578

1.关于list容器

list是一种序列式容器。list容器完成的功能实际上和数据结构中的双向链表是极其相似的，list中的数据元素是通过链表指针串连成逻辑意义上的线性表，也就是list也具有链表的主要优点，即：在链表的任一位置进行元素的插入、删除操作都是快速的。list的实现大概是这样的：list的每个节点有三个域：前驱元素指针域、数据域和后继元素指针域。前驱元素指针域保存了前驱元素的首地址；数据域则是本节点的数据；后继元素指针域则保存了后继元素的首地址。其实，list和循环链表也有相似的地方，即：头节点的前驱元素指针域保存的是链表中尾元素的首地址，list的尾节点的后继元素指针域则保存了头节点的首地址，这样，list实际上就构成了一个双向循环链。由于list元素节点并不要求在一段连续的内存中，显然在list中是不支持快速随机存取的，因此对于迭代器，只能通过“++”或“--”操作将迭代器移动到后继/前驱节点元素处。而不能对迭代器进行+n或-n的操作，这点，是与vector等不同的地方。

我想把三个常用的序列式放在一起对比一下是有必要的：

vector： vector和built-in数组类似，拥有一段连续的内存空间，能非常好的支持随即存取，即[]操作符，但由于它的内存空间是连续的，所以在中间进行插入和删除会造成内存块的拷贝，另外，当插入较多的元素后，预留内存空间可能不够，需要重新申请一块足够大的内存并把原来的数据拷贝到新的内存空间。这些影响了vector的效率，但是实际上用的最多的还是vector容器，建议大多数时候使用vector效率一般是不错的。vector的用法解析可以参考本人的另一篇随笔：http://www.cnblogs.com/BeyondAnyTime/archive/2012/08/08/2627666.html

list: list就是数据结构中的双向链表(根据sgi stl源代码)，因此它的内存空间是不连续的，通过指针来进行数据的访问，这个特点使得它的随即存取变的非常没有效率，因此它没有提供[]操作符的重载。但由于链表的特点，它可以以很好的效率支持任意地方的删除和插入。

deque: deque是一个double-ended queue，它的具体实现不太清楚，但知道它具有以下两个特点：它支持[]操作符，也就是支持随即存取，并且和vector的效率相差无几，它支持在两端的操作：push_back,push_front,pop_back,pop_front等，并且在两端操作上与list的效率也差不多。

因此在实际使用时，如何选择这三个容器中哪一个，应根据你的需要而定，具体可以遵循下面的**原则**：

1. 如果你需要高效的随即存取，而不在乎插入和删除的效率，使用vector
2. 如果你需要大量的插入和删除，而不关心随即存取，则应使用list
3. 如果你需要随即存取，而且关心两端数据的插入和删除，则应使用deque。

2.list中常用的函数

2.1list中的构造函数：

list() 声明一个空列表；

list(n) 声明一个有n个元素的列表，每个元素都是由其默认构造函数T()构造出来的

list(n,val) 声明一个由n个元素的列表，每个元素都是由其复制构造函数T(val)得来的

list(n,val) 声明一个和上面一样的列表

list(first,last) 声明一个列表，其元素的初始值来源于由区间所指定的序列中的元素

2.2 begin()和end()： 通过调用list容器的成员函数begin()得到一个指向容器起始位置的iterator，可以调用list容器的 end() 函数来得到list末端下一位置，相当于：int a[n]中的第n+1个位置a[n]，实际上是不存在的，不能访问，经常作为循环结束判断结束条件使用。

2.3 push_back() 和push_front()： 使用list的成员函数push_back和push_front插入一个元素到list中。其中push_back()从list的末端插入，而push_front()实现的从list的头部插入。

2.4 empty()： 利用empty() 判断list是否为空。

2.5 resize()： 如果调用resize(n)将list的长度改为只容纳n个元素，超出的元素将被删除，如果需要扩展那么调用默认构造函数T()将元素加到list末端。如果调用resize(n,val)，则扩展元素要调用构造函数T(val)函数进行元素构造，其余部分相同。

2.6 clear()： 清空list中的所有元素。

2.7 front()和back()： 通过front()可以获得list容器中的头部元素，通过back()可以获得list容器的最后一个元素。但是有一点要注意，就是list中元素是空的时候，这时候调用front()和back()会发生什么呢？实际上会发生不能正常读取数据的情况，但是这并不报错，那我们编程序时就要注意了，个人觉得在使用之前最好先调用empty()函数判断list是否为空。

2.8 pop_back和pop_front()： 通过删除最后一个元素，通过pop_front()删除第一个元素；序列必须不为空，如果当list为空的时候调用pop_back()和pop_front()会使程序崩掉。

2.9 assign()： 具体和vector中的操作类似，也是有两种情况，第一种是：l1.assign(n,val)将 l1中元素变为n个T(val) 。第二种情况是：l1.assign(l2.begin(),l2.end())将l2中的从l2.begin()到l2.end()之间的数值赋值给l1。

2.10 swap()： 交换两个链表(两个重载)，一个是l1.swap(l2); 另外一个是swap(l1,l2)，都可能完成连个链表的交换。

2.11 reverse()： 通过reverse()完成list的逆置。

2.12 merge()： 合并两个链表并使之默认升序(也可改)，l1.merge(l2，greater<int>()); 调用结束后l2变为空，l1中元素包含原来l1 和 l2中的元素，并且排好序，升序。其实默认是升序，greater<int>()可以省略，另外greater<int>()是可以变的，也可以不按升序排列。

208.实现 Trie (前缀树)（中等）

1. 题目描述

   实现一个 Trie (前缀树)，包含 `insert`，`search`，和 `startsWith` 这三个操作。

   示例:

```
Trie trie = new Trie();

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

说明:

- 你可以假设所有的输入都是由小写字母 `a-z` 构成的。
- 保证所有输入均为非空字符串。

2. 简单实现

```
class Trie {
public:
    unordered_map<char, Trie*> children;
    bool isWord;
    /** Initialize your data structure here. */
    Trie() {
        isWord = false;
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new Trie();
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            // cout << word[i];
            if(cur->children.count(word[i]) <= 0)
                return false;
            cur = cur->children[word[i]];
        }
        // cout << 'a';
        return cur->isWord;
    }

    /** Returns if there is any word in the trie that starts with the given prefix.
*/
    bool startsWith(string prefix) {
        Trie* cur = this;
        for(int i = 0; i < prefix.size(); i++){
```

```
            if(cur->children.count(prefix[i]) <= 0)
                return false;
            cur = cur->children[prefix[i]];
        }
        return true;
    }
};
```

341.扁平化嵌套列表迭代器（中等）

1. 题目描述

   给定一个嵌套的整型列表。设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

   列表中的项或者为一个整数，或者是另一个列表。

   示例 1:

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

   示例 2:

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

2. 简单实现

   初始化时利用递归将嵌套列表扁平化

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

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

3. 使用栈

```
class NestedIterator {
private:
  stack<NestedInteger> st;
public:
  NestedIterator(vector<NestedInteger> &nestedList) {
    //从右往左进栈
    for (auto iter = nestedList.rbegin(); iter != nestedList.rend(); iter++)
      st.push(*iter);
  }
  int next() {
    auto t = st.top();
    st.pop();
    return t.getInteger();
  }
  bool hasNext() {//因为每次会先调用该函数，因此在此进行扁平化
    while (!st.empty()) {
      auto cur = st.top();
      if (cur.isInteger()) return true;//直到栈顶为数字
      st.pop();
      auto curList = cur.getList();
      for (auto iter = curList.rbegin(); iter != curList.rend(); iter++)
        st.push(*iter);
    }
```

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

295. 数据流的中位数（困难）

1. 题目描述

   中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

   例如，

   [2,3,4] 的中位数是 3

   [2,3] 的中位数是 (2 + 3) / 2 = 2.5

   设计一个支持以下两种操作的数据结构：

   - void addNum(int num) - 从数据流中添加一个整数到数据结构中。
   - double findMedian() - 返回目前所有元素的中位数。

   示例：

   ```
   addNum(1)
   addNum(2)
   findMedian() -> 1.5
   addNum(3)
   findMedian() -> 2
   ```

   进阶：

   1. 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？
   2. 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

2. 简单实现

   在插入时维护一个有序数组

```
class MedianFinder {
public:
    vector<int> data;
    int size;
    /** initialize your data structure here. */
    MedianFinder() {
        size = 0;
    }

    void addNum(int num) {
        if(size == 0)
            data.push_back(num);
        else{
            int l = 0, r = size -1;
            while(l <= r){
                int mid = l + (r - l) / 2;
                if(data[mid] == num){
                    data.insert(data.begin()+mid, num);
```

```
                    size++;
                    return;
                }
                else if(data[mid] < num)
                    l = mid + 1;
                else
                    r = mid - 1;
            }
            data.insert(data.begin()+l, num);
        }
        size++;
    }

    double findMedian() {
        if(size % 2 == 1)
            return data[size/2];
        else
            return float(data[size/2-1] + data[size/2]) / 2;
    }
};
```

3. 进阶实现

在进阶情况下，中位数必然在0-100之间，用数组统计小于0、0-100、大于100的数字的个数即可，可以节省大量空间

```
class MedianFinder {
public:
    vector<int> data;
    int size;
    /** initialize your data structure here. */
    MedianFinder() {
        data = vector<int>(103);
        size = 0;
    }
    void addNum(int num) {
        if(num < 0)
            data[0]++;
        else if(num > 100)
            data[102]++;
        else
            data[num+1]++;
        size++;
    }
    double findMedian() {
        int cnt = 0;
        int aim = -1;
        if(size % 2 == 1)
            aim = size / 2 + 1;
        else
            aim = size / 2;
        for(int i = 0; i < 103; i++){
            if(cnt + data[i] >= aim){
```

```
                if(size % 2 == 1)
                    return i-1;
                else{
                    if(cnt + data[i] >= aim+1)
                        return i-1;
                    else{
                        int j = i + 1;
                        while(j < 103){
                            if(data[j] > 0)
                                return float(i+j-2)/2;
                            j++;
                        }
                    }
                }
            }
            else
                cnt += data[i];
        }
        return -1;
    }
};
```

4. 最优解法——两个堆

用以下方式维护两个堆：

- 用于存储输入数字中较小一半的最大堆
- 用于存储输入数字的较大一半的最小堆

这样就可以访问输入中的中值：它们组成堆的顶部！

```
class MedianFinder {
    priority_queue<int> lo; // max heap
    priority_queue<int, vector<int>, greater<int>> hi; // min heap
public:
    // Adds a number into the data structure.
    void addNum(int num) {
        lo.push(num);// Add to max heap
        hi.push(lo.top());// balancing step
        lo.pop();
        if (lo.size() < hi.size()) {// maintain size property
            lo.push(hi.top());
            hi.pop();
        }
    }
    // Returns the median of current data stream
    double findMedian() {
        return lo.size() > hi.size() ? (double) lo.top() : (lo.top() + hi.top()) *
0.5;
    }
};
```

数学

179. 最大数（中等）

1. 题目描述

   给定一组非负整数，重新排列它们的顺序使之组成一个最大的整数。

   示例 1:

   ```
   输入: [10,2]
   输出: 210
   ```

   示例 2:

   ```
   输入: [3,30,34,5,9]
   输出: 9534330
   ```

   说明: 输出结果可能非常大，所以你需要返回一个字符串而不是整数。

2. 简单实现

   改写sort的排序函数即可

   ```
   class Solution {
   public:
       static bool cmp(string a, string b){
           if(stol(a+b) > stol(b+a))//转成long防止溢出
               return true;
           else
               return false;
       }
       string largestNumber(vector<int>& nums) {
           vector<string> v(nums.size());
           for(int i = 0; i < nums.size(); i++)
               v[i] = to_string(nums[i]);
           sort(v.begin(), v.end(), cmp);
           string ans = "";
           for(int i = 0; i < v.size(); i++)
               ans += v[i];
           if(ans[0] == '0')//防止出现000的情况
               ans = "0";
           return ans;
       }
   };
   ```

149. 直线上最多的点数（困难）

1. 题目描述

   给定一个二维平面，平面上有 *n* 个点，求最多有多少个点在同一条直线上。

   示例 1:

```
输入: [[1,1],[2,2],[3,3]]
输出: 3
解释:
^
|
|        o
|     o
|  o
+------------->
0  1  2  3  4
```

示例 2:

```
输入: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出: 4
解释:
^
|
|  o
|     o        o
|        o
|  o        o
+------------------->
0  1  2  3  4  5  6
```

2. 正确解法

只能两两比较，问题的关键在于避免浮点数的误差问题

方法一：

Given：points 一系列坐标点
Targer: 共线坐标最多的点
Main idea: 遍历每一个点， 计算其他点与这个点的斜率来判断是否共线，记录每个点与其他点共线最多的次数
Tricks: 1. 本质上来说就是记录斜率查字典，但是计算斜率会变成flaot损失精度，比如(0,0)
(1,10000000000000000000000000000001) 和(1,100000000000000000000000000000000)就有可能被判定为共线
所以在字典中， 我们不记录斜率，而记录“简化后的”横纵坐标之差，
i.e.,： [[1,1],[2,2],[3,3],[6,6]],从[1,1]开始遍历，差值为:
[1,1], [2,2],[5,5] 对于每一对 除以 gcd 都变为[1,1]
所以字典 dict['1'+"a+'1']=3. 我们加'a'表示隔断，毕竟字典不能放数组，所以用 1a1表示差[1,1]
2. 不要用 math.gcd: 自己写gcd, 比如[-1,-2] 对于点[0,0]
math.gcd=1, 横纵坐标只差就变成了[-1,-2] 而我们想要的是都变成正值，即
gcd=-1, 横纵坐标只差 [1,2]
3.由于对于每个点，我们计算的是点其他点共线的最大次数，最后结果要+1

方法二：采用乘法

```
class Solution {
public:
    int maxPoints(vector<vector<int>>& points)
    {
        //两点确定一条直线
```

```
        const int size = points.size();
        if(size<3)
            return size;

        int Max=0;
        for(int i=0;i< size;i++)//i表示数组中的第i+1个点
        {
            //same用来表示和i一样的点
            int same=1;
            for(int j=i+1;j < size;j++)//j表示数组中的第j+1个点
            {
                int count=0;
                // i、j在数组中是重复点，计数
                if(points[i][0]==points[j][0]&&points[i][1]==points[j][1])
                    same++;
                else//i和j不是重复点，则计算和直线ij在一条直线上的点
                {
                    count++;
                    long long xDiff = (long long)(points[i][0] - points[j][0]);//
Δx1
                    long long yDiff = (long long)(points[i][1] - points[j][1]);//
Δy1

//Δy1/Δx1=Δy2/Δx2 => Δx1*Δy2=Δy1*Δx2，计算和直线ji在一条直线上的点
                    for (int k = j + 1; k < size; ++k)
                    {
                        if (xDiff * (points[i][1] - points[k][1]) == yDiff *
(points[i][0] - points[k][0]))
                            count++;
                    }
                }
                Max=max(Max,same+count);
            }
            if(Max> size /2)
                return Max;//若某次最大个数超过所有点的一半，则不可能存在其他直线通过更多的点
        }
        return Max;
    }
};
```

其他

406. 根据身高重建队列（中等）

1. 题目描述

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

注意： 总人数少于1100人。

示例

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

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

2. 简单实现

类似于插入排序的方法，按 `for(k从小到大){ for(h从小到大){ } }` 的顺序处理，可以保证后插入的人不影响已经插入的人

```
class Solution {
public:
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        map<int, vector<int>> m;//按k值进行分组，且k从小到大排序，存储k相同的h
        for(int i = 0; i < people.size(); i++)
            m[people[i][1]].push_back(people[i][0]);
        vector<vector<int>> ans;
        //处理k=0的情况，一定按h是从小到大排序的
        sort(m[0].begin(), m[0].end());
        for(int i = 0; i < m[0].size(); i++)
            ans.push_back({m[0][i], 0});
        for(auto it = m.begin(); it != m.end(); it++){//k从小到大遍历
            if(it == m.begin())
                continue;
            sort(it->second.begin(), it->second.end());//h从小到大排序
            int k = it->first;
            for(int i = 0; i < it->second.size(); i++){//h从小到大遍历
                int idx = 0;//当前检查到的ans索引
                int cnt = 0;//当前大于等于h的个数
                while(idx < ans.size() && cnt < k){//找到满足cnt = k的第一个可插入点
                    if(ans[idx][0] >= it->second[i])
                        cnt++;
                    idx++;
                }
                if(idx >= ans.size())//队尾，直接插入，此时cnt一定等于k（否则之后要在它前面
插入比它高且k值大于等于它的k值的人，这是不可能的，因为这个插入的人的要求也将永远无法达到）
                    ans.push_back({it->second[i], k});
                else{
                    //要保证插入的位置不影响后面的人的k值，所以要站在第一个比它高的人的前一个
                     while(idx < ans.size() && ans[idx][0] <= it->second[i])
                         idx++;
                    ans.insert(ans.begin()+idx, {it->second[i], k});
                }
            }
        }
        return ans;
    }
};
```

3. 最优解法

   - 先排身高更高的，这是要防止后排入人员影响先排入人员位置
   - 每次排入新人员 `[h,k]` 时，已处于队列的人身高都 `>=h` ，所以新排入位置就是 `people[k]`

   有了这两个思路代码实现就非常简单了

   1. 先将 `people` 按照**身高降序排序**，又由于每次插入的位置是 `k` ，所以**相同身高需要按 `k` 升序排序**，否则插入位置会越界
   2. 由于后续需要频繁使用 `insert()` 操作，建议使用 `list` 作为中间容器
   3. 循环地从头读取 `people` ，根据 `people[i][1]` 也就是 `k` ,插入 `list` ，注意 `list` 的迭代器不支持随机访问，需要使用 `advance()` 找到应插入位置
   4. 将完成所有插入操作的 `list` 重建为 `vector` 返回

   见代码

```
class Solution {
public:
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        // 排序
        sort(people.begin(), people.end(),
                [](const vector<int>& lhs, const vector<int>& rhs)
                 {return lhs[0] == rhs[0] ? lhs[1] <= rhs[1] : lhs[0] > rhs[0];});
        int len = people.size();
        list<vector<int>> tmp;
        // 循环插入
        for(int i = 0; i < len; ++i){
            auto pos = tmp.begin();
            advance(pos, people[i][1]);
            tmp.insert(pos, people[i]);
        }
        // 重建vector返回
        return vector<vector<int>>(tmp.begin(), tmp.end());
    }
};
```

42. 接雨水（困难）

1. 题目描述

   给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

   上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。 **感谢 Marcos** 贡献此图。

   示例:

```
输入：[0,1,0,2,1,0,1,3,2,1,2,1]
输出：6
```

2. 简单实现

观察与某个柱子形成储水区的条件：

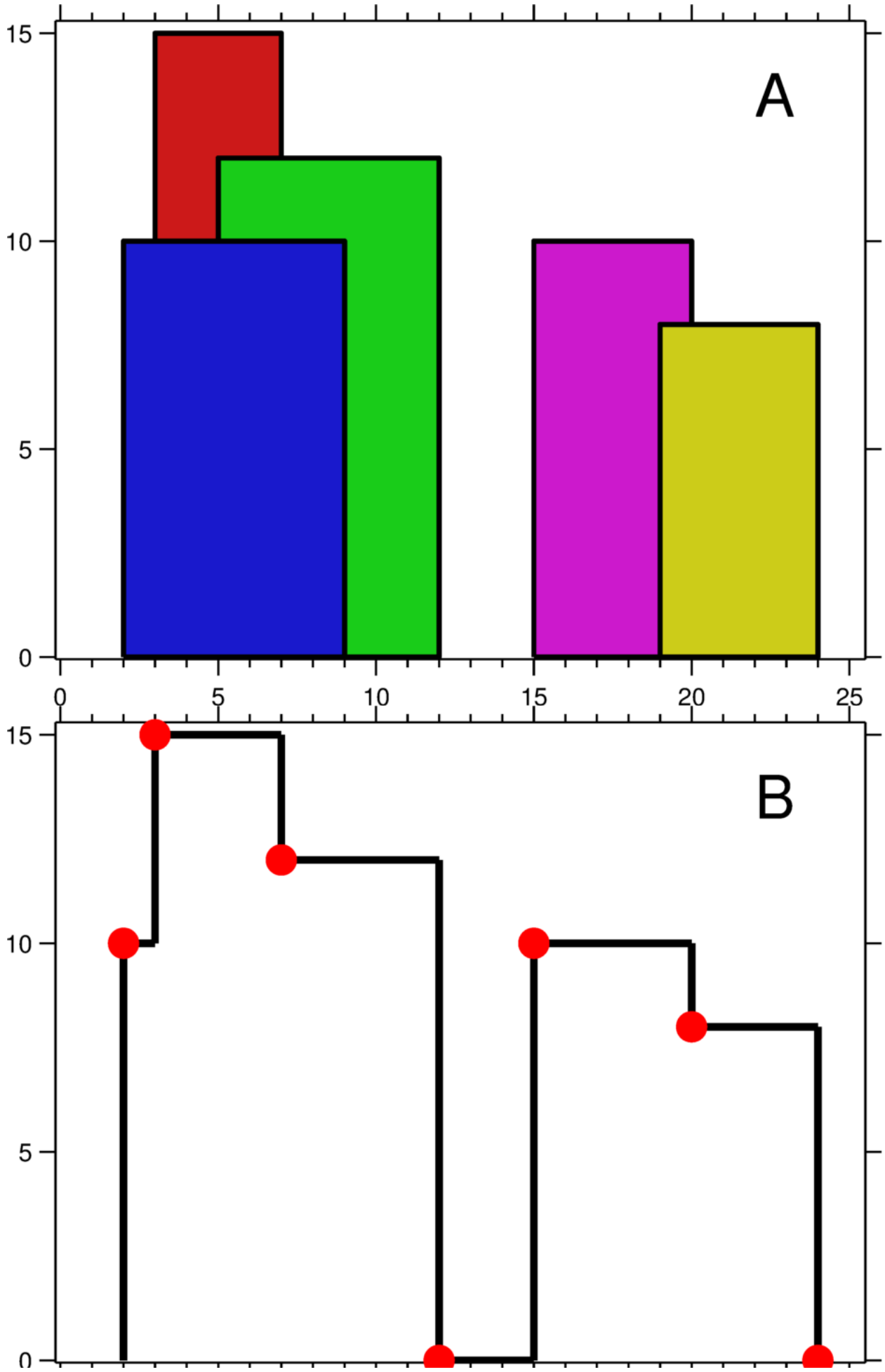
- 从最左侧柱子开始，寻找高度不低于该柱子的柱子，则能与它形成一片储水区的柱子，比如上图中的[1,0,2]，[2,1,0,1,3]，[2,1,2]形成了三个储水区，而存水量很容易计算，见代码注释
- 上图示例不包含所有情况，例如[6, 3, 5,3,2,4]中储水区[6,3,5]无法被发现，具体算法见代码注释

```
class Solution {
public:
    int trap(vector<int>& height) {
        if(height.size() == 0) return 0;
        int ans = 0;
        int idx = 0;
        while(idx < height.size() - 1){
            int i = idx+1;
            int cnt = 0;
            while(i < height.size() && height[i] < height[idx]){//向右找到第一个不矮于
当前柱子的
                cnt += height[idx] - height[i];//累计雨量
                i++;
            }
            if(i < height.size()){//找到了
                ans += cnt;
                idx = i;
            }
            else{//右侧柱子全比它矮，则找到当前柱子右侧的第一个“谷”，即例子中的[6,3,5]
                i--;
                int next = i;
                cnt = 0;
                i--;
                while(i > idx){
                    if(height[i] >= height[next]){//非谷
                        next = i;
                        cnt = 0;
                    }
                    else
                        cnt += height[next] - height[i];
                    i--;
                }
                ans += cnt;
                idx = next;
            }
        }
        return ans;
    }
};
```

216. 天际线问题（困难）

1. 题目描述

城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。现在，假设您获得了城市风光照片（图A）上**显示的所有建筑物的位置和高度**，请编写一个程序以输出由这些建筑物**形成的天际线**（图B）。


A
B
0
5
10
15
20
25

每个建筑物的几何信息用三元组 `[Li, Ri, Hi]` 表示，其中 `Li` 和 `Ri` 分别是第 i 座建筑物左右边缘的 x 坐标，`Hi` 是其高度。可以保证 `0 ≤ Li, Ri ≤ INT_MAX`，`0 < Hi ≤ INT_MAX` 和 `Ri - Li > 0`。您可以假设所有建筑物都是在绝对平坦且高度为 0 的表面上的完美矩形。

例如，图A中所有建筑物的尺寸记录为：`[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]`。

输出是以 `[ [x1,y1], [x2, y2], [x3, y3], ... ]` 格式的“**关键点**”（图B中的红点）的列表，它们唯一地定义了天际线。**关键点是水平线段的左端点。**请注意，最右侧建筑物的最后一个关键点仅用于标记天际线的终点，并始终为零高度。此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。

例如，图B中的天际线应该表示为：`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`。

说明:

- 任何输入列表中的建筑物数量保证在 `[0, 10000]` 范围内。
- 输入列表已经按左 `x` 坐标 `Li` 进行升序排列。
- 输出列表必须按 x 位排序。
- 输出天际线中不得有连续的相同高度的水平线。例如 `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` 是不正确的答案；三条高度为 5 的线应该在最终输出中合并为一个：`[...[2 3], [4 5], [12 7], ...]`

2. 官方解法——分治算法

方法：分治

想法

这个题是一道经典的分治算法题，通常如同归并排序一样。

我们依据下面的算法流程来求解分治问题：

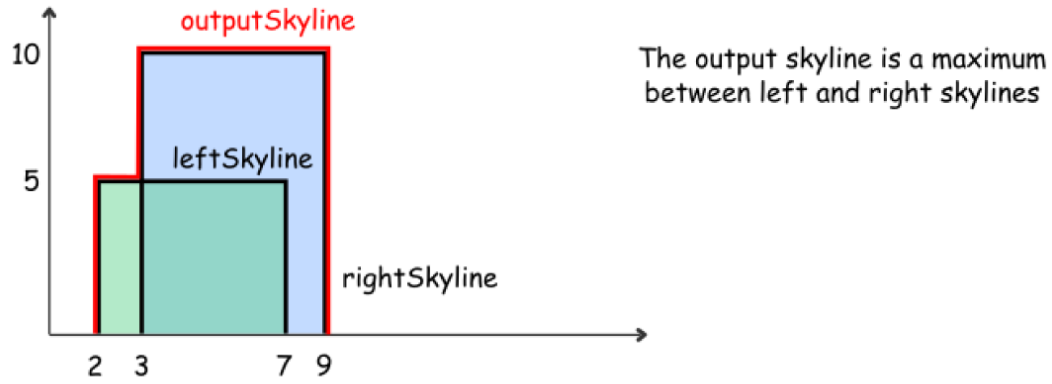
- 定义基本问题。
- 将问题分解为子问题，并递归地分别求解。
- 将子问题合并成原问题的解。

算法

求解 `n` 栋楼的天际线：

- 如果 `n == 0`：返回一个空列表
- 如果 `n == 1`：返回一栋楼的天际线
- `leftSkyline` = 求解前 n/2 栋楼的天际线。
- `rightSkyline` = 求解后 n/2 栋楼的天际线。
- 合并 `leftSkyline` 和 `rightSkyline`.

现在，让我们进一步讨论每一个步骤的细节：



我们这里用两个指针 `pR` 和 `pL` 分别记录两个天际线的当前元素，再用三个整数变量 `leftY`， `rightY` 和 `currY` 分别记录 `左` 天际线、 `右` 天际线和 `合并` 天际线的当前高度。

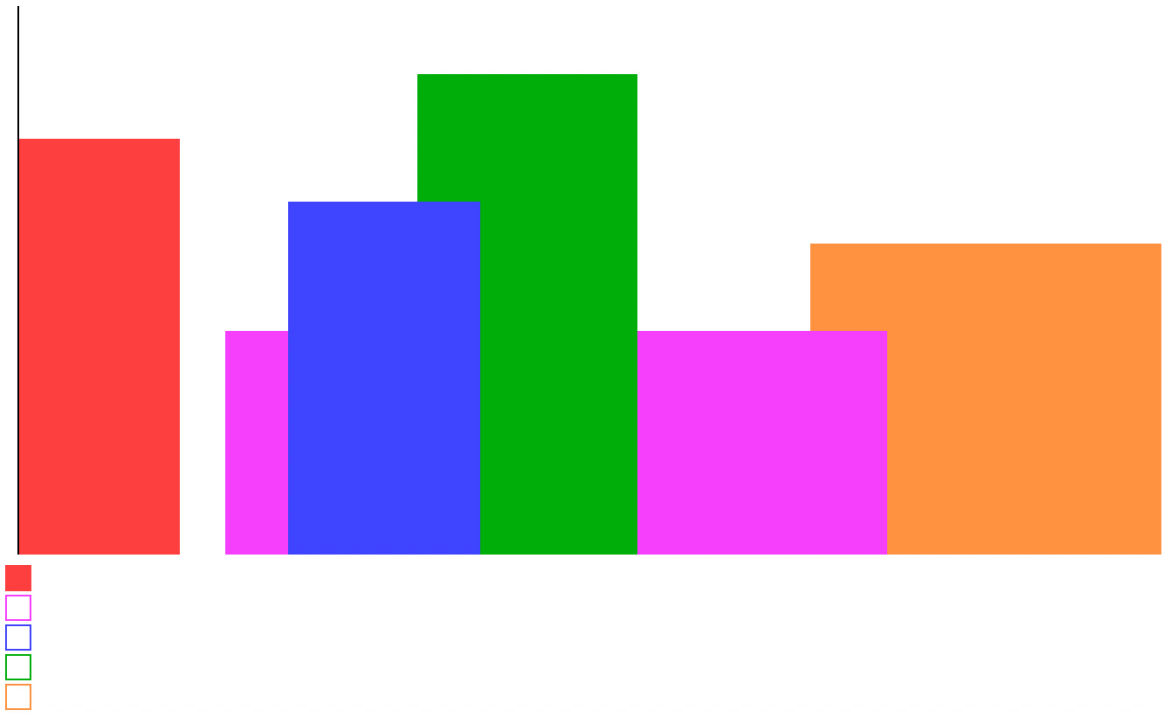
mergeSkylines (left, right) :

- currY = leftY = rightY = 0
- 当我们在一段两个天际线都可以见到的区域时（ `pR < nR` 且 `pL < nL` ）：
  - 选择 `x` 坐标较小的一个元素，如果它是左天际线的元素，就移动 `pL` 同时更新 `leftY` 。如果是右天际线，则移动 `pR` 且更新 `rightY` 。
  - 计算较高的高度作为当前点的高度： `maxY = max(leftY, rightY)` 。
  - 更新结果天际线： `(x, maxY)` ，前提是 `maxY` 不等于 `currY` .
- 如果左天际线还有元素没有被处理，也就是（ `pL < nL` ），则按照上述步骤处理这些元素。
- 如果右天际线还有元素没有被处理，也就是（ `pR < nR` ），则按照上述步骤处理这些元素。
- 返回结果天际线。

3. 扫描线法

使用扫描线，从左至右扫过。如果遇到左端点，将高度入堆，如果遇到右端点，则将高度从堆中删除。使用 last 变量记录上一个转折点。

可以参考下面的动图，扫描线下方的方格就是堆。



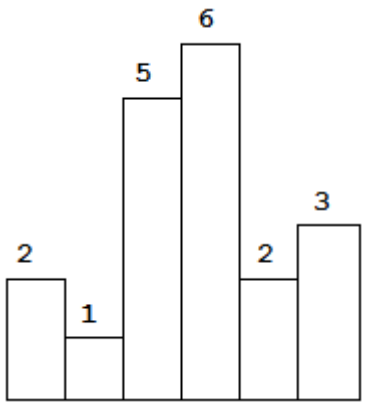
4. 是

84. 柱状图中最大的矩形（困难）

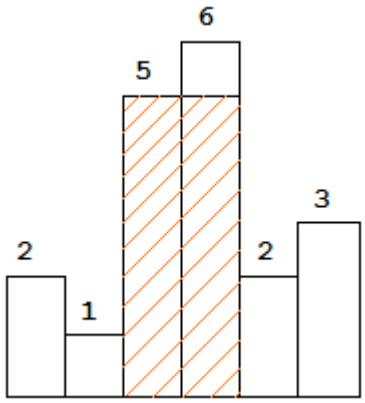
1. 题目描述

   给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

   求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

示例:

```
输入: [2,1,5,6,2,3]
输出: 10
```

2. 暴力实现——超时

两个柱子之间能形成的最大矩形为`min(各自的高度)*两者相隔宽度`，但这样两两遍历会有一个全为1的很长的case超时，因此可以加一个预判断：即所有的柱子都一样高时，直接可以计算最大矩形面积，这样虽然可以通过，但是有些取巧

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int len = heights.size();
        if(len == 0) return 0;
        else if(len == 1) return heights[0];
        int ans = 0;
        int minh = heights[0], maxh = 0;
        for(int i = 0; i < len; i++){
            minh = min(minh, heights[i]);
            maxh = max(maxh, heights[i]);
        }
        if(minh == maxh)//所有柱子一样高
            return minh*len;
        for(int i = 0; i < len; i++){
            int minh = heights[i];
            for(int j = i; j < len; j++){
                minh = min(minh, heights[j]);
                ans = max(ans, minh * (j-i+1));
            }
        }
        return ans;
    }
};
```

3. 单调栈

1. 单调栈分为单调递增栈和单调递减栈
    11. 单调递增栈即栈内元素保持单调递增的栈
    12. 同理单调递减栈即栈内元素保持单调递减的栈
2. 操作规则（下面都以单调递增栈为例）
    21. 如果新的元素比栈顶元素大，就入栈
    22. 如果新的元素较小，那就一直把栈内元素弹出来，直到栈顶比新元素小
3. 加入这样一个规则之后，会有什么效果
    31. 栈内的元素是递增的
    32. 当一个栈内元素被弹出来时，说明新元素是第一个比该元素小的元素
    33. 因为栈内元素是递增的，所以出栈的元素前面第一个比该元素小的就是新的栈顶

思路

1. 对于一个高度，如果能得到向左和向右的边界
2. 那么就能对每个高度求一次面积
3. 遍历所有高度，即可得出最大面积

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int ans = 0;
        vector<int> st;//单调栈，按高度递增存储柱子的索引
        //添加左右0边界，可以将原边界也统一运算
        heights.insert(heights.begin(), 0);
        heights.push_back(0);
        for (int i = 0; i < heights.size(); i++) {
            while (!st.empty() && heights[st.back()] > heights[i]) {//i号柱子为当前栈顶高度的右边界
                int cur = st.back();//为cur号柱子找到左右边界
                st.pop_back();
                int left = st.back() + 1;//cur号柱子左侧第一个比它矮的柱子的右边那根柱子
                int right = i - 1;//cur号柱子右侧第一个比它矮的柱子的左边那根柱子
                ans = max(ans, (right - left + 1) * heights[cur]);
            }
            st.push_back(i);
        }
        return ans;
    }
};
```