1. **基础**

**二分(递归 & 非递归)**

If left > right return;

mid = left + (right - left) / 2;

**快排**

partition //双指针

if left >= right return;

**归并排序**

If left >= right return;

mid = left + (right - left) / 2;

mergeSort(nums, left, mid - 1, tmp)

mergeSort(nums, mid, right, tmp)

merge(nums, left, mid, right, tmp) // nums => tmp

void merge(nums, left, mid, right, tmp) {

int p = left, q = mid;

Int s = left;

while(p < mid && q <= right) { //因为这边是while，所以下面是if即可

If (nums[p] < nums[q]) tmp[s++] = nums[p++];

else tmp[s++] = nums[q++];

}

while(p < mid) tmp[s++] = nums[p++];

while(q < right) tmp[s++] = nums[q++];

for(int i = left; i <= right; ++i) nums[i] = tmp[i];

return;

}

**反转链表(递归 & 非递归)**

非递归: pre, cur, next;

递归:

ListNode\* reverseList(ListNode\* head) {

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

ListNode\* revese = reverseList(head->next); //尾递归就是最后pHead返回，且不变。

head->next->next = head;

head->next = NULL;

return reverse;

}

**BFS(递归 & 非递归)**

非递归: 栈

**DFS(递归 & 非递归)**

非递归: 队列。 queue

注意：直接for 循环当前队列，不需要借助额外的内存空间

DP-最长递增子序列

**回溯 // 参考permutation/子集集合**

class Solution {

public:

    void dfs(vector<int>& nums, int i, vector<int>& curset, vector<vector<int>>& result) {

        if (i >= nums.size()) {

            result.push\_back(curset);

            return;

        }

        // two branches of the dfs tree, or more branches.

        // 0

        vector<int> save(curset); //保存状态

        dfs(nums, i + 1, curset, result);

        // 1

        curset = save;

        curset.push\_back(nums[i]);

        dfs(nums, i + 1, curset, result);

    }

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

        vector<vector<int>> result;

        vector<int> curset;

        dfs(nums, 0, curset, result);

        return result;

    }

};

简单trick

- 边界的确认: 举一些简单的例子。

1. **基础方法总结**
2. **滑动窗口类型:** 扩大j，缩小i，维持一个窗口

特点：输入是线性结构，比如数组/字符串，求**最长/最短的子串或满足特定要求的子串**。

比如: 最长无重复的子串。 //最长/最短，区别DP

方法: 滑动窗口(两个指针) + hash + 二分

**经典题目:**

1. [Subarray Sum Equals K](https://leetcode-cn.com/problems/subarray-sum-equals-k/): 求连续子序列和为K的个数。

Tips：因为求和，所以sum构成序列 + hashmap。

1. [Minimum Size Subarray Sum](https://leetcode-cn.com/problems/minimum-size-subarray-sum/): 连续子序列的和为s，全部为正数  //非常好的题目

Tips：

思路一、sum构成序列，每个sum，往前面找 target = sum - s(参考插入排序)，二分查找。

时间复杂度为O(N)。

思路二、两个指针进行窗口滑动，最初两个指针挨着，先j向右扩展，然后左边i收缩。O(N)

//典型的滑动窗口的思路

3) Longest Substring with at least  K times Characters (medium) #395

最少重复K次的最长子串，比如s = "ababbc", k = 2，ans=”ababb”

Tips：分而治之的方法，首先统计<K的index，然后以此为间隔 + 递归longestSubstring(因为分隔之后，计数可能<K)；

也可以用滑动窗口思路: 窗口内计数++，窗口外计数--。

1. Fruits into Baskets (medium)

最长子序列，其中只有两个元素。

Tips：哈希表 + 左右指针。

1. **两个指针类型 //感觉此类题目也挺考验基本功的。 两个指针 + 一个遍历，或者直接两个指针**

特点: 避免一个指针来回遍历。在**已排序**的数组/链表中, 找**一些组合满足某种限制条件。**

区别在于，没有窗口的概念。

「双指针」，当我们需要枚举数组中的两个元素时，如果我们发现随着第一个元素的递增，第二个元素是递减的，那么就可以使用双指针的方法

其中，组合可以是两个数，三个数，或者是子数组。  
sort(nums.begin(), nums.end());

1) [3Sum](https://leetcode-cn.com/problems/3sum/): 求三个数之和为0

思路一、sort + 两个指针 + hashmap，两个数确定条件下，第三个数直接可以查出;

思路二、sort + 三个指针。//第一个指针遍历，第二第三个指针分别为left/right，left向右，right向左。 // 因为a + b + c = 0; b变大时，c变小。 则b从小到大变化，同时c从大到小变化，并列关系。所以时间复杂度是O(N2)

2) 3Sum Closest: 求三个数之和最接近target

跟3sum的思路二是一样的，新增一个变量保存最接近的sum即可。其他移动都一样。

3) Triplets with Smaller Sum (medium): 三个数之和< target的个数。

类似

4）Subarrays with Product Less than a Target (medium)： 求连续子序列<k的所有个数。

#713 乘积小于K的子数组

Tips: 双指针 + 区间个数判断。 //因为连续，所以right指针可以依次后移，不走重复，right指针为主导，以元素结尾为连续子序列，O(N)完成。

1. Dutch National Flag Problem (medium): 三色旗问题

Tips: 双指示指针(一队头，一队尾) + 一个移动指针。 //队头队尾可以防止重叠

注意细节:

1. 队尾非2时，为last;  2) 交换last, i 条件；3）交换的为0 咋办；4）最后的两个指针需要交换不，细细分析后，不需要。  //所以一共是3个细节点。

// 边界条件，特殊化去思考；细节点多考虑一些cases

1. **快慢指针**

什么时候需要用快慢指针模式？

1. 问题需要处理**环上**的问题，比如环形链表和环形数组
2. 当你需要知道**链表的长度或是某个特别位置的信息**的时候

那啥时候用快慢指针而不是上面的双指针呢？

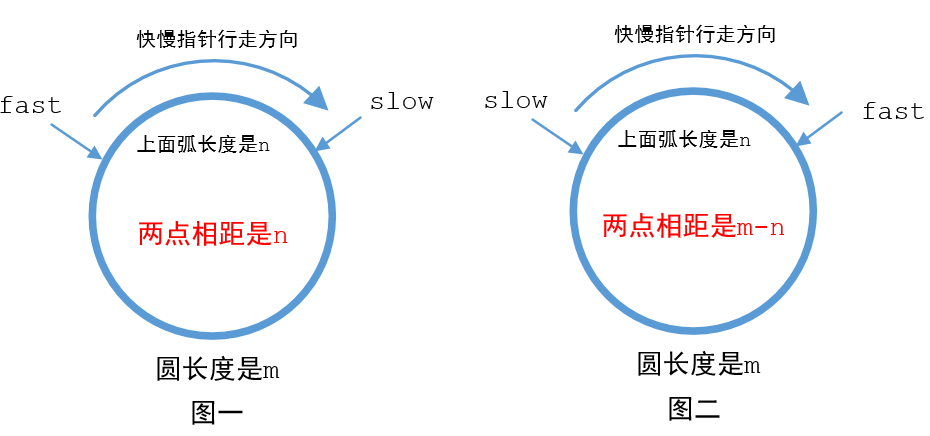
- 有些情形下，咱们不应该用双指针，比如我们在单链表上不能往回移动的时候。一个典型的需要用到快慢指针的模式的是当你需要去判断一个链表是否是回文的时候。

即快慢指针针对链表，可以理解为链表上的双指针。

1. LinkedList Cycle (easy)

两种思路: 1) hashmap，存放指针；2) 快指针和慢指针，快走2步和走一步。

2）的分析，快慢指针一定在环内时，且相距N，环长度为M （M> N）, 则左边图中的快每走一次，都与slow缩小一个距离，当N次，则相遇。右图则是M-N次。



b、倒数第K的节点: 快走K步，指向待删除的前一个指针。

**4、Pattern: Merge Intervals，区间合并类型【不常见】**

当你需要产生一堆相互之间没有交集的区间的时候

当你听到重叠区间的时候

a、Merge Intervals (medium)

Tips: 合并区间

思路一、6种区间合并情况都考虑到。 + flag标记 + 递归

思路二、排序 + 双指针  //双指针的威力

1. **Pattern: In-place Reversal of a LinkedList，链表翻转**

需要去翻转链表，要求不能使用额外空间的时候

1. Reverse a LinkedList (easy) //基本题，递归 & 非递归

Tips:

1. 非递归: pre, cur, next三个指针即可。
2. 递归: 注意递归函数返回的是尾部节点，用于最终返回，而左边的反转的节点，则直接可调用参数可以获得，这个得注意。

b、Reverse a Sub-list (medium): 在m, n 区间内反转链表。

使用pre, cur,next三指针 + 记录m-1, n+1的位置 + 一些case测试下边界即m, n 的各种情况。

注意m == n的情况

1. **Pattern: Tree Breadth First Search，BFS**

queue

a、Binary Tree Level Order Traversal (easy)  // 简单的中序层次遍历

思路: 队列 + vector存放flag (**或者直接for 循环当前队列，不需要借助额外的内存空间，后者比前者简单**)

**7、 Pattern: Tree Depth First Search，DFS**

DFS递归 或者 stack【注意，先right子树，后left子树】

a、[112. Path Sum](https://leetcode-cn.com/problems/path-sum/) (easy)

思路: 直接DFS，递归版本即可。

b、All Paths for a Sum (medium)

**[113. Path Sum II](https://leetcode-cn.com/problems/path-sum-ii/): 打印所有路径**

思路: 同上+ father[] // 用map记录节点的father, 最后reverse(vector)， AC一次过。

**8、Pattern: Modified Binary Search，改造过的二分**

模式: 已排序，或者创造条件排序

中点的计算= left + (right - left) / 2;

1. [704. Binary Search](https://leetcode-cn.com/problems/binary-search/) // 简单的二分，递归 & 非递归版本。

b、[34. Find First and Last Position of Element in Sorted Array](https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/): 查找目标出现的第一个，最后一个Index

Tips: 二分 + left-mid-righ遍历下就行。 //因为是两个下标。

另一种变形题目: 二分 + 判断是否左边不相等，如果相等，则right = mid - 1. //变形，第一个index。

**9、Pattern: Top ‘K’ Elements，前K个系列**

215. Kth Largest Element in an Array(中等)  // 重点掌握

参考qsort, 没问题。

注意: 只能击败5%的人，原因在于partition没有随机性，要么随机选择/或者三值中位数法，能达到击败95%的人。

特别注意，rand\_idx需要跟right交换下，不然导致污染rand\_idx中间数问题。

分析: 时间复杂度是O(N), 空间复杂度O(logN)

#### [347. Top K Frequent Elements](https://leetcode-cn.com/problems/top-k-frequent-elements/)(中等)

#### 思路: 1) topK largest  + hashmap; 2) 小顶堆，使用C++优先队列的STL。 //比小顶堆小则过滤掉。

两种方法优缺点: 前者需要全部载入内存；不适合大数据，后者可以输入流式计算，时间复杂度后者O(NlogK)

**10、Pattern: K-way merge，多路归并**

每当你的输入是K个排好序的数组，你就可以用堆来高效顺序遍历其中所有数组的所有元素。你可以将每个数组中最小的一个元素加入到最小堆中，从而得到全局最小值。

识别K路归并：

该问题的输入是排好序的数组，链表或是矩阵；

如果问题让咱们合并多个排好序的集合，或是需要找这些集合中最小的元素。

a) [21. Merge Two Sorted Lists](https://leetcode-cn.com/problems/merge-two-sorted-lists/)(简单)

思路: **新建一个node作为开始的节点**，这样省事，一次AC

b) Merge K Sorted Lists (难)

[23. Merge k Sorted Lists](https://leetcode-cn.com/problems/merge-k-sorted-lists/)

思路: **最小堆实现K路归并**， 一次AC。 //此题类似经典题可以看看

static bool cmp(pair<int, int>& m, pair<int, int>& n) {

        return m.second > n.second;

 }

priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(&cmp)> q(cmp);

1. **DP**

最优子结构 + 重复子问题，适用于解最优化问题。

两大特性:   【围绕递归树结构展开】  //一定是求最值，最长/最大/最少/最。。。

1. 重叠子问题。即递归树展开，很多节点都是一样；
2. 最优子结构。子问题之间是相互独立的，可以看成了树的不同分支。 然后由小规模的子问题的解，可以推导出更大规模问题的解。最优子结构性质作为动态规划问题的必要条件，一定是让你求最值的，以后碰到那种恶心人的最值题，思路往动态规划想就对了，这就是套路。

最重要：写出状态转移方程。

1. 递归跳出/终止条件。
2. 递归信任。即自顶向下分析，n状态可以由n-1, n-2等状态转移过来。

最后，使用递归+ 备忘录(自顶向下)，或者DP table(多重循环) 自底向上计算。

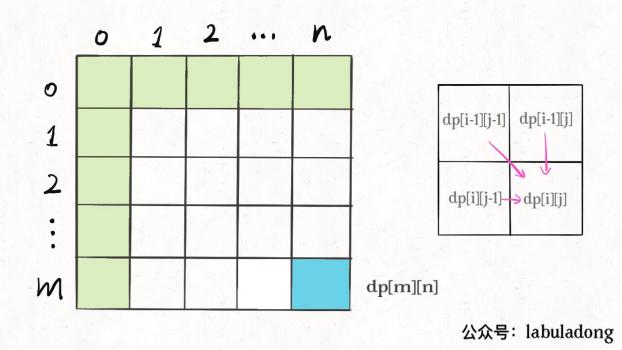
关于DP table 迭代遍历的方向:

1. 遍历的过程中，所需的状态必须是已经计算出来的。
2. 遍历的终点必须是存储结果的那个位置。

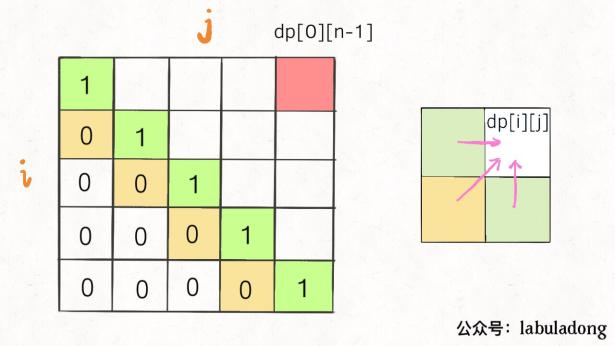
**动态规划的核心设计思想是数学归纳法。**

**即以上的递归信任，自顶向下分析。第n的状态跟前n-1 xxxx相关。**

1）从左往右，从上往下



1. 从下往上，从左往右



**经典题目**

1、最长递增子序列

状态数组(dp数组)定义: dp[i] 表示以第i元素结尾的最长递增子序列。

dp[i] = max(dp[k]) (k<i && nums[k] < nums[i]) + 1

N平方的时间复杂度。

评论: **这是一个第n状态跟前面所有状态都有关系的例子。**

2、最长公共子序列LCS

状态数组定义: dp[i][j]表示a[0,1, …,i] 与b[0, 1,…, j]的LCS。// 因为涉及两个子序列，所以定义dp[i][j]很容易想到。

dp[i][j] = dp[i-1][j-1] + 1  if a[i] == b[j]

dp[I][j] = dp[i-1][j], ap[i][j-1], if a[i] != b[j]

评论: 状态转移方程分情况（不同分支）而定。

变形: 最长公共子串

与最长公共子序列不同的是，必须连续，因此状态**dp[i][j]就定义为以i, j结尾**的公共子串长度。

比如AB ABC 这个dp=0(C!=B)， XAB AB，dp=2.

状态转移方程，只有当str1[i]=str2[j]，才发生转移。 //因为是以i, j结尾

初始状态，注意dp[i][0]， dp[0][j]的情况，其他初始为0。

评论: 当找到dp状态转移方程后，其比最长公共子序列简单。

总结: 其实两者之间差别不大。

3、最长回文子序列。

[b, x, a, b, y]

状态数组定义: dp[i][j]表示第i-j个元素间的最长回文子序列。 //为何定义二维，因为是回文，方便比较最外围的元素。

dp[i][j] = dp[i+1][j-1]   if a[i]== b[j]

dp[i][j] = max(dp[i][j-1], dp[i+1][j]) if a[i] != b[j]

跟LCS非常相似，注意i由i+1,这是因为状态定义有关，仍然是自顶向下分析。 //只是迭代遍历的方向不同。

变形: 最长回文子串

s = “babad” => “bab”

1）DP做的时候，特别注意是子串，而非子序列，所以dp状态定义的时候可以定义为dp[i][j]是否为子串，**其子串自然而然为i,j之间的元素**，因为连续，所以可以定义是否；而如果是子序列，可能不连续，dp[i][j]定义为i,j之间的最大回文子序列。 //注意两者区别。

然后，这题dp会超时。

 // dp[i][j] <=  d[i+1][j-1]  //判断s[i] == s[j]

 // 边界条件: dp[i][i] = true;

**通过从以上经典题目可以归纳出三个模板**

**1、第一种思路模板是一个一维的 dp 数组**：**在子数组 array[0..i] 中，我们要求的子序列（最长递增子序列）的长度是 dp[i]**。

2、**第二种思路模板是一个二维的 dp 数组**：

**2.1 涉及两个字符串/数组时**（比如最长公共子序列），dp 数组的含义如下：

**在子数组 arr1[0..i] 和子数组 arr2[0..j] 中，我们要求的子序列（最长公共子序列）长度为 dp[i][j]**。

**2.2 只涉及一个字符串/数组时**（比如本文要讲的最长回文子序列），dp 数组的含义如下：

**在子数组 array[i..j] 中//区间范围内，我们要求的子序列（最长回文子序列）的长度为 dp[i][j]**。

**2.3 背包问题。 dp[i][j]定义为可选择前i个元素时，能达到j的最优方案。  //背包套路。**

其中2.2相对特别些，不容易想到。可以想想是不是可以套用2.2的方法。

4、a) Equal Subset Sum Partition，相等子集划分问题

#### [416. Partition Equal Subset Sum](https://leetcode-cn.com/problems/partition-equal-subset-sum/)

思路: 转化为背包问题，找target = sum / 2的0-1背包。 dp[i][j]

注意：

1）初始化，dp[i][j]定义为可选择i个物品时重量和达到j是否可以。 且当第i个位nums[i]， 都为true；

2）二维数组定义。 Vector<vector<int>> dp(1001, vector<int>(2001, 0));

3）可以转为一维数组dp[j], 但是重要j的遍历从大到小，避免被覆盖。

5、Unbounded Knapsack，无限背包

找零钱问题: 给你 k 种面值的硬币，面值分别为 c1, c2 ... ck，每种硬币的数量无限，再给一个总金额 amount，问你最少需要几枚硬币凑出这个金额。

1. 确定状态: 原问题/子问题中变化的量，显然，是金额。dp[n]表示总额为n需要至少多少枚硬币。

dp[n] = argmin(dp[n- nums[i]] )  + 1 //这个式子中的i是所有硬币的遍历，即其不受N的限制，因为其为无限个，即每次都可以从0-N-1个中取硬币。 //第N跟之前所有状态发生联系。

dp[0] = 0

6、最长重复子串 【并非最值都是DP】

三种思路

1）二分 + hash。  //可叠加。  因为无非就是1 - n-1个长度，二分查找。 时间复杂度为O(NlogN)

长度是n， n/2有重复子串，则 n / 2- n 之间 二分。

已知查看长度为k, 则将str中所有k的字符串用hash存储，如果重复，则return True。  // unordered\_map

2）dp方法  O(n2)

dp[i][j]表示为str i, j 重复的子串的长度。

f(i,j)=f(i−1,j−1)+1

3）暴力解法 O(n2)

step从1- N-1

从第一个j开始遍历，跟j + step进行比较，记录max长度以及最后一个字符的位置。

7、股市交易:

已经知道每一天股票的价格，可以无限次交易，每天只能进行一次，询问最大收益。

**设f[i]表示到第i天的时候，最大收益是多少。**

首先f[i]可以直接从上一天继承过来，即f[i]= f[i-1]。

其次，如果在第i天卖出，那么需要选择一天买入，设第j天买入，则

f[i] = max(f[j-1] + a[i] - a[j]) // j = 0- i- 1 //又是一个第N跟前面所有状态有关的例子。

class Solution {

public:

    int maxProfit(vector<int>& prices) {

        int n = prices.size();

        int f[n + 10], mx = -prices[0];

        f[0] = 0;

        for (int i = 1; i < n; ++i) {

            f[i] = max(f[i - 1], mx + prices[i]);

            mx = max(mx, f[i - 1] - prices[i]);

        }

        return f[n  - 1];

    }

};

评论: 关键是状态的定义。以及转移方程定义。

8、最短编辑距离: 关键是状态的定义。

算法

1.动态规划：**dp[i][j]表示word1的前i个字符编辑成word2的前j个字符需要的最小操作数**

2.初始状态：dp[i][0] = i，i次删除；dp[0][i] = i，i次插入

3.过渡公式：

当i字符等于j字符时：dp[i][j] = dp[i-1][j-1]，不需要额外操作

当i字符不等于j字符时：dp[i][j] = Math.min(insert, delete, replace)

int insert = dp[i][j-1] + 1; i个编辑成j-1个字符，再插入一个j

int delete = dp[i-1][j] + 1; i-1个编辑成j个字母，再删除一个i

int replace = dp[i-1][j-1] + 1; i-1个编辑成j-1个字母，再将i替换成j

评论: 关键是状态的定义。

1. **其他类型，如剑指offer上题**
2. 矩阵的蛇形打印: 注意上下左右三个或者四个角标就行。四个循环

2、二叉树的最近公共祖先

//参考别人精简的代码。

class Solution {

public:

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

        if (root == NULL) return NULL;

        if (root->val == p->val || root->val == q->val) return root;

        TreeNode\* left = lowestCommonAncestor(root->left, p, q);

        TreeNode\* right = lowestCommonAncestor(root->right, p, q);

        if (left != NULL && right != NULL) return root;

        return left == NULL ? right : left;

    }

};

3、矩形覆盖

我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

分析第N种情况, 可以由N-1, 竖着放；或者N-2, 横着放，因此f(n) = f(n-1) + f(n-2);

1. 数值的整数次方

2(n) = 2 (n / 2) \* 2 (n/2) 快速求幂法

1. 二叉搜索树转为双向链表（好题）

理解二叉排序树的中序遍历。 + pLast作为全局变量

中序遍历时，刚好root前的前一个值是left中最大的；而right的最小值的恰好在root遍历完的下一个遍历。

因此，可以使用pLast全局变量指向cur的前一个遍历结点。

6、1-n中1出现的次数

再考虑一种新思路。统计某个位置上 1出现的次数。如34，1在十位上出现的次数是10次  
（10到19），1在个位上出现的次数是4次（1，11，21，31），因此34中1出现了14次。

对于整数n，将这个整数分为三部分：当前位数字cur，更高位数字high，更低位数字low，如：对于n=21034，当位数是十位时，cur=3，high=210，low=4。  
我们从个位到最高位 依次计算每个位置出现1的次数：  
在计算时，会出现三种情况  
1）当前位的数字等于0时，例如n=21034，在百位上的数字cur=0，百位上是1的情况有：00100-00199，01100-01199，……，20100-20199。一共有21\*100种情况，即high\*100;  
2）当前位的数字等于1时，例如n=21034，在千位上的数字cur=1，千位上是1的情况有：01000-01999，11000-11999，21000-21034。一共有2\*1000+（34+1）种情况，即high\*1000+(low+1)。  
3）当前位的数字大于1时，例如n=21034，在十位上的数字cur=3，十位上是1的情况有：00010-00019，……，21010-21019。一共有(210+1)\*10种情况，即(high+1)\*10。

1. 两个链表的公共结点

思路1、stack; 思路2、快慢指针，遍历长度，长度差则为快指针都走的步数。

1. 数字在排序数组中出现的次数

分别用二分算法确定上下界，下界为第一个index(这个问题之前遇到过)；上界则是第一个比之大的数，同样常规二分。（也是不错的题目）

1. 二叉树的下一个结点

要么是右子树的最左边的结点，要么就是父节点(第一次是left孩子)

public TreeLinkNode GetNext(TreeLinkNode pNode){

        if(pNode == null) return null;

        if(pNode.right != null){

            pNode = pNode.right;

            while(pNode.left != null)

                pNode = pNode.left;

            return pNode;

        }

        while(pNode.next != null){

            if(pNode.next.left == pNode)

                return pNode.next;

            pNode = pNode.next;

        }

        return null;

    }

1. 剪绳子分段乘积最大问题，dp问题  
           for (int i = 4; i <= n; i++) {

            int res=0;//记录最大的

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

                res=Math.max(res,dp[j]\*dp[i-j]);//这里需要是两个dp值的乘积，而不是我之前想的j\*dp[i-j]

            }

            dp[i]=res;

        }

        return dp[n];

1. 滑动窗口的最大值

直觉上，直接双指针滑动窗口，但是区间内的最值，时间复杂度有点高。可以用优先队列(即最大堆即可)

1. 二叉排序树的第K个节点

本质上就是中序遍历。

class Solution {

public:

    int m;

    TreeNode\* ans;

    void dfs(TreeNode\* p){

        if(!p || m < 1) return;//不满足条件直接返回NULL/每次递归出口：

        dfs(p -> left);//走到了最左边结点,到空不继续递归，该节点左右走完了回溯上一层

        if(m == 1) ans = p;//最左边结点 / m-到1的时候，当前结点就是第m小

        if(--m > 0) dfs(p -> right);// 右子树同样处理/遍历该节点的右节点 (左中右)

    }

    TreeNode\* KthNode(TreeNode\* p, unsigned int k){

        ans = NULL; m = k;//初始化 ans=NULL 不满足条件返回NULL

        dfs(p);

        return ans;

    }

};

1. **错题集合&分类**

1、Sort List:  [148. Sort List](https://leetcode-cn.com/problems/sort-list/)

链表的排序。

1）如果是用qsort对其val进行交换，节点指针不改变，则可以实现，但是会超时，即使随机。

直接用归并排序。

特别注意: fast,slow指针找到Middle之后，需要断开两个链表，然后各自排序，然后merge，会将他们连接起来。【链表找某个位置的数，借助快慢指针】

merge可以使用递归或者非递归。 //非递归借用一个虚拟节点简单化。

2、[Rotated Sorted Array](https://leetcode-cn.com/problems/search-in-rotated-sorted-array/):  [33. Search in Rotated Sorted Array](https://leetcode-cn.com/problems/search-in-rotated-sorted-array/) //二分的变体

nums = [4,5,6,7,0,1,2], target = 0

思路： 直接二分就行，比较mid, left，right， mid的左边/右边一定有一个是sorted，在sorted的部分先判断target是否在有序部分，在的话就是常规二分；不在的话就落在另一边，将问题缩小了一半规模，继续递归。

3、Given the head of a linked list, rotate the list to the right by k places.

思路很简单:  首先连接成循环链表，然后取k % len的节点，断开即可。 //其实思路简单。

注意: 因为是翻转而非反转。

4、找到字符串的最长无重复字符子串  // 能做出来

双指针(构成滑动窗口) + hash， I 移动，j = I + 1, 遇到重复，则 i = 重复位置。

1. **预测赢家**

甲乙轮流从数组的头部和尾部中取数，请问第一人取的最大和。

方法一、差值递归方法: 将最大和转为差值的计算。

因为AB两人都是最大化自己的收益，且最大化自己与别人的差值，这个是递归操作。

重点在于对差值的递归。

差值的递归可以理解为:

当前收益值： 当前元素A - 差值递归，即 A - (B - A) = A + A - B。 //最大化这个当前收益值。

private int play(int[] nums, int lo, int hi) {

if (lo > hi) return 0;

int planA = nums[lo] - play(nums, lo + 1, hi);

int planB = nums[hi] - play(nums, lo, hi - 1);

return Math.max(planA, planB);

}

求完差值后， n + m = sum和；n - m = 差值。即可以求出最大和。

方法二、dp方法: 对差值的最优化问题，每次都最大化差值(最优子结构)，将大问题分解为子问题。且两种前后选择则包含了重复子问题。

定义dp[i][j]为元素i到元素j之间的最大差值。

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

初始状态: dp[i][i] = nums[i], 且i < j。

刷题

1. 前序和中序重构二叉树(简单)

前序的第一个数，在中序中找出，然后left, right分别递归。

1. 数组的排列[重复 & 非重复情况]

class Solution {

public:

void swap(vector<int> &num, int i, int j) {

int tmp = num[i];

num[i] = num[j];

num[j] = tmp;

}

void permutation(vector<int>& num, int start, vector<vector<int>>& result) {

if (start >= num.size()) {

result.push\_back(num);

}

for (int i = start; i < num.size(); ++i) {

swap(num, i, start);

permutation(num, start + 1, result);

swap(num, i, start);

}

}

vector<vector<int> > permute(vector<int> &num) {

vector<vector<int> > result;

permutation(num, 0, result);

return result;

}

};

注意：有无重复数，差异在于多一条剪枝语句 if (i != idx && num[i] == num[idx]) continue; // 剪枝

1. 数字判断是回文

思路1、转为str；

思路2、后一半数字判断是不是跟前半段相等。

1. 将数组中0都移至队尾

partition思路。 两个指针O(N)

5、二叉树根与叶子最小的高度。

层次遍历，使用queue。 注意: for(q.size) //进队，则不需要visit数组

1. 最小覆盖子串: 给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。

两个指针的滑动窗口，left控制移除多余字符，right负责满足t要求，符合后记录min。不断更新min。

1. 查找有序二维数组的key

从左下角出发，二分即可。（这里的二分是选择行还是列，每次可以排除一半元素）

1. 删除链表倒数第K个节点。 //注意首部即可。快走K步。
2. 链表中节点每k个一组翻转。 //K翻转 + 递归。 确定好范围。
3. 数组中出现次数超过一半的数字

思路1：此数必为中位数，快排求中位数；

思路2：相互低效，最后剩余为此数字。

1. 二叉树的后序遍历是否正确。 from 剑指offer

解题思路篇

1. 单链表的排序
2. 值排序，放进vector，最后更改链表值。
3. 归并排序，快慢指针，找到中点，然后断开。分治。
4. 水位变化

给定一个整形数组arr，已知其中所有的值都是非负的，将这个数组看作一个容器，请返回容器能装多少水。

思路: left, right双向指针，定位当前标尺，且标尺发生变化(标尺通过当前值跟left, right比)。

public static long maxWater(int[] arr) {

    if (arr == null || arr.length <= 2) {

        return 0;

    }

    int left = 0, right = arr.length - 1;

    long sum = 0;

    // 找出左右边界的最小值作为水位高度

    int mark = Math.min(arr[left], arr[right]);

    while (left < right) {

        // 如果左边较低，则左边界向右遍历， 否则右边界向左移动

        if (arr[left] < arr[right]) {

            left++;

            // 如果当前标尺小于水位，则水量累加

            if (arr[left] < mark) {

                sum += mark - arr[left];

            } else { // 否则，将此标尺和右边边界高度进行比较，找出剩下数组中的新水位

                mark = Math.min(arr[left], arr[right]);

            }

        } else {

            right--;

            // 同理，如果当前标尺小于水位，则水量累加

            if (arr[right] < mark) {

                sum += mark - arr[right];

            } else { // 否则，将此标尺和左边界的高度进行比较，找出剩余数组中的新水位

                mark = Math.min(arr[right], arr[left]);

            }

        }

    }

    return sum;

}

1. 设计LRU的缓存机制(周末刷下)

思路上，第一反应是优先队列，但是细想，队列的调整是O(logK)，且每次其实只需要调整最近使用的那个节点，把其放置队尾，队头则是最不常被使用的节点。

相比数组，链表在调整结点位置上天然适合，O(1)操作即可完成。

Hashmap，从key到ListNode查询，以便调整其至队尾。

为何使用双向链表，因为调整时，方便指向其pre。

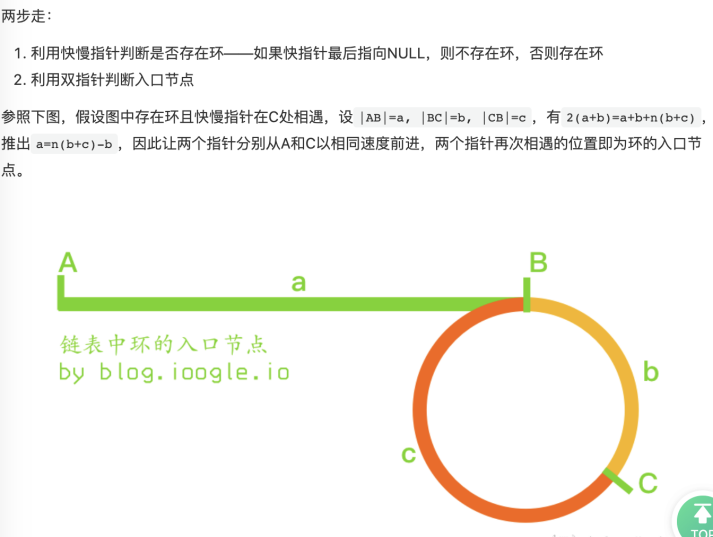
Tips： 1）因涉及删除操作，使用虚拟队头；2）使用cur记录队尾。【有点麻烦，最好方式是也使用虚拟队尾】

set(): 如果> capacity, 则队头删除。插入新节点至队尾。

get(): 将访问节点调整至队尾，返回val。

1. 括号序列: 栈数据结构可以搞定。

## 链表中环的入口节点: 最终的a = n(b + c) -b = n(b + c) + c



1. 求平方根【代码可以写下】

1- x/ 2 的二分查找。 满足 k \* k <= num