暑假 LintCode 数据结构做题报告

专业班级：19C199

学生姓名：李子强

学生学号：20191002184

指导老师：张剑波

目 录

一、LintCode 做题报告

1.1、对于通过题目的截图

提交时间	</>我的代码	面试真题
8 天前	Accepted	摊平嵌套的列表
8 天前	Accepted	不超过K的最大矩形元素之和
8 天前	Accepted	数据流中第一个唯一的数字
8 天前	Accepted	二叉树的锯齿形层次遍历
9 天前	Accepted	移掉K位数字
9 天前	Accepted	小行星的碰撞
9 天前	Accepted	移除多余字符
9 天前	Accepted	字符串解码
9 天前	Accepted	有效的括号序列
9 天前	Accepted	移除K位
10 天前	Accepted	132 模式
11 天前	Accepted	简化路径
11 天前	Accepted	将表达式转换为逆波兰表达式
11 天前	Accepted	最长有效括号
11 天前	Accepted	表达式求值
11 天前	Accepted	表达式求值
11 天前	Accepted	逆波兰表达式求值
13 天前	Accepted	LRU缓存策略
13 天前	Accepted	LRU缓存策略
13 天前	Accepted	交换链表当中两个节点
13 天前	Accepted	回文链表
13 天前	Accepted	旋转链表
15 天前	Accepted	链表求和
15 天前	Accepted	删除排序链表中的重复数字 II
15 天前	Accepted	有序链表转换为二叉搜索树
15 天前	Accepted	带环链表 II
15 天前	Accepted	合并k个排序链表
15 天前	Accepted	复制带随机指针的链表
16 天前	Accepted	重排链表
16 天前	Accepted	链表排序
16 天前	Accepted	链表排序
16 天前	Accepted	翻转链表 II
16 天前	Accepted	主元素 III
21 天前	Accepted	链表划分
23 天前	Accepted	搜索区间
23 天前	Accepted	买卖股票的最佳时机 III
23 天前	Accepted	丢失的第一个正整数
24 天前	Accepted	连续子数组求和 II
24 天前	Accepted	三数之和
24 天前	Accepted	最大子数组差
24 天前	Accepted	最大子数组 II
24 天前	Accepted	矩阵幂级数
24 天前	Accepted	最短子数组 II
25 天前	Accepted	找出第 k 小的距离对
1 个月前	Accepted	最大子数组 V
1 个月前	Accepted	最大矩形
1 个月前	Accepted	直方图最大矩形覆盖
1 个月前	Accepted	两个排序数组的中位数
1 个月前	Accepted	最大子数组 III

1.2、做题总结

LintCode 的这种方式还是挺有意思的，只要设计一个函数就行，不用在意其他的细节。
动态的使用非常广泛，这方面确实是以前没有太注意的。
题目真的很有意思，不算是思维题，但是设计的方法都很巧妙。二分或者归并的应用非常广泛。
给的数据很强，各种特殊情况都要考虑，不然肯定会 WA。
通过率<40%的题目无论难度，基本都带很多的坑，或者卡时间，甚至能卡内存。

二、数据结构做题记录

2.1、数组

LintCode-42 最大子数组 II

/*首先分成两部分，那么为了枚举这个分界点，我们就不能再进行复杂的操作了。
分界线以左的部分我们要求一个，分界线以右的部分也要求一个，分别用一个循环就能解决，具体怎么解决，dp 经典求最大连续子序列和。
需要注意的是，在求分界线以右的部分时，由于一般的 dp 是通过确立终点，求前 n 项的子序列和，那么这里应该反向 dp。
最后再来一个循环，枚举这个分界点，找到最大值。
加上前缀和的话，复杂度是 O(3n)的，当然常数忽略。
吐槽一下，这个细节的处理真的麻烦，因为题目是 vector，要是直接下标从 1 开始就舒服多了。*/

```
class Solution {
public:
    /*
     * @param nums: A list of integers
     * @return: An integer denotes the sum of max two non-overlapping subarrays
     */
    int maxTwoSubArrays(vector<int> &nums) {
        // write your code here
        int n=nums.size();
        vector<int>l(n+1),r(n+1);
```

```
        int ma = -1e9;
        int res = 0;
        for(int i=1;i<n;i++){
            res = max(res+nums[i-1], nums[i-1]);
            if(res>=ma)ma = res;
            l[i]=ma;
        }
        ma=-1e9,res=0;
        for(int i=n-1;i>=1;--i){
            res = max(res+nums[i], nums[i]);
            if(res>=ma)ma = res;
            r[i]=ma;
        }

        int ans=-1e9;
        for(int i=1;i<=n-1;++i)
            ans=max(ans,l[i]+r[i]);
    return ans;
    }
};
```

LintCode-43 最大子数组 III

/*给定一个整数数组和一个整数 k，找出 k 个不重叠子数组使得它们的和最大。每个子数组的数字在数组中的位置应该是连续的。返回最大的和。

因为要找 k 个不重叠子数组，那么区间长度和位置都是不确定的，我们可以用 dfs 搜索去一步步尝试、回溯，但是这题也是非常明显可以用 dp 的。
我们先假设 dp 为二维，dp[i][j]表示将前 i 个数分成 j 个子数组的最大和，当我们在求解 dp[i][j]时，明显选择顺推更加方便的。
此时我们要确定状态转移方程，以 i 为递推的核心不太合适，因为根据将前 i-1 个数分成 j 个子数组的和难以写出递推方程。那么换个思路，由于 dp[x][y](x<i,y<j)是我们在顺推 dp 过程已知的，将前 i 个数分成 j 个子数组=max(将前 x 个数分成 j-1 个子数组+从第 x+1 个数到 i 的和），用个循环来枚举即可。
这样状态转移方程就出来了，先计算出 num[x][y]（num[y]-num[x]），表示从 x->y 的和，再列出转移方程 dp[i][j] = max(dp[x][j-1]+num[x+1][i])，枚举 x 即可。

当然这题还有一个问题，计算出 x->y 的和，即 num[x][y]（num[y]-num[x]）。区间查询可以用树状数组或者线段树，建树+查询的复杂度应该是 O（nlogn），不过这里还是直接 for 循环累加吧，因为比较简单。*/

```
class Solution {
public:
    /**
```

```
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping subarrays
     * @return: An integer denote the sum of max k non-overlapping subarrays
     */
    int maxSubArray(vector<int>& nums, int k) {

        vector<vector<int> > dp(nums.size() + 1, vector<int>(k + 1, INT_MIN));
        for (int i = 0; i < nums.size(); i++)
            dp[i][0] = 0;
        for (int i = 1; i <= nums.size(); i++)
        {
            for (int j = 1; j <= min(i, k); j++)
            {
                int endMax = 0,ma = INT_MIN;
                for (int x = i; x >= j; x--)
                {
                    endMax = max(nums[x - 1], nums[x - 1] + endMax);
                    ma = max(ma, endMax);
                    dp[i][j] = max(dp[i][j], dp[x - 1][j - 1] + ma);
                }
            }
        }
        return dp[nums.size()][k];
    }
};
```

LintCode-45 最大子数组差

/*因为做过 42 题，如果是求区间最大，那么很轻松，这里的结果带个绝对值的话，那可以分类讨论。
我们分别求出以 i 为边界的 A1、A2、B1、B2 四个数组，本别代表前 i 个数的区间最大和、最小和，后 i 个数的区间最大和、最小和。
最后因为差值的绝对值最大，那么 A1、B2 和 A2、B1 组合一下，求解即可。
时间复杂度是 O(6n)，空间复杂度是 O(4n)。*/

```
class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: An integer indicate the value of maximum difference between two substrings
     */
        int maxDiffSubArrays(vector<int> &nums) {
        // write your code here
        int n=nums.size();
```

```
        vector<int>l1(n+1),r1(n+1),l2(n+1),r2(n+1);
        int ma = -1e9,mi=1e9;
        int res = 0;
        for(int i=1;i<n;i++){
            res = max(res+nums[i-1], nums[i-1]);
            if(res>=ma)ma = res;
            l1[i]=ma;
        }
        ma=-1e9,res=0;
        for(int i=n-1;i>=1;--i){
            res = max(res+nums[i], nums[i]);
            if(res>=ma)ma = res;
            r1[i]=ma;
        }
        res=0;
        for(int i=1;i<n;i++){
            res = min(res+nums[i-1], nums[i-1]);
            if(res<=mi)mi = res;
            l2[i]=mi;
        }
        mi=1e9,res=0;
        for(int i=n-1;i>=1;--i){
            res = min(res+nums[i], nums[i]);
            if(res<=mi)mi = res;
            r2[i]=mi;
        }
        int ans=-1e9;
        for(int i=1;i<=n-1;++i)ans=max(ans,abs(l1[i]-r2[i]));
        for(int i=1;i<=n-1;++i)ans=max(ans,abs(l2[i]-r1[i]));
        return ans;
    }
};
```

LintCode-57 三数之和

/*这题还是挺有意思的，三数之和为 0，暴力解枚举三遍，复杂度略小于 O(n3)，因为不用重复枚举。
换种思路，因为涉及到三个数，二分之类的方法估计也只能降低到 O(n2logn)左右，那么一般可以用指针移动或者滑块窗口的方式，这里可以用指针移动。
要让搜索不具有盲目性，肯定要先排序，二分都要排序。。。直接 sort，排序最坏的情况复杂度也就 O(n2)，当然，这里用有序的话可以方便跳过相等的元素。
然后还是，参考了答案。。太菜了自己。
题解还是 tql，2sum 还是简单的，O(nlogn)快排，然后首尾各一个指针，a+b<0 则尾指针左移，反之则首指针右移，这样就实现 O(n)了，这个还是比较简单的。

问题是 3sum，其实只要枚举一下 a+b=-c 的-c 就行了，也就转换成了 2sum 的问题，复杂度也就在 O(n2)的样子。还是太蠢了，要是给出 2sum 的 O(n)那我应该能知道怎么做，还是见识不够。*/

```
class Solution {
public:
    /**
     * @param numbers : Give an array numbers of n integer
     * @return : Find all unique triplets in the array which gives the sum of zero.
     */
    vector<vector<int> > threeSum(vector<int> &nums) {
        vector<vector<int> > result;

        sort(nums.begin(), nums.end());
        for (int i = 0; i < nums.size(); i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            // two sum;
            int start = i + 1, end = nums.size() - 1;
            int target = -nums[i];
            while (start < end) {
                if (start > i + 1 && nums[start - 1] == nums[start]) {
                    start++;
                    continue;
                }
                if (nums[start] + nums[end] < target) start++;
                else if (nums[start] + nums[end] > target) end--;
                else {
                    vector<int> triple;
                    triple.push_back(nums[i]);
                    triple.push_back(nums[start]);
                    triple.push_back(nums[end]);
                    result.push_back(triple);
                    start++;
                }
            }
        }

        return result;
    }
};
```

LintCode-61 搜索区间

/*给定一个包含 n 个整数的排序数组，找出给定目标值 target 的起始和结束位置。
又是头疼的 O(logn)题目，既然排序排好了，那就二分查找。查到某个位置后，尝试向左向右拓展就行。
不过需要注意的是，因为可能不存在 target，因此我们在这里采取的方式是两次二分，分别找左、右，这样可以避免因为二分查找不到位置而无法拓展的情况。*/

```
class Solution {
public:
    /**
     * @param A: an integer sorted array
     * @param target: an integer to be inserted
     * @return: a list of length 2, [index1, index2]
     */
vector<int> searchRange(vector<int> &A, int target) {
    // write your code here
    vector<int> ans;
    int ansl = -1;
    int l = 0, r = A.size() - 1;
    while(l <= r) {
       int mid = l + (r - l) / 2;
       if (A[mid] > target)
          r = mid - 1;
       if (A[mid] < target)
          l = mid + 1;
       if (A[mid] == target)
       {
          ansl = mid;
          r = mid - 1;
       }
    }

    int ansr = -1;
    l = 0, r = A.size() - 1;
    while(l <= r) {
       int mid = l + (r - l) / 2;
       if (A[mid] > target)
          r = mid - 1;
       if (A[mid] < target)
          l = mid + 1;
       if (A[mid] == target)
          {
          ansr = mid;
```

```
            l = mid + 1;
        }
    }
    ans.push_back(ansl);
    ans.push_back(ansr);
    return ans;
  }
};
```

LintCode-65 两个排序数组的中位数

/*这题非常简单，但是复杂度要求 O(log(n+m))，暴力肯定不行了。
我们学过快速排序、归并排序这种 O(nlogn)的算法（数据比较合理的情况下）。那初步想法是将两个数组合并，直接 sort。。。不过合并两个数组比较麻烦，虽然 vector 也行，但是时间复杂度怎么也肯定超了。
其次想到的就是利用快排、归并的原理，自己写个指针移动，但是 O(logn)太狠了，没写出来。
最后想到是二分（其实是从归并想到的），一步步缩减范围，不过太菜了，感觉很麻烦，还是没写出来。
最后的最后去网上查了一下这题，带了个关键字二分，结果还真有。那么我就借用下 zxzxy1988 这位大佬博主的写法，假装是自己写出来的吧。（求放过，不查真的不会写啊）

中位数比较特殊，这里用了更加通用的方法，寻找合并后的第 k 个数。
首先取出 A、B 数组的的一个数，由于两个数组长度不一样，因此我们选择一个 k，假设两个数组的长度都大于 k/2，这样就可以取出 A[k/2-1]和 B[k/2-1]。
将取出的两个数进行比较，假设 A[k/2 - 1] < B[k/2 - 1]，那么当 A 和 B 合并之后，A 的下标从 0 到 k/2 - 1 之间的数都不可能是第 k 大的数，剔除掉这部分的数，再次重复进行这个操作。
在这样的循环下，k 的值在不断减半，最终一定会来到 k=1。
k=1 时，比较 A[0]，B[0]，哪个小哪个就是答案。
当前，之前只考虑了 A[k/2 - 1] < B[k/2 - 1]的情况。对于 A[k/2 - 1] > B[k/2 - 1]，剔除 B 的下标从 0 到 k/2 - 1 之间的数。对于 A[k/2 - 1] = B[k/2 - 1]，那么这两个数就是第 k 大的数，也直接返回就好。
还有种情况没考虑，某个数组长度不到 k/2，如果如果本次剔除的正好是这个数组，直接在下一次返回另一个数组的第 k 个数即可。
关于中位数，要分奇偶讨论，奇数取中间，偶数取中间两个的平均数。*/

```
class Solution {
    /*
     * @param A: An integer array
     * @param B: An integer array
     * @return: a double whose format is *.5 or *.0
     */
public:
```

```
    int findKth(vector<int>& A, int idx1, vector<int>& B, int idx2, int k) {
        // 如果 idx1 >= A.length，那就直接返回 B[idx2]起第 k 小的数即可；
        // 如果 idx2 >= B.length 亦然
        if (idx1 >= A.size()) {
            return B[idx2 + k - 1];
        }
        if (idx2 >= B.size()) {
            return A[idx1 + k - 1];
        }
        // 如果 k = 1，直接返回 A 与 B 开头数较小者
        if (k == 1) {
            return min(A[idx1], B[idx2]);
        }

        int halfKOfA = idx1 + k / 2 - 1, halfKOfB = idx2 + k / 2 - 1;
        if (A[halfKOfA] == B[halfKOfB])
            return A[halfKOfA];
        if (halfKOfA >= A.size() || A[halfKOfA] > B[halfKOfB]) {
            return findKth(A, idx1, B, halfKOfB + 1, k - k / 2);
        }
        if (halfKOfB >= B.size() || A[halfKOfA] < B[halfKOfB]) {
            return findKth(A, halfKOfA + 1, B, idx2, k - k / 2);
        }

        return 0;
    }
    double findMedianSortedArrays(vector<int>& A, vector<int>& B) {
        // write your code here
        int sumLen = A.size() + B.size();
        if (sumLen % 2 != 0) {
            return findKth(A, 0, B, 0, sumLen / 2 + 1);
        }
        else {
            return (findKth(A, 0, B, 0, sumLen / 2) +
                findKth(A, 0, B, 0, sumLen / 2 + 1)) / 2.0;
        }
    }

};
```

LintCode-122 直方图最大矩形覆盖

/*这题是单调栈的经典（我记得悬线法也行），最后的重新入栈这步操作当时让我自闭了，后来见了更复杂题目的重新入栈操作，我只能称为，神仙操作。

虽然我手中已经有模板了，但是这并不妨碍我再水一遍
单调栈就和它名字一样，是单调的栈，所以这题放在数组这块，可能是想让我用悬线法吧。

这里选用单调递减栈，递减是出栈顺序，从栈顶往下看，它是递减的。
从栈中取出当前的栈顶，尝试去向两边拓展，如果遇到的数比自己大，那么继续拓展，反之则停止。
对于栈来说，我们查看栈顶的大小，如果栈顶比较小，那么就可以拓展，也就是尝试去入栈。
如果不能拓展了，那么我们取出栈顶，计算一下刚才拓展的矩形面积，并且对最大面积进行更新。
面积的算法也很简单，高就是出栈的这个数对应的高度，宽就是停止拓展的地方与当前出栈的数的差。
对于第一次，我们算出了停止拓展左边那个矩形的面积，但是这不一定是最大的，因为当前栈中对应的图的高度是递增的（从左往右），如果当前栈顶处的数对应的高度比较大，就继续出栈，以该栈顶的高度为高，宽度+1，算出矩形面积。
下面是神仙操作，对于本题来说还是非常容易理解的。当当前栈顶小于之前停止拓展的数时，我们将最后一次出栈的栈顶压回，同时修改高度为当前停止拓展的数对应的高度。
这样做的目的是为了获取宽是连续且比较长的、但是高比较短的矩形面积，这可能也是我们的答案。（语文太差，解释不清楚，我当时看的也晕）*/

```
class Solution {
public:
    /**
     * @param height: A list of integer
     * @return: The area of largest rectangle in the histogram
     */
    int largestRectangleArea(vector<int>& heights) {
        heights.push_back(-1);
        stack<int> st;
        int ret = 0, top;
        for (int i = 0; i < heights.size(); i++)
        {
            if (st.empty() || heights[st.top()] <= heights[i])
            {
                st.push(i);
            }
            else
            {
                while (!st.empty() && heights[st.top()] > heights[i])
                {
                    top = st.top();
                    st.pop();

                    int tmp = (i - top) * heights[top];
                    if (tmp > ret)
```

```
                    ret = tmp;
                }
                st.push(top);
                heights[top] = heights[i];
            }
        }
        return ret;
    }
};
```

LintCode-151 买卖股票的最佳时机 III

/*假设你有一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。设计一个算法来找到最大的利润。你最多可以完成两笔交易。
这感觉是 dp，但是这个状态有点难啊，如果是只买卖一次的话，那么状态转移方程应该是……编不出来，这状态太难找了。
dp 还是太弱了，等下再找两题补补。

根据九章算法的解法，将买卖的过程分成第一次买前，第一次买后持有，第一次卖后第二次买前，第二次买后持有，第二次卖后这五个状态。
定义 dp[i][j]为前 i 天结束后，在第 j 个阶段的最大获利。则阶段 1、3、5，即手中没有股票的状态下， dp[i][j] = max(dp[i-1][j], dp[i-1][j-1] + P[i-1] - P[i-2])，可能是前一天的同一阶段不变，也可能是前一天的上个阶段买入。
阶段 2、4 即持有状态下，dp[i][j] = max(dp[i-1][j-1]，dp[i-1][j] + P[i-1] - P[i-2]，也就是前一天的上一阶段买入，或者什么都不做，但是股票的价格会变动，因此今天的利润应该是前一天的同一阶段加上价格变动差值。
值得一提的是，这里 dp 是指当前的总最大利润，也就是说卖出股票的时机不限（可以在买入的同一天的晚些时候）。
不难，但是这样的思想确实在 dp 我还没遇到太多，多状态，学习学习。
然后拿动态创建内存就段错误，我明明是在用完就析构了……看来还是创建静态的，让程序自己析构比较好。*/

```
class Solution {
public:
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    int maxProfit(vector<int> &prices) {
        int n = prices.size();
        if (n == 0)
            return 0;
     int dp[n + 1][5];
        dp[0][0] = 0;
```

```
        for (int k = 1; k <= 5; k++)
            dp[0][k] = -100000000;

        dp[0][1] = 0;
        for (int i = 1; i <= n; i++) {
            // 手中未持有股票
            for (int j = 1; j <= 5; j += 2) {
                // 前一天也未持有
                dp[i][j] = dp[i - 1][j];
                if (j > 1 && i > 1 && dp[i - 1][j - 1] !=-100000000) {
                    //                                  前一天持有，今天卖了获利。
                    dp[i][j] =max(dp[i][j], dp[i - 1][j - 1] + prices[i - 1] - prices[i - 2]);
                }
            }
            // 手中持有股票
            for (int j = 2; j <= 5; j += 2) {
                //前一天未持有，今天买进
                dp[i][j] = dp[i - 1][j - 1];
                if (i > 1 && dp[i - 1][j] != -100000000) {
                    //                                  前一天持有了，计算今天的利润
                    dp[i][j] =max(dp[i][j], dp[i - 1][j] + prices[i - 1] - prices[i - 2]);
                }
            }
        }

        int res = 0;
        for (int j = 1; j <= 5; j += 2)
            res = max(res, dp[n][j]);

        return res;
    }
};
```

LintCode-189 丢失的第一个正整数

/*乍一看这题还挺吓人的，24%的通过率和 O(n)复杂度的要求。
这题其实很简单，利用离散化的思想，我在提交后看了下九章算法的题解，感觉做麻烦了，下面给出我的解法（不过我因为还创建了一个 vector 的对象，因此速度还是相对比较慢的）。
给个数组，找其中没有出现过的最小正整数。我们知道，A 的长度为 n 时，最多包含 n 个正整数，而我们要求的答案就在 1~n+1 之间，这是显然的。
当 A[i]为正数时，我们知道 A[i]是不能取了，我们新创建一个 vector 对象 B，标记 B[A[i]]，假如 A[i]=3，B[3]就被标记了，不可能是我们要的答案。

但是这存在一个问题，当 A[i]太大时，会超过复杂度 O(n)，或者 B[A[i]]会越界。结合之前分析的结果在 1~n+1 之间，那么超过 n 的就不用考虑了。
最后跑一遍 B 数组，遇到的第一个没有被标记的就是答案。当然，跑完数组还没遇到，就返回 n+1。*/

```
class Solution {
public:
    /**
     * @param A: An array of integers
     * @return: An integer
     */
    int firstMissingPositive(vector<int> &A) {
        // write your code here
        vector<int>B;
        int n=A.size();

        for(int i=0;i<n;++i)B.push_back(0);
        B.push_back(0);

        for(int i=0;i<n;++i)
            if(A[i]>0&&A[i]<=n)B[A[i]]=1;

        for(int i=1;i<=n;++i)
            if(B[i]==0)return i;

            return n+1;

    }
};
```

LintCode-403 连续子数组求和 II

/*本来看到这题目感觉不想做的，但是这题的通过率只有 20%，瞬间有了好奇。
本质还是找最大子数组和，只不过这次是首尾相连了。
因为做过很多字符串的题目，对于首尾相连下意识的想法就是，重复该数组，使其扩展成原来的两倍，通过卡 dp 的长度为 n，但是因为要限制 dp 的长度，那么复杂度肯定就不止 O(n)了。
去搜了下循环数组有没有什么解决方案，无意间看到了 51Nod 1050 的题解，然后去看了下九章算法给的参考答案，自己还是太弱了。
这个题目的最大字段和有两种可能，普通数组的最大和、首位相连的最大和。
对于第二种情况，是因为数组中间存在某段和为负值，且绝对值很大，这点其实很好想，因为如果都是正数，或者掺杂了大负数，那么理应会从头到尾。
其实这个解释并不好，因为没有个很好的逻辑推理过程。我们可以从另一个角度理解，在这

种情况下，最大的循环部分子区间包括数组的后半段和前半段，而空出了中间的一段。
为什么会空出中间这段？从结果来分析，无论是加上中间这段的左、右端点，亦或是左子区间、右子区间，都会让总和变小，也就是左、右任意子区间恒负。同时选出的循环部分，满足左、右任意子区间恒正。
那么我们可以得到一个结论：中间这段无论是向左还是向右拓展都会增大，同时向左、向右回缩也都会增大(因为回缩的区间恒负)。那么就证明出了中间部分为数组中最小子区间。
类似的，循环部分无论是向左还是向右拓展都会减小，同时向左、向右回缩也都会减小(因为回缩的区间恒正)。该部分就是第二种情况所求。

这种证明的方式某些题目看多了，连我都会了。。。代码的写法还是一如既往地非常舒服和巧妙，把指针移动演示的真是淋漓尽致，学习学习。*/

```
class Solution {
public:
    /**
     * @param A an integer array
     * @return    A list of integers includes the index of
     *                the first number and the index of the last number
     */
    vector<int> continuousSubarraySumII(vector<int>& A) {
        // Write your code here
        vector<int> result;
        result.push_back(0);
        result.push_back(0);

        int total = 0;
        int len = A.size();
        int start = 0, end = 0;
        int sum = 0;
        int ans = -0x7fffffff;
        for (int i = 0; i < len; ++i) {
            total += A[i];
            if (sum < 0) {
                sum = A[i];
                start = end = i;
            }
            else {
                sum += A[i];
                end = i;
            }
            if (sum >= ans) {
                ans = sum;
                result[0] = start;
                result[1] = end;
```

```
            }
        }
        sum = 0;
        start = 0, end = -1;
        for (int i = 0; i < len; ++i) {
            if (sum > 0) {
                sum = A[i];
                start = end = i;
            }
            else {
                sum += A[i];
                end = i;
            }
            if (total - sum > ans && !(start == 0 && end == len-1)) {
                ans = total - sum;
                result[0] = (end + 1) % len;
                result[1] = (start - 1 + len) % len;
            }
        }
        return result;
    }
};
```

LintCode-510 最大矩阵

/*悬线法经典，感谢王知琨大神的论文让我入门。悬线法明明非常好用，但还是好多人喜欢用单调栈啊。

我们枚举每一个有效极大子矩阵，再根据悬线递推 left[i][j]、right[i][j]、height[i][j]三个参数即可。
这题是模板题，悬线法的原理其实是比较容易理解的，当然因为原理的核心部分相当于个 dp 的过程，大部分 dp 都是大佬的艺术，只会读，不会写。
算法原理不写了，让我快乐水一道吧。
结果段错误，内存爆了。思考了一下感觉应该是因为三个二维数组太大了，也不好减小内存，所以直接换单调栈。
怪不得这么多人喜欢单调栈。
模板，对每一行进行求解，不解释了。*/

```
class Solution {
public:
    /**
     * @param matrix a boolean 2D matrix
     * @return an integer
```

```
 */
int maximalRectangle(vector<vector<bool> >& matrix) {
        // Write your code here
        int res = 0;
        if (matrix.empty()) {
                return res;
        }
         int m = matrix.size();
         int n = matrix[0].size();

        vector<int> temp(n, 0);
        vector<vector<int>> height(m, temp);

        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                        if (i == 0 && matrix[i][j]) {
                                height[i][j] = 1;
                                continue;
                        }
                        if (matrix[i][j]) {
                                height[i][j] = height[i - 1][j] + 1;
                        }
                }
        }
        for (int i = 0; i < m; i++) {
                res = max(res, largestRectangleArea(height[i]));
        }
        return res;
}

int largestRectangleArea(vector<int>& height) {
        // write your code here
        stack<int> S;
        height.push_back(0);
        int sum = 0;
        for (int i = 0; i < height.size(); i++) {
                if (S.empty() || height[i] > height[S.top()]) S.push(i);
                else {
                        int tmp = S.top();
                        S.pop();
                        sum = max(sum, height[tmp] * (S.empty() ? i : i - S.top() - 1));
                        i--;
                }
        }
```

```
            return sum;
        }
};
```

LintCode-621 最大子数组 V

/*这简单的题目，这尴尬的通过率，暴力肯定是 TLE。
初步想法，树状数组求前缀和，枚举 k1~k2，复杂度大概在 O((k2-k1)*nlogn)~O((k2-k1)*n2logn)。。
非常讨厌的一个地方是 vector 从 0 开始，树状数组都是从 1 开始的，因此对于函数部分重写了一点。
复杂度还是大了，测试 1106ms 通过了 93%，感觉就差一点，但是就算是降低了一点复杂度过了，也有点过于勉强，那么应该换种思路。
顺便一提，我试了一下暴力，1107ms 通过了 93%，看来这个树状数组的建树过程还是太麻烦了。
去借鉴了一下网上的做法，没有解析，看来好久才看懂。因为太过习惯用树状数组求前缀和的问题了，感觉 O(nlogn)已经很好了，现在又被打脸了。
这题降低复杂度的关键在于 k1~k2 的枚举，这里用了队列来存储信息。我查到的解法用的是 java，里面的 queue 应该是 deque，说实话这可能是我第一次遇到用 deque 的题（好像以前用过？）。

滑动窗口算法，我还没有遇到过，可以进我的 note 了，其实本题的侧重点明显在搜索是哪个区间上。
算法本身也没什么好说的，利用滑块也很好想象，deque 更是非常契合首入尾出的操作。
不过对于本题来说，因为是 k1~k2 一个范围，因此我们在滑块滑动的基础上，需要进行一些升级。
当 i=k1 时，此时我们就获得了一个长度 k1 的滑块，我们计算这个滑块的值。同时继续尾插 0。
当 i=k1+1 时，我们先判断是否已经超出 k2 的范围了，超出则 pop 队首，就转化成了和 i=k1 一样的情况。如果没有超出范围，这个时候我们比较 sum[q.back()]和 sum[i - k1]，当大于时，我们知道滑块即将碰面的元素是个负数，因此以刚才尾插的 0 开头的长度为 k1+1 的滑块的值比长度为 k1 的滑块的值小，那么这个新滑块就不可能是最大值了，因此我们将这个 0 弹出，之后在计算滑块值就不用计算它的了。
如此循环即可，不得不说这个算法非常的巧妙，队列中存储的是滑块开头的下标，又剔除了不可能的情况，因此在 k2 限制队首的前提下，用当前指针的位置减去队首对应的前缀和就是区间值了。

说实话看懂代码并不容易，花了很多时间，就勉强算做我写的吧。。。
（这题明明比之前做的几题难，这个 k1~k2 这么要命通过率都有 31%，自己还是太弱了）
*/

```
class Solution {
public:
```

```
    /**
     * @param nums: an array of integers
     * @param k1: An integer
     * @param k2: An integer
     * @return: the largest sum
     */
    int maxSubarray5(vector<int>& nums, int k1, int k2) {
        // write your code here
        if (nums.size() < k1) {
            return 0;
        }

        int ma = -1e9;;

        vector<int>sum(nums.size() + 1);
        //队列:用于最小和值的下标队列（保留 i-k1~i-k2 之间的和值，并将最小和值保留在队首）
        deque<int>q;

        for (int i = 1; i <= nums.size(); i++) {
            //一直求和，和=前一元素为止的和+尾元素
            sum[i] = sum[i - 1] + nums[i - 1];

            //最大窗口移动时，将队首移除队列
            if (!q.empty() && i - q.front() > k2) {
                q.pop_front();
            }

            if (i >= k1) {
                while (!q.empty() && sum[q.back()] > sum[i - k1]) {
                    //如果前面的和值大于 i-k1 下标和值，则将 i-k1 替换进去，否则直接将 i-k1 加入队列
                    q.pop_back();
                }
                q.push_back(i - k1);
            }

            // [i - k2, i - k1]
            if (!q.empty()) {
                ma = max(ma, sum[i] - sum[q.front()]);
            }
        }

        return ma;
```

```
    }
};
```

LintCode-947 矩阵幂级数

/*矩阵幂级数。。。根据我对线代的了解，矩阵如果不特殊的话，不同的幂之间应该没有什么联系。因此直接跑个循环，分别求出每个幂级数就行，根据通过率，首先把这种想法 pass。这是个很经典的题目了，优化的关键在幂级数（毕竟学过矩阵的，知道它没法动）。
作为一道数学题，那么看到这种式子想做的肯定是提出 A^k 出来，那么如果我们将它分别长度相同的两部分（奇数就舍弃一项），那么将后一半分提出个 A^(k/2)，也就是二分的思想。无限二分，直到 k=1，不能二分为止。
九章算法给的模板非常 nice。。。虽然我也有矩阵快速幂的板子，但是这个太香了，更改一下别扭的地方之后，顺便更新下我的 note。
不得不说，用结构体封装算法的做法我已经见了很多次，但是这里的细节处理，让人感觉非常的舒服~*/

```
class Solution {
public:
    struct Matrix{
        long long mat[30][30];
        int n, mod;
        void init(int _n, int _mod) {
            n = _n; mod = _mod;
        }
        void zero(){
            for(int i = 0; i < n; ++i)
                for(int j = 0; j < n; ++j)
                    mat[i][j] = 0;
        }
        void unit() {
            for(int i = 0; i < n; ++i)
                for(int j = 0; j < n; ++j)
                    mat[i][j] = (i == j);
        }
        Matrix operator *(const Matrix&m)const{
            Matrix tmp;
            tmp.init(n, mod);
            tmp.zero();
            for(int i = 0 ; i < n; ++i)for(int k = 0 ; k < n; ++k){
                if(mat[i][k]==0) continue;
                for(int j = 0 ; j < n; ++j){
                    tmp.mat[i][j] += mat[i][k] * m.mat[k][j];
                    tmp.mat[i][j] %= mod;
                }
```

```
            }
            return tmp;
        }
        Matrix operator +(const Matrix&m)const{
            Matrix tmp;
            tmp.init(n, mod);
            for(int i = 0 ; i < n; ++i)for(int j = 0 ; j < n; ++j){
                tmp.mat[i][j] = (mat[i][j] + m.mat[i][j]) % mod;
            }
            return tmp;
        }
    }mA, mB;
    Matrix power(Matrix mA, int a){
        mB.init(mA.n, mA.mod);
        mB.unit();
        while(a){
            if(a & 1)mB = mB * mA;
            mA = mA * mA;
            a /= 2;
        }
        return mB;
    }
    Matrix solve(Matrix mA, int k){
        if(k == 1)return mA;
        int hf = k / 2;
        Matrix ans = solve(mA, hf), tmp = power(mA, hf);
        ans = ans + ans * tmp;
        if(k & 1)ans = ans + power(mA, k);
        return ans;
    }
    vector<vector<int>> matrixPowerSeries(vector<vector<int>> &A, int k, int m) {
        // write your code here
        int n = A.size();
        mA.init(n, m);
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j)
                mA.mat[i][j] = A[i][j];
        Matrix tmp = solve(mA, k);
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j)
                A[i][j] = tmp.mat[i][j];
        return A;
    }
};
```

LintCode-1072 找出第 k 小的距离对

/*暴力来解就是直接两重循环打个表，然后快排就行了。不试了，绝对超时，因为这个打表要记录很多信息，复杂度超过 O(n2)了。
第一重循环是肯定要的，第二重可以进行优化，也接触了很多算法在这层都可以进行操作，那么应该是找一种 O(nlogn)的算法。
无意间看到了题目下面有人记录了个二分查找（咳咳，绝不是手动搜的），那么二分查找确实也符合要求，毕竟查找一次也就 O(logn)。
现在确定二分的具体实现。首先二分需要一个单调的数组，那么我们直接 sort 即可，这里的 sort 用快排的话，复杂度是加法，影响不大。然后我们将 L=两个点的最短距离（也不用过于严格，毕竟是找最大的），R=两个点的最长距离，这个最长的话可以跑一遍数组，找出最大和最小，直接作差。
二分的实现完成了，关键的地方在于 judge 函数。我们要根据当前的 mid，判断下一次查找的区间，数组单调的性质在这里也有要求。mid 的含义是当前的一段距离，我们在 judge 函数判断一下比当前这段距离小的距离对数有多少就行。

其实这样实现的话复杂度也不低，算不好，估计超过 O(n2)了，所以选用的 judge 函数的写法的效率比较高，不过也有优化不了多少。。。*/

```
class Solution {
public:
    /**
     * @param nums: a list of integers
     * @param k: a integer
     * @return: return a integer
     */
    int judge(vector<int>& nums, int mid) {
        int ans = 0, j = 0;
        for (int i = 1; i < nums.size(); ++i) {
            while (j < i && nums[i] - nums[j] > mid) ++j;
            ans += i - j;
        }
        return ans;
    }
    int smallestDistancePair(vector<int> &nums, int k) {
        sort(nums.begin(), nums.end());

        int l = 0, r = nums[nums.size() - 1] - nums[0];
        while (r > l) {
            int mid = (l + r) >> 1;
            if (judge(nums, mid) < k)
                l = mid + 1;
            else
                r = mid;
```

```
        }
        return mid;
    }

};
```

LintCode-1616 最短子数组 II

/*因为要返回的是最小（短）的连续子数组的长度，那么初步的想法是从小到大枚举所有长度，然后枚举所有的子数组。最坏的情况下，复杂度应该是 O(n2)，当然还要处理一下前缀和，不过不会增加多少复杂度。
之前在这种问题上卡了挺久了，也知道了连续子数组可以用滑动窗口来解，上面的那种暴力就不试了。
对于一个队列 deque 来说，是由首尾两个指针的，分别对应队首和队尾。现在就分为入队列和筛选结果两个部分。
对于筛选，我们可以将入队列时所有可能作为首位的值全部枚举一次，当 sum[i]-sum[q.front()]>=k 时，就可能是答案。
对于入队列，我们要将以当前下标 i 开始，可能成为答案的位置入队列。和之前结果的题类似，我们肯定是先无差别的将当前位置入队列，结合筛选的判断条件，如果当前位置对应的值为负数，那么我们要将前一个队尾给 pop。
虽然这类解法这里都很相似，但是我们还是要分析下原因。假设当前的队尾是 x-1,当前位置是 x，且 sum[x-1]>=sum[x]。在筛选时，当 i=y 时，我们计算 sum[y]-sum[x-1]和 sum[y]-sum[x]。显然，后者的长度更短，而且后者的值更大，因此后者比前者更有可能成为答案，这里的可能是从贪心的角度说的，实际上是一定。
和其他滑动数组解法相似，复杂度并不能说降低多少，但是我们注意到入队列和筛选时可以同步进行的。那么即使在最坏的情况下，复杂度也不可能超过 O(n2)。*/

```
class Solution {
public:
    /**
     * @param nums:
     * @param k:
     * @return: return the length of subarray
     */
    int smallestLengthII(vector<int> &nums, int k) {
        // Write your code here
        int n=nums.size(),ans=1e9;
        vector<int>sum(n+1,0);
        for(int i=0;i<n;++i)
            sum[i+1]=sum[i]+nums[i];

        deque<int> q;

        for(int i=0;i<n+1;++i)
```

```
        {
            while(!q.empty()&&sum[q.back()]>=sum[i])q.pop_back();
            q.push_back(i);
            while(!q.empty()&&sum[i]-sum[q.front()]>=k)
            {
            ans=min(ans,i-q.front());
            q.pop_front();
            }
        }

        return ans<1e9-1?ans:-1;
    }
};
public:
    /**
     * @param nums:
     * @param k:
     * @return: return the length of subarray
     */
    int smallestLengthII(vector<int> &nums, int k) {
        // Write your code here
        int n=nums.size(),ans=1e9;
        vector<int>sum(n+1,0);
        for(int i=0;i<n;++i)
            sum[i+1]=sum[i]+nums[i];

        deque q;

        for(int i=0;i<n;++i)
        {
            while(!q.empty()&&sum[d.back()]>=sum[i])q.pop_back();
            q.push_back(i);
            while(!q.empty()&&sum[i]-sum[q.front()]>=k)
            {
            ans=min(ans,i-q.front();
            q.pop_front();
            }
        }

        return ans<1e9-1?ans:-1;
};
```

2.2 链表

LintCode-36 翻转链表 II

/*题目大意：翻转链表的任意子区间。
其实就是对链表的基础操作，没有任何复杂的思路可言，就是细枝末节的操作。
当然，以上这种暴力的思想是不可取的（直觉）。
我们注意到将中间的一段链表翻转，那么需要考虑首尾的中间两个部分。
对于首尾是比较容易操作的，让相邻的结点的指向改变即可。
对于中间的部分，我们需要将指向反转，也就是说从 1->2->3 翻转为 1<-2<-3，也是比较容易操作的。
还有一个地方需要注意，就是考虑中间结点和首尾是不一样的。我们在 head 前再插入一个结点，去除特殊情况。*/

```
/**
 * Definition of singly-linked-list:
 * class ListNode {
 * public:
 *         int val;
 *         ListNode *next;
 *         ListNode(int val) {
 *             this->val = val;
 *             this->next = NULL;
 *         }
 * }
 */

class Solution {
public:
        ListNode *reverseBetween(ListNode *head, int m, int n) {
            // write your code here
            ListNode *dummy = new ListNode(0);
            dummy->next = head;
            ListNode *pre = dummy;
            ListNode *prepre = NULL;

            for(int i=0; i<m; i++){
                prepre = pre;
                pre = pre->next;
            }

            int times = n-m;
```

```
        ListNode *cur = pre->next;
        while(cur != NULL && times > 0){
            ListNode *next = cur->next;

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

            times--;
        }

        prepre->next->next = cur;
        prepre->next = pre;

        return dummy->next;
}
};
```

LintCode-48 主元素 III

/*给定一个整型数组，找到主元素，它在数组中的出现次数严格大于数组元素个数的 1/k。
要求时间复杂度为 O(n)，空间复杂度为 O(k)。
非常诡异的用链表解数组，让我想起了快慢指针在重复元素的应用。
为了得到某个具体数字的数量，我们要在跑数组的过程中计数，结合空间复杂度为 O(k)，
想要离散化处理，但是明显不符合题目要求，不会做，直接看答案了。
hash 表是我没想到的，几个月没用，拿空间换时间已经是我对它最后的印象了。
不过这里我就用 map 解了，毕竟这是书上的例题了。*/

```
class Solution {
public:
    /**
     * @param nums: A list of integers
     * @param k: An integer
     * @return: The majority number
     */
    int majorityNumber(vector<int> &nums, int k) {
        // write your code here
        map<int,int>m;
        for(int i=0;i<nums.size();++i){
            ++m[nums[i]];
            if(m[nums[i]]>nums.size()/k)return nums[i];
        }
    }
};
```

LintCode-96 链表划分

/*链表其实我不是很熟，比如熟悉的线性表只有数组和队列，栈的话只在单调栈用过，hash就不说了，取留余数法和线性探测散列万岁！
那从简单的难度做起，简单而且通过率<40%的感觉应该比较有意思。
给定一个单链表和数值 x，划分链表使得所有小于 x 的节点排在大于等于 x 的节点之前。
初步的想法是跑一遍链表，然后把小于 x 的节点全部排好。这其实不难，当遇到一个小于 x 的节点时，使这个新生成的链表最后一个尾插，遇到大于就跳过。当然还要一个大于等于 x 节点的新链表，最后组合一下即可。
细节还是有一些的，不过不用动态管理内存的话也不算复杂，想想自己在 C++课本上第一次看到这种数据结构时的不理解到现在的稍微有点点思路，还是有些感慨。*/

```
/**
 * Definition of singly-linked-list:
 * class ListNode {
 * public:
 *         int val;
 *         ListNode *next;
 *         ListNode(int val) {
 *             this->val = val;
 *             this->next = NULL;
 *         }
 * }
 */

class Solution {
public:
    /**
     * @param head: The first node of linked list
     * @param x: An integer
     * @return: A ListNode
     */
    ListNode * partition(ListNode * head, int x) {
        // write your code here
        ListNode *l = new ListNode(0);
        ListNode *r = new ListNode(0);
        ListNode *leftTail = l;
        ListNode *rightTail = r;

        while (head != nullptr) {
            if (head->val < x) {
```

```
                leftTail->next = head;
                leftTail = head;
            } else {
                rightTail->next = head;
                rightTail = head;
            }
            head = head->next;
        }

        leftTail->next = r->next;
        rightTail->next = nullptr;

        return l->next;
    }
};
```

LintCode-98 链表排序

/*快速排序是找个基准点（最左边的点），将这个点成为整个区间的中值（点），然后对该中点分成的左右区间递归确定新的中点，从而实现排序。
归并排序是先将序列拆开，但是相对位置关系保留。然后两两子区间合并，合并时实现升序。
实现的方式也是递归，不断拆分成两个子区间，当全部拆开后，再找个合并。
这里采用了归并排序，归并排序有一个要求，就是找到中间的点。这对于可以随机访问的数据结构很简单，但是对于链表来说，想想就麻烦。
之前写过一道快慢指针的题目，这里就用这种方式来确定中点，具体的做法在这题就不说了。
再提一点，归并排序其实是控制子区间的起始和终止，也是就是 merge_sort(l,r)，l,r 才是拆分的核心。对于链表来说，也是从 l->r 的合并/拆分。*/

```
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: You should return the head of the sorted linked list,
                    using constant space complexity.
     */
    ListNode *sortList(ListNode *head) {
        // write your code here
        if(head == NULL || head->next == NULL) {
            return head;
        }

        ListNode *fast = head, *slow = head, *temp = head;
        while(fast != NULL && fast->next != NULL) {
            temp = slow;
```

```
            slow = slow->next;
            fast = fast->next->next;
        }
        temp->next = NULL;
        ListNode *l=sortList(head),*r=sortList(slow);
        return mergeList(l, r);
    }

    ListNode *mergeList(ListNode *head1, ListNode *head2) {
        if(head1 == NULL) {
            return head2;
        }
        if(head2 == NULL) {
            return head1;
        }

        ListNode newHead(0);
        ListNode *temp = *newHead;
        while(head1 != NULL && head2 != NULL) {
            if(head1->val < head2->val) {
                temp->next = head1;
                head1 = head1->next;
            }
            else {
                temp->next = head2;
                head2 = head2->next;
            }
            temp = temp->next;
        }
        if(head1 != NULL) {
            temp->next = head1;
        }
        else if(head2 != NULL) {
            temp->next = head2;
        }

        return newHead.next;
    }
};
```

LintCode-99 重排链表

/*重排列表是需要一定的规律的，就像翻转一样，所以肯定是从重排后的序列特点入手。
很容易就能看出，重排后的序列是新建一个 list，然后从原 list 的首、尾按顺序插入到新的

list 中。但是这种做法不太符合题目“原地操作”的要求。
因为单链表的结构限制，要找到当前结点的前一节点是不容易的。换句话说，要实现列表的翻转，那么需要 pre、cur 和 next 三个指针来实现 pre 和 cur 的指向翻转并向下推进。
如果我们按照目标序列进行设计，那么需要从当前列表的尾部向前推进，这是一件相当麻烦的事。结合翻转，我们可以将原 list 的后半段进行 reverse，然后设置两个指针，进行交错的重构 list。*/

```
/**
 * Definition of singly-linked-list:
 * class ListNode {
 * public:
 *         int val;
 *         ListNode *next;
 *         ListNode(int val) {
 *             this->val = val;
 *             this->next = NULL;
 *         }
 * }
 */

class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: void
     */
    void reorderList(ListNode *head) {
        // write your code here
        if(head == NULL || head->next == NULL) return;

            ListNode *fast = head, *slow = head, *temp = head;
            while(fast != NULL && fast->next != NULL) {
                temp = slow;
                slow = slow->next;
                fast = fast->next->next;
            }
            temp->next = NULL;

            slow = reverse(slow);

            ListNode *newHead = head;

            while(newHead != NULL) {
                if(newHead->next == NULL) {
```

```
                    newHead->next = slow;
                    break;
                }
                else{
                    temp = newHead->next;
                    newHead->next = slow;
                    slow = slow->next;
                    newHead->next->next = temp;
                    newHead = temp;
                }
            }

        }

        ListNode *reverse(ListNode *head) {
            ListNode *pre=NULL,*cur=NULL,*next=NULL;

            pre = head;
            if(pre == NULL || pre->next == NULL) {
                return pre;
            }
            cur = pre->next;
            if(cur->next == NULL) {
                cur->next = pre;
                pre->next = NULL;
                return cur;
            }
            next = cur->next;
            if(cur->next != NULL) {
                while(next!= NULL) {
                    cur->next = pre;
                    if(pre == head) {
                        pre->next = NULL;
                    }
                    pre = cur;
                    cur = next;
                    next = next->next;
                }
                cur->next = pre;
                return cur;
            }
        }
};
```

LintCode-103 带环链表 II

/*我所知道的快慢指针的应用大概有求链表中点、判断是否存在环、返回环的长度、确定环的入口点以及求倒数第 k 个链表元素。
对于求倒数第 k 个链表元素大概还有个印象，其他的几个都非常简单。本题正好是确定环的入口点，是在碰撞之后，令 slow=head，然后 slow 和 fast 再次相遇的点。这个是唯一证明有点难度的，那么这里正好来证明一下。
慢指针一次走 1 步，快指针一次走 2 步。假设慢指针到入口点，快指针此时距离它 m 步。
我们可以看做快指针以一次走 1 步的速度追慢指针，因此在慢指针走了 m 步，快指针走了 2m 步后，到达碰撞点。
假设 head 到入口点的距离与 a,环的长度为 n，记从头开始，慢指针一共走了 s 步，快指针走了 2s 步。
从链表首到碰撞点：s=a+m,2s=a+m+x*n。(这里的 2s 是从结果来说的，slow 是直接走到碰撞点，fast 可以看做走到碰撞点后，又转了 x 圈)
由此，a=n*x-m。同时我们知道，n-m 时 slow 从碰撞点正好回到环的入口,n*x-m 就是 slow 回到环的入口。当然可以采取点优化，令 slow=head,fast=fast->next，节省点空间。
那么，在头结点和碰撞节点分别设一个指针，同步前进，最后会在入口节点相遇。
本题比较简单，就是快慢指针的直接应用，就不详细说了。*/

```
/**
* Definition of singly-linked-list:
* class ListNode {
* public:
*         int val;
*         ListNode *next;
*         ListNode(int val) {
*             this->val = val;
*             this->next = NULL;
*         }
* }
*/

class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: The node where the cycle begins. if there is no cycle, return null
     */
    ListNode * detectCycle(ListNode * head) {
        // write your code here
        ListNode*slow=head,*fast=head;
        while(fast&&fast->next){
            slow=slow->next;
            fast=fast->next->next;
```

```
            if(slow==fast)break;
        }
        if(fast==NULL||fast->next==NULL)return NULL;
        slow=head;
        while(slow!=fast)slow=slow->next,fast=fast->next;
        return slow;

    }
};
```

LintCode-104 合并 k 个排序链表

/*合并链表是个什么操作，这题通过率都能<40%。根据样例来看，应该是要求升序排序吧。
题目虽然没有交代，但是给我们的每个链表应该都是已经排序好的，那么最容易想到的办法就是 merge_sort 的方法，那么复杂度应该是 O(nlogn)。
为什么是归并不是快排，因为我在前面一题用的归并，可以 paste 了。
这题就水了，充个数。*/

```
class Solution {
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
public:
    ListNode *mergeKLists(vector<ListNode*> &lists) {
        if (lists.size() == 0) {
            return nullptr;
        }
        return mergeHelper(lists, 0, lists.size() - 1);
    }

     ListNode *mergeHelper(vector<ListNode*> &lists, int start, int end) {
        if (start == end) {
            return lists[start];
        }

        int mid = (start + end) / 2;
        ListNode *left = mergeHelper(lists, start, mid);
        ListNode *right = mergeHelper(lists, mid + 1, end);
        return mergeTwoLists(left, right);
    }

    ListNode *mergeTwoLists(ListNode *list1, ListNode *list2) {
        ListNode *dummy = new ListNode(0);
```

```
        ListNode *tail = dummy;
        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val < list2->val) {
                tail->next = list1;
                tail = list1;
                list1 = list1->next;
            }
            else {
                tail->next = list2;
                tail = list2;
                list2 = list2->next;
            }
        }
        if (list1 != nullptr) {
            tail->next = list1;
        }
        else {
            tail->next = list2;
        }

        return dummy->next;
    }
};
```

LintCode-105 复制带随机指针的链表

/*给出一个链表，每个节点包含一个额外增加的随机指针可以指向链表中的任何节点或空的节点。返回一个深拷贝的链表。
依稀记得自己听过深拷贝这个词，所以去查了下，原来就是不会和拷贝对象绑定。看来自己记忆力下滑严重，什么都不会了。
初步的想法是，用个数组把所有信息都储存，然后添加到当前对象之类的。如果对应到链表来说，实现起来并没有那么容易。
在我尝试了一会后，我才想起来深拷贝的含义，也就是我不能用原来的结点。
想要完整拷贝一个链表，空间复杂度 O(1)，没啥思路，去看了下别人的笔记。
将旧连表中各个结点逐个拷贝后，将每个结点安插在旧结点的 next 结点。旧连表指针只遍历旧结点，新链表指针只遍历新结点，然后将两链表拆分，返回新链表即可。
原来这就是 O(1)的复杂度……我果然不会算复杂度。
逐个拷贝的思路很简单，设置两个指针，第二个指针拷贝第一个指针，然后修改下 next。
然后 WA 了好多次。大概是两个原因，1 是因为我只记得去改变指向了，忽略了 random 需要重新修改；2 是我是通过 cur->next=cur->next->next 和 cur=cur->next 来实现遍历和分离的，忽略了一个问题，那就是 cur 为边界时，cur->next->next 不存在。*/

```
/**
 * Definition for singly-linked list with a random pointer.
 * struct RandomListNode {
 *     int label;
 *     RandomListNode *next, *random;
 *     RandomListNode(int x) : label(x), next(NULL), random(NULL) {}
 * };
 */
class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head) {
        if (!head) return NULL;
        RandomListNode* cur = head;
        while (cur) {
            RandomListNode* node = new RandomListNode(cur->label);
            node->next = cur->next;
            cur->next = node;
            cur = node->next;
        }
        cur = head;
        while (cur) {
            if (cur->random) {
                cur->next->random = cur->random->next;
            }
            cur = cur->next->next;
        }
        cur = head;

        RandomListNode* first = cur->next;
        while (cur) {
            RandomListNode* tmp = cur->next;
            cur->next = tmp->next;
            if (tmp->next) {
                tmp->next = tmp->next->next;
            }
            cur = cur->next;
        }

        return first;
    }
};
```

LintCode-106 有序链表转换成二叉搜索树

/*虽然不懂高度平衡的含义，但是应该是要从中点开始，然后构建一个二叉树。
很明显使用递归的方法做的，每次取中点，然后中点作为根结点，分别指向左右子区间的中点。
有一个值得思考的地方，就是分割实际上是要以中间点为界，分割成包含中点在内的三个部分。那么我们用快慢指针直接定位在中点，对于左半部分的分割就存在一点问题——无法与中点分离开。
为了解决这个问题，对于快慢指针进行一些调整，fast 的初始值定为 head->next 就行。这样的话，slow->next 就是中点。*/

```
/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *         int val;
 *         ListNode *next;
 *         ListNode(int val) {
 *                 this->val = val;
 *                 this->next = NULL;
 *         }
 * }
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *         int val;
 *         TreeNode *left, *right;
 *         TreeNode(int val) {
 *                 this->val = val;
 *                 this->left = this->right = NULL;
 *         }
 * }
 */


class Solution {
public:
    /*
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    TreeNode * sortedListToBST(ListNode * head) {
        // write your code here
        if(head==nullptr)return nullptr;
```

```
        if(head->next==nullptr)return new TreeNode(head->val);

        ListNode *slow=head,*fast=head->next;
        while(fast->next&&fast->next->next){
            slow=slow->next;
            fast=fast->next->next;
        }
        ListNode *mid=slow->next;
        TreeNode *tree=new TreeNode(mid->val);
        slow->next=nullptr;
        tree->left=sortedListToBST(head);
        tree->right=sortedListToBST(mid->next);
        return tree;
    }
};
```

LintCode-113 删除排序链表中的重复数字

/*这个还是比较有意思，链表的去重操作。
一般去重比较习惯的就是借助 map(好像是红黑二叉树吧)，那这里似乎也可以这样。题目所给的 list->map->去重后的 list。这种想法我也就想想，应该不行。
如果可以考虑 O(n2)复杂度的话，那么对于每个节点，从当前位置往后搜索，去除重复的，似乎也是可行的。
题目还有个条件，就是该链表是升序排列的，那么我们可以设置两个指针 l,r。l、r 开始在同一位置,r 向右移动，知道 r->next!=l，删除 l~r 的内容即可。
当然这样的话 l 和 r 也是要删除的，因此设计为删除的区间应为 l->next~r->next。同时更特殊的情况是 head 需要被删除。因此还是先在首部添加个空指针简化处理。*/

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        // write your code here
        ListNode *dummy = new ListNode(0);
        dummy->next = head;

        ListNode *l = dummy;
        ListNode *r = head;

        while(r){
            r = r->next;

            while(r && l->next->val == r->val)
                r = r->next;
```

```
            if(l->next->next == r)
                l = l->next;
            else   //如果两指针间存在重复元素，则将 left 与 right 连接，删除重复元素
                l->next = r;
        }

        return dummy->next;
    }
};
```

LintCode-134 LRU 缓存策略

/*LRU 即最近最少使用，学习一下新东西(其实计算机组成原理访问主存就提过这种策略)。
我记得这个的原理是设置一个未访问次数计数器，当需要替换时，把计数值最大的换出。当命中时，清零。
其实这个数据结构的设计思路非常的多，可以用 map，但是 map 其实没有必要，因为如果把命中的都放在数据结构的最后，那么每次删除最前面的元素就行。可以用 vector 或者 deque，set 和 get 都视为命中，set 就尾插，get 就先 pop 再尾插，每次 set 时把首位 pop。
但是这里既然是链表的题目，那么用链表吧。链表可以灵活的插入、删除，因此效率应该很高。
要实现数据结构的话还是比较复杂的，但是操作都是一些基本的操作，并不难。*/

```
class LRUCache{
public:
    struct ListNode {
        ListNode *next;
        int key, value;

        ListNode(int k, int v) {
            key = k;
            value = v;
            next = NULL;
        }
    };

    int count, cap;
    ListNode *head, *tail;
    LRUCache(int capacity) {
        count = 0;
        cap = capacity;
        head = tail = NULL;
    }
```

```
int get(int key) {
        ListNode *cur = head, *pre = NULL;
        int res = -1;
        while (cur) {
                if (cur->key == key) {
                        res = cur->value;
                        break;
                }
                pre = cur;
                cur = cur->next;
        }
        if (res != -1 && cur != tail) {
                if (pre) pre->next = cur->next;
                else head = cur->next;

                cur->next = NULL;
                tail->next = cur;
                tail = cur;
        }
        return res;
}


void set(int key, int value) {
        ListNode *cur = head, *pre = NULL;
        bool found = false;
        while (cur) {
                if (cur->key == key) {
                        found = true;
                        cur->value = value;
                        break;
                }
                pre = cur;
                cur = cur->next;
        }
        if (!found) {
                count++;
                ListNode *q = new ListNode(key, value);
                if (!tail) head = tail = q;
                else {
                        tail->next = q;
                        tail = q;
                }
```

```
                if (count > cap) {
                    count--;
                    head=head->next;
                }
            }
            else {
                if (cur != tail) {
                    if (pre) pre->next = cur->next;
                      else head = cur->next;

                    cur->next = NULL;
                    tail->next = cur;
                    tail = cur;
                }
            }
        }
};
```

LintCode-167 链表求和

/*模拟大数运算，只不过这里变成了链表。
个人感觉数组还是最麻烦的，string 和 list 之类的比数组写起来舒服一点。
有一个小小的要注意的点。这里的链表是代表这个数的反序，那么我们就在模拟计算的过程中不用翻转这个数了。计算结果也是返回反序，那么也不用翻转*/

```
class Solution {
public:
    /**
     * @param l1: the first list
     * @param l2: the second list
     * @return: the sum list of l1 and l2
     */
    ListNode *addLists(ListNode *l1, ListNode *l2) {
        // write your code here
        ListNode *head = new ListNode(0);
        ListNode *temp = head;
        int in = 0;
        while (l1!= NULL && l2 != NULL) {
            int sum = l1->val + l2->val + in;
            ListNode *node = new ListNode(sum % 10);
            in = (sum >= 10);
            temp->next = node;
            l1 = l1->next;
```

```
            l2 = l2->next;
            temp = temp->next;
        }
        while (l1 != NULL) {
            int sum = l1->val + in;
            ListNode *node = new ListNode(sum % 10);
            in = (sum >= 10);
            temp->next = node;
            l1 = l1->next;
            temp = temp->next;
        }
        while (l2 != NULL) {
            int sum = l2->val + in;
            ListNode *node = new ListNode(sum % 10);
            in = (sum >= 10);
            temp->next = node;
            l2 = l2->next;
            temp = temp->next;
        }
        if (l1 == NULL && l2 == NULL && in == 1) {
            ListNode *node = new ListNode(in);
            temp->next = node;
        }
        return head->next;
    }
};
```

LintCode-170 旋转链表

/*旋转链表，也就是整体向右移 k 位的操作。
对于字符串或者数组之类的循环移位操作，已经比较熟悉了，感觉链表会更加简单一些。
一般而言，可以将链表复制一遍，插在尾部，再截图第 k 位之后与原链表长度相等的部分即可。当然，实际上是指针移位，这与链表切合的也很好。
我们可以先找到第 k 位，把前 k 位用一个新建链表存储。再新建一个链表，存储原链表 k 位之后的内容。最后将两个链表合并即可。
不过题目没有说 k 的取值，那 k 可能会很长，因此先跑一边链表记录下长度，再给 k 取模，这样比较保险。
随手写的，边界条件开始有几个没考虑到，但是几分钟就写出来了，开心。*/

```
/**
 * Definition of singly-linked-list:
 * class ListNode {
 * public:
```

```
 *          int val;
 *          ListNode *next;
 *          ListNode(int val) {
 *              this->val = val;
 *              this->next = NULL;
 *          }
 * }
 */

class Solution {
public:
    /**
     * @param head: the List
     * @param k: rotate to the right k places
     * @return: the list after rotation
     */
    ListNode * rotateRight(ListNode * head, int k) {
        // write your code here
        if(head==nullptr||head->next==nullptr)return head;
        ListNode *nothing=head;
        int mod=1;
        while(nothing->next!=nullptr)mod++,nothing=nothing->next;
        k%=mod;
        if(k==0)return head;
        ListNode *list1=head,*list2=head;

        for(int i=1;i<=k;++i)list1=list1->next;
        for(int i=1;i<mod-k;++i)list2=list2->next;
        ListNode *list3=list2->next;
        nothing->next=head;
        list2->next=nullptr;
        return list3;

    }
};
```

LintCode-223 回文链表

/*这个题目有意思，设计一种方案，检测该链表是否为回文链表。
比较暴力一点的想法就是找中点，截断，翻转，判断。
不那么暴力一点的话，我们在刚才的查询中点的过程中，把前半链表扫描了一遍，这部分的结点信息都可以获取，接着要扫描后半部分，可以获取后半部分的信息。
如果回文成立，那么应该是前半部分链表存储的结点信息后入先出，与后半部分结点信息顺序比较，也就是用个栈。(也可以用个向量存，反向扫描)

O(n)的时间和 O(1)的额外空间，符合要求。
这次我比较小心的考虑了各种情况，还顺手保留了截取的两个部分(结果没用到)，提交超过了 100%的提交，震惊。*/

```
/**
 * Definition of singly-linked-list:
 * class ListNode {
 * public:
 *         int val;
 *         ListNode *next;
 *         ListNode(int val) {
 *             this->val = val;
 *             this->next = NULL;
 *         }
 * }
 */

class Solution {
public:
    /**
     * @param head: A ListNode.
     * @return: A boolean.
     */
    bool isPalindrome(ListNode * head) {
        // write your code here
        if(head==nullptr||head->next==nullptr)return true;
        if(head->next->next==nullptr)
        {
            if(head->val==head->next->val)return true;
            else return false;
        }
        stack<int>p;
        ListNode *slow=head,*fast=head,*tmp=slow;
        while(fast->next&&fast->next->next)
        {
            p.push(slow->val);
            tmp=slow;
             slow=slow->next;
            fast=fast->next->next;
        }
        if(fast->next)p.push(slow->val),tmp=slow,slow=slow->next,tmp->next=nullptr;
        else tmp->next=nullptr,slow=slow->next;
        bool flag=true;
        while(slow)
```

```
        {
            if(slow->val==p.top())p.pop(),slow=slow->next;
                else{
                    flag=false;
                    break;
                }
        }
        return flag;
    }
};
```

LintCode-511 交换链表当中两个节点

/*选了道链表通过率最低的题目来做做，让我看看到底是什么题目这么优秀。

给你一个链表以及两个权值 v1 和 v2，交换链表中权值为 v1 和 v2 的这两个节点。保证链表中节点权值各不相同，如果没有找到对应节点，那么什么也不用做。

先不考虑特殊情况，那么可以跑一遍链表，记录下两个目标节点的 pre，然后就很简单的可以操作了。

要考虑的情况太多，这题也太……对得起通过率了。*/

```
class Solution {
public:
    /*
     * @param head: a ListNode
     * @param v1: An integer
     * @param v2: An integer
     * @return: a new head of singly-linked list
     */
    ListNode * swapNodes(ListNode * head, int v1, int v2) {
        // write your code here
        if (head == NULL || head->next == NULL) {
            return head;
```

```
}
if (v1 == v2) {
        return head;
}
ListNode * newHead = new ListNode(-1);
newHead->next = head;
ListNode *node1Prec = NULL, *node1 = NULL, *node1Next = NULL;
ListNode *node2Prec = NULL, *node2 = NULL, *node2Next = NULL;
ListNode *tempPrec = newHead, *temp = head, *tempNext = head->next;



while (temp != NULL) {
        if (temp->val == v1) {
                node1Prec = tempPrec;
                node1 = temp;
                node1Next = tempNext;
        }
        else if (temp->val == v2) {
                node2Prec = tempPrec;
                node2 = temp;
                node2Next = tempNext;
        }

        if (node1 != NULL && node2 != NULL) {
                break;
        }

        tempPrec = tempPrec->next;
        temp = temp->next;
        if (tempNext != NULL) {
```

```
                tempNext = tempNext->next;
            }
        }

        if (node1 != NULL && node2 != NULL) {
            if (node1->next == node2) {
                node1->next = node2Next;
                node2->next = node1;
                node1Prec->next = node2;
            }
            else if (node2->next == node1) {
                node2->next = node1Next;
                node1->next = node2;
                node2Prec->next = node1;
            }
            else {
                node1Prec->next = node2;
                node1->next = node2Next;

                node2Prec->next = node1;
                node2->next = node1Next;
            }
        }

        return newHead->next;
    }
};
```

2.3 栈与队列

LintCode-71 二叉树的锯齿形层次遍历

/*本质是二叉树的层次遍历。不过在偶数层要翻转。
层次遍历还是比较简单的，当队列不为空时，根据当前队列的长度依次访问每个节点，奇数层则依次记录每个节点的内容，偶数层需要翻转一次，然后再将每个节点的两个子节点入队列并记录该层的答案。*/

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param root: A Tree
     * @return: A list of lists of integer include the zigzag level order traversal of its nodes' values.
     */
    vector<vector<int>> zigzagLevelOrder(TreeNode * root) {
        // write your code here
        vector<vector<int>> result ;
        if(root == nullptr)
            return result;
        queue<TreeNode*> q ;
        //正反向标志
        bool isForward = true;
        q.push(root);
        while(!q.empty()) {
            //由于下面还要向 queue 里面添加节点，这个 size 就是上一层的节点总数
            int size = q.size();
            vector<int> subList ;
            for(int i = 0 ; i < size ; i++) {
```

```
                TreeNode *current = q.front();
                if(isForward)
                    subList.push_back(current->val);
                else
                    subList.insert(subList.begin(),current->val);
                if(current->left != nullptr)
                    q.push(current->left);
                if(current->right != nullptr)
                    q.push(current->right);
                  q.pop();
            }
            result.push_back(subList);
            //方向反转
            isForward = !isForward;
        }
        return result;
    }
};
```

LintCode-193 最长有效括号

/*做个通过率 39%的题目压压惊。
给出一个只包含'(' 和')'的字符串，找出其中最长的左右括号正确匹配的合法子串。
遇到'('就入栈，遇到')'且栈不为空就出栈，同时记录第一个'('的位置，计算最大值。当栈为空且遇到')'时，更新第一个'('的位置，计算最大值。同时考虑结束时栈中还剩有'('的情况，及时与栈顶记录的位置计算最大值即可。*/

```
class Solution {
public:
    /**
     * @param s: a string
     * @return: return a integer
     */
    int longestValidParentheses(string &s) {
        int n = s.size();
        stack<int> st;
        int startPos = 0;
        int maxCount = 0;
        for (int i = 0; i < n; ++i) {
            if (s[i] == '(') st.push(i);
            else    {
                if (!st.empty()) {
                    st.pop();
                    if (st.empty()) {
```

```
                    maxCount = max(maxCount, i - startPos + 1);
                }
                else {
                    maxCount = max(maxCount, i - st.top());
                }
            }
            else {
                if (i < n - 1) startPos = i + 1;
            }
        }
    }

    return maxCount;
  }
};
```

LintCode-368 表达式求值

/*我们是学过用栈来模拟计算器的，因此这个表达式求值那肯定不在话下(纯属吹牛，我现在看书都可能读不懂那个计算器了)。
我又拿起了课本，尝试去抄了这题。
从左到右扫描表达式，遇到操作数就压入操作数栈，遇到运算符就和当前栈顶的运算符比较优先级，若优先级高或者栈顶为空，那么压入栈中，当然，如果优先级比栈顶低的话，就要弹出栈顶，然后根据该栈顶运算符弹出对应的操作数，进行运算后，把运算结果压入操作数栈，重复，知道符合之前的条件再压入当前运算符。
这个在当前的我看来还是有点晕的，不过现在已经没什么问题了。
表达式结束后，逐个弹出运算符栈顶，根据弹出的运算符，取出操作数，进行运算后，将结果压回操作数栈。
想要实现还是不容易的，我认为比较麻烦的地点在于根据运算符弹出操作数，当然，有课本的我选择直接抄。
传统的做法是直接求解中缀表达式，但是我们已经非常熟悉逆波兰表达式的解法了，那么可以先将中缀表达式转化成后缀表达式，再求解。
关于逆波兰表达式的求解，我在另一题中已经讲过了，那么现在问题就是将中缀表达式转化成后缀表达式这是一个麻烦的地方，下面讲讲具体的实现。
我们需要分别有无括号两种情况。
1、没有括号时。我们规定，数字的优先级>乘除>加减，这是由后缀表达式的定义决定的，换句话说，我们要实现数字在左，符号在右，这样的优先级序列有利于我们实现这个目标。
从左到右扫描，我们首先判断一下该字符的类型。如果是数字，那么优先级最高，直接压入栈中。如果是运算符，若栈不为空且栈顶的优先级大，那么我们取出栈顶，尾插到目标后缀表达式中，因为这符合我们对优先级大的在左边的要求。
当然，这个操作需要反复执行，以确保当前字符是栈中优先级最大的，最后压入栈中。举个例子，3+4*5，按照顺序，依次执行：3 入栈，3 出栈(尾插)，+入栈，4 入栈，4 出栈(尾插)，*入栈，5 入栈,5 出栈(尾插)，*出栈(尾插),+出栈(尾插)。结果就是 345*+。
2、有括号时。括号的内部是进行基本的操作，相对于括号内部来说，括号的优先级是最低

的。那么我们在读到"("时正常入栈，因为括号的优先级最低，在读取字符期间不会出栈，因此在读到")"之前和没有括号的操作一样。当读到")"时，我们把栈中，"("之后的元素全部出栈，这样来说比较抽象，举个例子。4*(5+6)，依次执行：4 入栈，4 出栈，*入栈，(入栈，5 入栈，5 出栈，+入栈，6 入栈，6 出栈，+出栈，(出栈，*出栈。结果就是 456+*。
其实括号就是改变了优先级，本来栈是以优先级为序单调的，但是以"("为界，这样就可以在读到")"时， 优先进行"("右边的操作了。 */

```
class Solution {
public:

    /**
     * @param expression: a list of strings
     * @return: an integer
     */
    int evaluateExpression(vector<string> &expression) {
        // write your code here
        vector<string>tokens=convertToRPN(expression);
        int n = tokens.size();
        if (tokens.empty()) return 0;
        stack<int> stack;
        for (int i = 0; i < n; i++) {

            if (tokens[i] != "+" && tokens[i] != "-" && tokens[i] != "*" && tokens[i] != "/")
                stack.push(atoi(tokens[i].c_str()));

            else {
                int num1 = stack.top();
                stack.pop();
                int num2 = stack.top();
                stack.pop();
                int num3 = 0;
                if (tokens[i] == "+") {
                    num3 = num2 + num1;
                }
                else if (tokens[i] == "-") {
                    num3 = num2 - num1;
                }
                else if (tokens[i] == "*") {
                    num3 = num2 * num1;
                }
                else if (tokens[i] == "/") {
                    num3 = num2 / num1;
                }
                stack.push(num3);
```

```
            }
        }
        return stack.top();
}

int getLevel(string opt) {
        if (opt == "(")
                return 0;
        if (opt == "+" || opt == "-")
                return 1;
        if (opt == "*" || opt == "/")
                return 2;

        return 3;
}

bool isOperator(string c) {
        return (c == "+" || c == "-" || c == "*" || c == "/");
}
//create reverse polish notation string
vector<string> convertToRPN(vector<string> &expression) {
        stack<string> st;
        vector<string> RPN;
        int len = expression.size();
        for (int i = 0; i < len; ++i) {
                string c = expression[i];
                if (c == "(")
                        st.push(c);
                else if (c == ")") {
                        while (st.top() != "(") {
                                RPN.push_back(st.top());
                                st.pop();
                        }
                        st.pop();
                }
                else {
                        if (!isOperator(c))
                                st.push(c);
                        else {
                                while (!st.empty() && getLevel(st.top()) >= getLevel(c)) {
                                        RPN.push_back(st.top());
                                        st.pop();
                                }
                                st.push(c);
```

```
                }
            }
        }

        while (! st.empty()) {
            RPN.push_back(st.top());
            st.pop();
        }

        return RPN;
    }
};
```

LintCode-370 将表达式转换为逆波兰表达式

/*之前在某题，我写了逆波兰表达式的求解函数。又在某题，我写了将中缀表达式转换成后缀表达式，进而代入我之前写的那个求解函数。现在这题，让我去写表达式->逆波兰表达式的转换……
让我光明正大的水一题吧。*/

```
class Solution {
public:
    /**
     * @param expression: A string array
     * @return: The Reverse Polish notation of this expression
     */
    int getLevel(string opt) {
        if (opt == "(")
            return 0;
        if (opt == "+" || opt == "-")
            return 1;
        if (opt == "*" || opt == "/")
            return 2;

        return 3;
    }

    bool isOperator(string c) {
        return (c == "+" || c == "-" || c == "*" || c == "/");
    }
```

```
    vector<string> convertToRPN(vector<string> &expression) {
        stack<string> st;
        vector<string> RPN;
        int len = expression.size();
        for (int i = 0; i < len; ++i) {
            string c = expression[i];
            if (c == "("
                st.push(c);
            else if (c == ")") {
                while (st.top() != "(") {
                    RPN.push_back(st.top());
                    st.pop();
                }
                st.pop();
            }
            else {
                if (!isOperator(c))
                    st.push(c);
                else {
                    while (!st.empty() && getLevel(st.top()) >= getLevel(c)) {
                            RPN.push_back(st.top());
                            st.pop();
                    }
                    st.push(c);
                }
            }
        }

        while (! st.empty()) {
            RPN.push_back(st.top());
            st.pop();
        }

        return RPN;
    }
};
```

LintCode-421 简化路径

/*这个题目看着挺有意思的，虽然通过率只有 27%。
给定一个文件的绝对路径(Unix-style)，请进行路径简化。
Unix 中, . 表示当前目录, .. 表示父目录。
结果必须以 / 开头，并且两个目录名之间有且只有一个 /。最后一个目录名(如果存在)后不能出现 / 。你需要保证结果是正确表示路径的最短的字符串。

说实话，我没太看懂题目，稍微去查了下相关信息，以下是我的理解。
核心的问题在于父目录，也就是/../。当遇到这个时，应该返回上一级的文件。由于上一级文件和当前文件是相邻的，我们只要一个个压栈，读到/../就出栈，也就是返回上一个被压栈的元素即可。
有很多没用的信息需要过滤，因此我们以/为界，将整个目录分割成很多小段进行处理即可。
理论已经有了，但是要实现起来的话，是个比较复杂的过程，可以分成分割路径、简化路径和重写路径(来源：网上大佬的做法)。
分割路径：实现方法多种多样，这里的想法是，因为虽然路径都是以"/"开头，但是我们最好以"/"为结尾考虑。大致就是，创建一个缓存的区域，在读取到"/"之前，将字符串的部分内容加入缓存区中，当读到"/"时，将缓存区的内容记录，清空，重复进行操作。
简化路径：简化路径实际上就是把/./和/../给简化了，从左到右读取字符串，读取刚才在分割路径处理过的，一段段不含"/"的内容。当读到一半内容，压入栈，当读到"."，过滤，当读到".."，栈顶出栈。
重写路径：栈中已经存储了每一段的内容，现在只要将占中的内容取出即可。由于栈只能后入先出，因此可以将栈顶依次插入到答案的首部，当然，插入的同时要补上之前滤去的"/"。
*/

```
class Solution {
public:
    /*
     * @param path: the original path
     * @return: the simplified path
     */
    string simplifyPath(string &path) {
        // write your code here
        int n = path.size();
        if (!n) return string("");

        //  分割路径
        path += '/';
        vector<string> pathVector;
        string p;
        for (int i = 0; i <= n; i++) {
            if (path[i] == '/') {
                if (!p.empty()) {
                    pathVector.push_back(p);
                    p.clear();
                }
            }
            else {
                p += path[i];
            }
```

```
        }
        // 简化路径
        stack<string> pathStack;
        for (int i = 0; i < pathVector.size(); i++) {
            if (pathVector[i] != "." && pathVector[i] != "..") {
                pathStack.push(pathVector[i]);
            }
            else if (pathVector[i] == ".." && !pathStack.empty()) {
                pathStack.pop();
            }
        }
        // 重写路径
        string result;
        if (pathStack.empty()) {
            return "/";
        }
        while (!pathStack.empty()) {
            result.insert(0, "/" + pathStack.top());
            pathStack.pop();
        }
        return result;
    }
};
```

LintCode-423 有效的括号序列

/*又是一道括号的问题，这题看起来就非常简单。
栈的基本操作，难度也是简单题，33%的通过率纯属唬人，不解释了。*/

```
class Solution {
public:
    /**
     * @param s: A string
     * @return: whether the string is a valid parentheses
     */
    bool isValidParentheses(string &s) {
        // write your code here
        if(s.size()&1||s.size()==0)return false;
        stack<char>sk;
        for(int i=0;i<s.size();++i){
            if(sk.empty())sk.push(s[i]);
```

```
                else {
                    if(sk.top()=='('&&s[i]==')')sk.pop();
                    else if(sk.top()=='['&&s[i]==']')sk.pop();
                    else if(sk.top()=='{'&&s[i]=='}')sk.pop();
                    else sk.push(s[i]);
                }
            }
            return sk.empty();
        }
};
```

LintCode-424 逆波兰表达式求值

```
/*对于中缀表达式来说，我们需要运算符和操作数两个栈。对于后缀表达式来说，我们只需要一个操作数栈，因为我们只需要按照顺序处理运算符就行了。
曾经写过递归的一道题目，就是处理逆波兰表达式，这种递归定义的方式还是给我留下了比较深的印象。
对于栈，刚开始学的时候只是知道这个东西，不太明白用处。现在虽然我还没学数据结构这个课，再回过头看这个逆波兰表达式或者计算器就感觉用栈是非常好的选择了。
原理不想讲太多，从左到右读取字符串，遇到操作数入栈，遇到运算符，根据该运算符选择出栈元素个数，进行运算后将结果压回栈中。因为逆波兰表达式的特点，最后栈中一定只剩一个操作数，就是答案。
这里说说 atoi 函数吧，我在两个月前才知道 algorithm 中有很多将 char 转化成 int/double 或者反过来的操作，感觉自己真是菜。哦对了，还有 c_str 函数，依稀记得自己曾经学过这个。
*/

class Solution {
public:
    /*
     * @param tokens: The Reverse Polish Notation
     * @return: the value
     */
    int evalRPN(vector<string> tokens) {
        // write your code here
        int n = tokens.size();
        if (tokens.empty()) return 0;
        stack<int> stack;
        for (int i = 0; i < n; i++) {

            if (tokens[i] != "+" && tokens[i] != "-" && tokens[i] != "*" && tokens[i] != "/")
                stack.push(atoi(tokens[i].c_str()));

            else {
                int num1 = stack.top();
```

```
                stack.pop();
                int num2 = stack.top();
                stack.pop();
                int num3 = 0;
                if (tokens[i] == "+") {
                    num3 = num2 + num1;
                }
                else if (tokens[i] == "-") {
                    num3 = num2 - num1;
                }
                else if (tokens[i] == "*") {
                    num3 = num2 * num1;
                }
                else if (tokens[i] == "/") {
                    num3 = num2 / num1;
                }
                stack.push(num3);
            }
        }
        return stack.top();
    }
};
```

LintCode-528 摊平嵌套的列表

/*这题还是挺有意思的，实际上只是要求我们写一个迭代器，给了我们很多函数了。
由于有 isInreger 和 getList 两个函数，那么我们在完成迭代器基本功能的基础上，只需要写一个摊平函数。
我们从左到右读取数组的每一个元素，判断为数字的话，那么肯定是直接尾插到答案。如果不是数字，那么就像处理栈一样，因为存在括号的嵌套，或者说是存在两个'['相邻的情况。
我们结合 getList 函数，直接递归的处理即可。*/

```
class NestedIterator {

    vector<int> flattenList;
    vector<int>::iterator p;

    void flatten(vector<NestedInteger> nestedList){
        int i;
        for(i=0;i<nestedList.size();i++){
            if(nestedList[i].isInteger())
                flattenList.push_back(nestedList[i].getInteger());
            else
                flatten(nestedList[i].getList());
```

```
            }
            return;
        }

public:
    NestedIterator(vector<NestedInteger> &nestedList) {
        // Initialize your data structure here.
        flatten(nestedList);
        p=flattenList.begin();
    }

    // @return {int} the next element in the iteration
    int next() {
        // Write your code here
        int result=*p;
        p++;
        return result;
    }

    // @return {boolean} true if the iteration has more element or false
    bool hasNext() {
        // Write your code here
        return (p!=flattenList.end());
    }
};
```

LintCode-575 字符串解码

/*这个字符串解码还是很有意思的，非常典型的栈结构。
挑战里说了，不用递归的方式，那么递归是怎样的呢？
以 3[2[ad]3[pf]]xyz 为例。我们把[xxx]看作一个基本结构，那么基本结构有两种，一种是单个字母 x/字母串 xxx，另一个就是这个。
读到一个基本结构做一种操作。读到 3 时，输出 3.读到[]时，递归调用函数，直到最后转换成另一个基本结构——一个字母/字母串。具体实现就不说了，大概就是 2[ad]->adad，同级函数还有 3[pf]->pfpfpf，相加后带回上一个函数……

那现在来看看栈的解法。定义一个缓存区，用来存储基本结构一个字母/字母串。从左往右，依次入栈。当读到"]"时，将栈中"["之后的元素都存入缓存区，然后再结合当前栈顶的数字，进行一些重复、翻转操作后重新入栈。
没有特别多的细节需要注意，毕竟通过率也有 37%，简单题。
提交的时候发现数字可能不止一位。。。*/

```
class Solution {
public:
```

```
    /**
     * @param s: an expression includes numbers, letters and brackets
     * @return: a string
     */
        string expressionExpand(string &s) {//"5[10[abcd]Ac20[abcde]]"
        // write your code here
          stack<char>sk;
        for(int i=0;i<s.size();++i){
            if(s[i]!=']')sk.push(s[i]);
            else {
                string tmp;
                while(sk.top()!='['){
                    tmp+=sk.top();
                    sk.pop();
                }
                sk.pop();
                string n="";
                while(!sk.empty()&&sk.top()>='0'&&sk.top()<='9')n+=sk.top(),sk.pop();
                reverse(n.begin(),n.end());
                reverse(tmp.begin(),tmp.end());

                int n1=atoi(n.c_str());
                while(n1--)
                for(string::iterator it=tmp.begin();it!=tmp.end();++it)
                    sk.push(*it);
            }
        }
        string ans;
        while(!sk.empty())
            ans+=sk.top(),sk.pop();
        reverse(ans.begin(),ans.end());
        return ans;
    }
};
```

LintCode-636 132 模式

/*只能想到三重循环遍历，那就暴力来吧。
三重循环思路就是枚举每一个数 A，在满足 A<B 的情况下，再枚举每一个数 B，最后查找是否有 C 满足，A<B<C。对于本题的 132 序列来说，A 是从左向右枚举，B 是从右向左枚举，C 是在 A、B 位置之间进行枚举。
这个解法我觉得还可以，但是网上看到了一种用栈维护的非常厉害的解法，一般涉及到栈的操作就莫名其妙容易晕啊，不得不说这个解法真的是非常的好。
下面我来给出自己的一些理解，代码直接 copy 的大佬的。

问题是找到三个数，实现"132"序列。将三个数的问题转化成两个，是解决问题的一般思路。我在之前的暴力想法是先确定"1"、"2"，然后再确定"3"。这样做的原因是，"3"是三个数中最大的，大小关系特殊，且位置是在正中间的，位置也很特殊。
但是其实还有一个比较特殊的分法，"1"、"32"。从大小来说，"1"是最小的，从位置来说，"1"在"32"的左边，因此我们可以自然的想到，从数组的右边进行遍历，找到一组"32"，然后继续遍历，再找到一个比"32"小的数就算结束了。
这其实还远远没有解决问题，稍微想想就知道，实现起来还是比较困难的。我们再思考一下，假设遍历"3"的位置，只要右边有数小于"3"，那么就可以去查找"1"。但是右边的小于"3"的数，也就是"2"的位置可能不止一个，那么很明显，贪心来看，让"2"在小于"3"的前提下，尽可能大的策略肯定是最优的。
总结一下，我们要实现的是：从右向左遍历，选取一个"3"并找到最大的"2"，然后继续遍历，如果有更大的"3"，那么原来的"3"就成为了"2"，从贪心的角度说，"3"越大，"2"越大的可能性就越大，存在"1"的可能性也就越大。继续遍历，如果找到了"1"，就结束。
关于确定尽可能大的"2"，这其实在栈的应用就非常的频繁了，栈的基本操作。*/

```
class Solution {
public:
    /**
     * @param nums: a list of n integers
     * @return: true if there is a 132 pattern or false
     */
    bool find132pattern(vector<int> &nums) {
        // write your code here
        if(nums.size()<3)return false;
        int B=INT_MIN;
        stack<int>sk;
        for(auto it=nums.rbegin();it!=nums.rend();++it)
        {
            if(B>*it)return true;
            else
                while(!sk.empty()&&*it>sk.top())B=sk.top(),sk.pop();

                sk.push(*it);
        }
        return false;
    }
};
```

LIntCode-685 数据流中第一个唯一的数字

/*给一个连续的数据流,写一个函数返回终止数字到达时的第一个唯一数字（包括终止数字）,

如果找不到这个终止数字, 返回 -1.
题目已经说了是第一个，那很显然可以用队列来解决，但是由于要统计出现的次数，即使是给队列开个辅助数组都不太方便，因此这里用 unordered_map 比较好。
在找到终止数字后跳出查找，然后因为 map 的性质，nums[number]就可以作为在 map 查询的上界。*/

```
class Solution {
public:
    /**
     * @param nums: a continuous stream of numbers
     * @param number: a number
     * @return: returns the first unique number
     */
    int firstUniqueNumber(vector<int> &nums, int number) {
        // Write your code here
        unordered_map<int,int> tmp;
        int ans = -1;
        for(int i = 0; i<nums.size(); ++i){
            tmp[nums[i]] ++;
            if(nums[i] == number) {
                ans = i;
                break;
            }
        }
        if(ans == -1) return -1;
        for(int j = 0; j<=ans; ++j)
            if(tmp[nums[j]] == 1)
                return nums[j];

        return -1;
    }
};
```

LintCode-834 移除多余字符

/*给定一个字符串 s 由小写字符组成，移除多余的字符使得每个字符只出现一次。你必须保证结果是字典序是最小的合法字符串。
想起来某个名为 set 的常用模板，但是还是不符合题意的。
和之前做的一个题有点像，要让字典序最小的话，那么删除哪个重复元素的结果是不同的。
字典序这个东西，我们知道，是根据首部比的，就和数字大小一样。考虑以下三种情况：
(1)待添加的字符字典序小于最后一个字符，且最后一个字符还有余量，则替换。
(2)待添加的字符字典序小于最后一个字符，但是最后一个字符没有余量，则直接添加待添加字符。
(3)待添加的字符字典序大于最后一个字符，直接添加。

这三点都是显然的。对于(1)来说，既然可以替换，那么此时替换成字典序最小的一定比不替换要优。对于(2）来说，因为不能改变相对顺序，如果没有余量了，只能直接添加。对于(3)来说，大于的情况就不用考虑太多了，直接添加就行。
贪心的证明是比较麻烦的，不证了。*/

```
class Solution {
public:
    /**
     * @param s: a string
     * @return: return a string
     */
    string removeDuplicateLetters(string &s) {
        // write your code here
        int len=s.size();
        string res="";
        vector<int> cnt(26,0);
        vector<bool> judge(26,false);
        for (auto i : s) {
            cnt[i-'a']++;
        }
        for(auto c : s)
        {
            cnt[c-'a']--;
            if(judge[c-'a']) continue;
            else
            {
                while(res.size()>0&&cnt[res.back()-'a']&&res.back()>=c)
                {
                    judge[res.back()-'a']=false;
                    res.pop_back();
                }
            }
            res.push_back(c);
            judge[c-'a']=true;
        }
        return res;
    }
};
```

LintCode-1001 小行星的碰撞

/*很有意思的一道题目，直接考虑两个数在不相等的前提下的所有情况。

元素依次入栈，只有栈顶是正数，当前是负数才会碰撞。负数时再比较绝对值，绝对值大于栈顶则会碰撞，而且可能和更前面的小行星碰撞，等于则会抵消。*/

```
class Solution {
public:
    vector<int> asteroidCollision(vector<int>& asteroids) {
        stack<int> sk;
        vector<int> ans;
        int len = asteroids.size();
        for(int i=0;i<len;i++){
            int aster = asteroids[i];
            if(aster > 0){
                sk.push(aster);
                continue;
            }
            if(!sk.empty()){
                aster = abs(aster);
                int t = sk.top();
                if(t > aster) continue;
                else if(t == aster) sk.pop();
                else{
                    while(!sk.empty() && sk.top()<aster) sk.pop();
                    if(sk.empty()) ans.push_back(-aster);
                    else if(sk.top() == aster) sk.pop();
                }
            }
            else ans.push_back(aster);
        }
        len = ans.size();
        while(!sk.empty()) ans.insert(ans.begin()+len,sk.top()),sk.pop();
        return ans;
    }
};
```

LintCode-1255 移除 K 位

/*想了一下如何用栈，贪心策略就出来了。
让栈是单调的，尽可能让结果的高位的数字小。那么如果当前元素大于栈顶，入栈，当前元素小于栈顶，出栈，直到栈为空或栈顶小于当前元素。
当前，出栈是有次数限制的，同时需要考虑前导零。新学了个 sring 的尾部操作，pop_back，但是不能 pop_front。*/

```
class Solution {
public:
```

```
    /**
     * @param num: a string
     * @param k: an integer
     * @return: return a string
     */
    string removeKdigits(string &num, int k) {
        // write your code here
        if(num.size()<=k)return "0";
        stack<int>sk;
        int count=0;
        for(int i=0;i<num.size();++i)
        {
            if(count<k)
            while(!sk.empty()&&num[i]<sk.top()&&count<k)
                count++,sk.pop();

            sk.push(num[i]);
        }
        while(count<k)sk.pop(),count++;
        string ans="";
        while(!sk.empty())ans+=sk.top(),sk.pop();
         while(ans[ans.size()-1]=='0')ans.pop_back();
         reverse(ans.begin(),ans.end());
         return ans==""?"0":ans;
    }
};
```

LintCode-1278 不超过 K 的最大矩阵元素之和

/*一点思路没有，甚至对题目的理解都抱有疑问。按照线索，花了一点时间去学习了矩阵的前缀和，dp 的思路很简单，dp[i][j]=dp[i-1][j]+dp[i][j-1]-dp[i-1][j-1]+a[i][j]。这个开始不太理解，结合图来说就很好理解了，矩阵的前缀和可以看做从左上角开始的一个矩阵，不解释了。刚才无意间看到了一个很有意思的 O(n3)的解法，开始也不太理解什么叫降维，但是拿手自己画画也就很快上手了。
趁热说下自己的理解：一位前缀和怎么求就不说了，那么我们可以通过降维的思想把二维转换为一维。对于一个矩阵，先用 f[i][j]=f[i-1][j]+a[i][j]这个式子求出列的前缀和，f[i][j]表示第 j 列，0~i 行的前缀和。
然后我们用两重循环去枚举两条横线，去分割矩阵。有个细节需要注意，我们分割矩阵的线是要画在数字上，并且数字的边界，也就是说，i 和 j 可以相等，表示画线的这一行。对于每个枚举的情况，再求出 0~n 列的前缀和，取最大值就可以了。
这个思想其实并非是第一次遇到，我在牛客上也用这样的方法解过贪心的题，确实感觉是很好的方法。
题目还要求小于 k，加个限制条件就行。
但是考虑到特殊情况太多，最后还是用了积分图法。*/

```
class Solution {
public:
    int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
        int m = matrix.size();
        int n = matrix[0].size();
        vector<vector<int>> dp(m, vector<int>(n, 0));
        dp[0][0] = matrix[0][0];
        for (int i = 1; i<m; ++i) {
            dp[i][0] = dp[i - 1][0] + matrix[i][0];
        }
        for (int i = 1; i<n; ++i) {
            dp[0][i] = dp[0][i - 1] + matrix[0][i];
        }
        for (int i = 1; i<m; ++i) {
            for (int j = 1; j<n; ++j) {
                dp[i][j] = matrix[i][j]+dp[i - 1][j] + dp[i][j - 1] - dp[i - 1][j - 1];
            }
        }
        int maxArea = INT_MIN;
        for (int a = 0; a<m; ++a) {
            for (int b = a; b<m; ++b) {
                for (int c = 0; c<n; ++c) {
                    for (int d = c; d<n; ++d) {
                        int topArea = a == 0 ? 0 : dp[a - 1][d];
                        int leftArea = c == 0 ? 0 : dp[b][c - 1];
                        int ltArea = (a == 0 || c == 0) ? 0 : dp[a - 1][c - 1];
                        int curSum = dp[b][d] - leftArea - topArea + ltArea;
                        if (curSum == k)
                            return k;
                        if (curSum<k)
                            maxArea = max(maxArea, curSum);
                    }
                }
            }
        }
        return maxArea;
    }
};
```