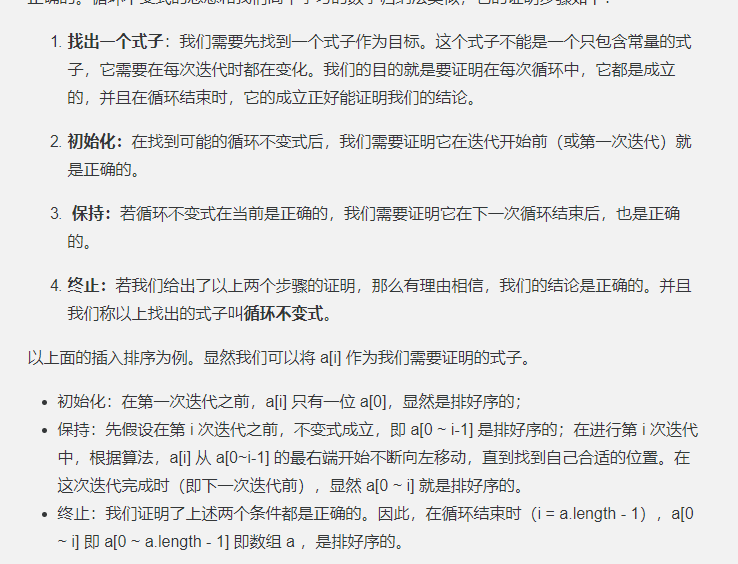
一些总结：

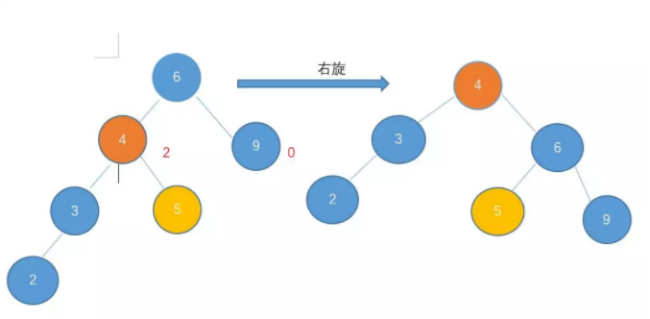
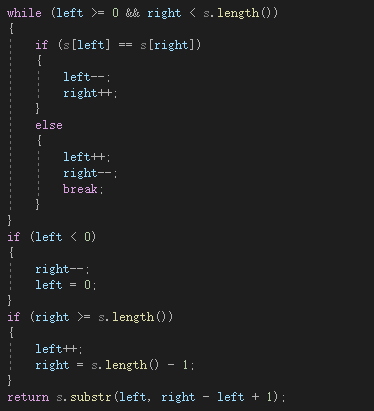
**常用算法总结：**

**快慢指针，哑结点，滑动窗口，中心扩展（回文），单调栈，**

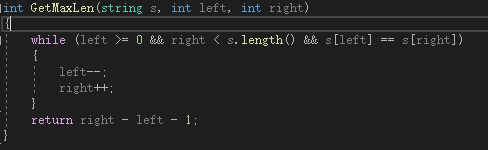
1. 学会使用哑节点来解决一些不好解决的边界问题，同理也要学会在列表中添加一些值，来达到简化操作的目的（时隔挺久，还是把哑结点使用给忘了，本来想打开文档记录一下哑结点的使用，结果发现第一条就是）
2. 一定要好好利用双指针，一切带有查找要素的算法，很多其实都可以用双指针来简化
3. 判断算法是否正确，用循环不变式来证明



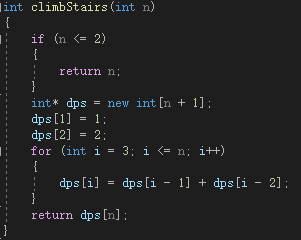
<https://www.cnblogs.com/dongkuo/p/4782669.html>

1. 堆：用来解决前K大问题（大数据求前K最大）
2. 堆：
   1. 堆的最后一个非叶子节点是 2\*len – 1
   2. 堆的子节点是 2 \* x + 1 和 2\*x + 2
3. 树：
   1. 节点的度：拥有子树的数目
   2. 叶子：度为0的结点
   3. 树的度：树中结点的最大的度
   4. 层次：根节点层级为1，后面+1
   5. 高度：最大层次
   6. 二叉树：
      1. 第i层的结点数目最多为pow(2, I - 1)
      2. 深度为k的二叉树最多有pow（2， k） - 1个节点
      3. 包含n个节点的二叉树高度至少为log2(n+1)
      4. 二叉树中，设叶子节点数为n0，度为2的结点数为n2，则n0=n2+1
      5. 满二叉树：都是满的
      6. **完全二叉树**：只有最后两层度可以小于2，并且最下一层的叶子节点集中在靠左的位置
      7. **二叉查找树**：
         1. 左孩子比父节点小，右孩子比父节点打，有一个特性是，通过**中序遍历**可以让节点有序
         2. 插入的时候直接找到最后一个小于或者大于的结点，然后替换
         3. 删除时需要考虑三种情况，无孩子，一个孩子，两个孩子，两个孩子的时候需要把右子树提上来
      8. **平衡二叉树：**
         1. 具有二叉查找树的全部特性
         2. 每个节点的左子树和右子树高度差至多等于1
         3. 
         4. 左-左：右旋
         5. 右-右：左旋
         6. 左-右：先左旋，后右旋
         7. 右-左：先右旋，后左旋
      9. **2-3树**
         1. 每个节点都有两个孩子（称为2节点）或三个孩子（称为3节点）
         2. 要么有两个或者三个孩子，要么一个孩子都没有
         3. 左子树小于较小元素，右子树大于较大元素，如果是3节点则存在中子树，大小介于两元素之间
         4. 所有叶子都在同一层级上
         5. 创建规则
            1. 加入新节点时，不会往空的位置添加节点，而是添加到最后一个叶子节点上
            2. 四节点可以被分解为三个2节点组成的树，并且分解后新树的根节点需要向上和父节点融合
      10. **红黑树：**
          1. 平衡二叉树最主要的问题是第二条规定太严格，导致每次插入新数据几乎都会破坏第二条规定，进而需要通过左旋和右旋来进行调整，性能问题很大
          2. 特性：
             1. 每个节点或是红色或是黑色
             2. 根节点是黑色（根节点要么是2要么是3，而这两种变换都会造成根节点是黑色）
             3. 每个叶子节点是黑色（这里的叶子是指为空的叶子节点）
             4. 如果一个节点是红色，则子节点必须是黑色的（由于红节点必然连接一个2或3，必是黑色）
             5. 从任意一个节点到叶子节点，经过的黑色节点是一样多的（由于红黑树是2-3树转变而来，因此每个黑色节点必然对应一个2或3，而2-3树的一条重要特性是平衡，所以黑色节点数量一定是一样的）
          3. 创建规则：
             1. 红黑树由2-3树转换而来
             2. 将2节点转换为红黑树的一个黑节点
             3. 将3节点拆开，左元素作为右元素的子树，并将左元素标记为红色
          4. 直接创建规则：
             1. 插入的节点都为红
             2. 如果为根节点则转为黑
             3. 红色节点只能为左节点，如果插到右节点需要左旋
4. 算最大回文串的时候有几个关键技巧：
   1. 我废了很大篇幅，去处理边界问题，一直在考虑越界

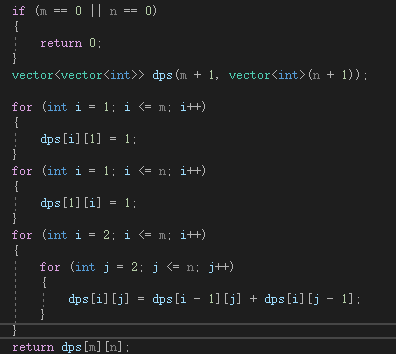
但是其实不用这么麻烦，还是没有思考本质，我一直在考虑的都是加入左右不等了，那多加的就要弄回去，如果越界了，那么就要纠正，但是其实并不需要这么做，**可以默认允许越界，也允许多加，但是最后的结果再减去多的就可以了**，正常的长度是r-l+1，允许多加后，结果写成r-l-1就可以了，也就是多减了2，以下是优化后的代码，很简洁了

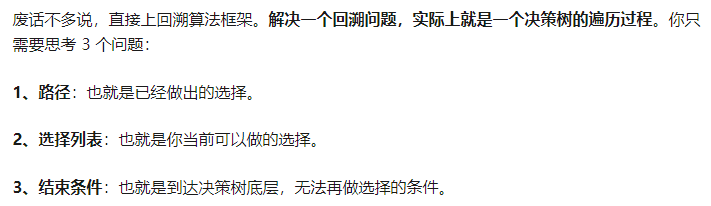
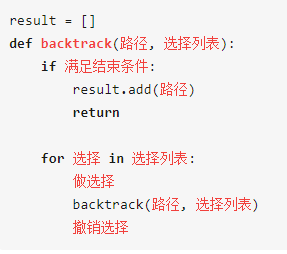


1. **动态规划总结：**
   1. 动态规划就是存一个备忘录，然后通过等价关系式推导出结果
   2. 套路：
      1. 先确定数组的含义
      2. 找出数组元素间的关系式
      3. 确定初始条件
   3. 例子1：跳台阶，一次可以跳一阶或两阶，问跳到m阶有几种跳法
      1. 这里只需要一维数组就可以，dps[m]代表跳到m阶的所有跳法数
      2. 跳到m阶有两种可能，一种是从m-1阶跳上来，另一种从m-2跳上来，所以dps[m] = dps[m-1] + dps[m - 2]
      3. 当m=0时，为0，m=1时为1，这里别忘了加m=2，为2



* 1. 例子2：路径总数，mxn的方格，从左上角走到右下角的所有可能路径树，只能向右或下走
     1. dps[m][n]为走到这里的总路径数
     2. dps[m][n] = dps[m-1][n] + dps[m][n-1]
     3. 当m=1或n=1时，需要初始化为1



1. 回溯法总结：
   1. 
2. 滑动窗口问题
   1. 刚开始做总想一步做到最好，所以一直在考虑跳跃滑动，而不是一步一步滑动，导致做个一个多小时提交了10次都没过…
   2. 滑动窗口最清晰的做法是
      1. 先滑动右窗口，直到满足条件
      2. 然后滑动左窗口来算最优条件
      3. 当左窗口不满足条件时，继续滑动右窗口
      4. 以此类推，直到右窗口超出范围