Algorithm

http://algorithm.yuanbin.me

目录

1. Preface
2. Part I - Basics
3. Basics Data Structure
    i. String
    ii. Linked List
    iii. Binary Tree
    iv. Huffman Compression
    v. Queue
    vi. Heap
    vii. Stack
    viii. Set
    ix. Map
4. Basics Sorting
    i. Bubble Sort
    ii. Selection Sort
    iii. Insertion Sort
    iv. Merge Sort
    v. Quick Sort
    vi. Heap Sort
    vii. Bucket Sort
    viii. Counting Sort
    ix. Radix Sort
5. Basics Algorithm
    i. Divide and Conquer
6. Basics Misc
    i. Bit Manipulation
    ii. Knapsack
7. Part II - Coding
8. String
    i. strStr
    ii. Two Strings Are Anagrams
    iii. Compare Strings
    iv. Anagrams
    v. Longest Common Substring
    vi. Rotate String
    vii. Reverse Words in a String
    viii. Valid Palindrome
    ix. Longest Palindromic Substring
    x. Space Replacement
    xi. Wildcard Matching
    xii. Length of Last Word
    xiii. Count and Say

9. Integer Array
    i. Remove Element
    ii. Zero Sum Subarray
    iii. Subarray Sum K
    iv. Subarray Sum Closest
    v. Recover Rotated Sorted Array
    vi. Product of Array Exclude Itself
    vii. Partition Array
    viii. First Missing Positive
    ix. 2 Sum
    x. 3 Sum
    xi. 3 Sum Closest
    xii. Remove Duplicates from Sorted Array
    xiii. Remove Duplicates from Sorted Array II
    xiv. Merge Sorted Array
    xv. Merge Sorted Array II
    xvi. Median
    xvii. Partition Array by Odd and Even
    xviii. Kth Largest Element
10. Binary Search
    i. Binary Search
    ii. Search Insert Position
    iii. Search for a Range
    iv. First Bad Version
    v. Search a 2D Matrix
    vi. Find Peak Element
    vii. Search in Rotated Sorted Array
    viii. Find Minimum in Rotated Sorted Array
    ix. Search a 2D Matrix II
    x. Median of two Sorted Arrays
    xi. Sqrt x
    xii. Wood Cut
11. Math and Bit Manipulation
    i. Single Number
    ii. Single Number II
    iii. Single Number III
    iv. O1 Check Power of 2
    v. Convert Integer A to Integer B
    vi. Factorial Trailing Zeroes
    vii. Unique Binary Search Trees
    viii. Update Bits
    ix. Fast Power
    x. Hash Function
    xi. Count 1 in Binary
    xii. Fibonacci
    xiii. A plus B Problem

xiv. Print Numbers by Recursion
xv. Majority Number
xvi. Majority Number II
xvii. Majority Number III
xviii. Digit Counts
xix. Ugly Number
xx. Plus One

12. Linked List
    i. Remove Duplicates from Sorted List
    ii. Remove Duplicates from Sorted List II
    iii. Remove Duplicates from Unsorted List
    iv. Partition List
    v. Two Lists Sum
    vi. Two Lists Sum Advanced
    vii. Remove Nth Node From End of List
    viii. Linked List Cycle
    ix. Linked List Cycle II
    x. Reverse Linked List
    xi. Reverse Linked List II
    xii. Merge Two Sorted Lists
    xiii. Merge k Sorted Lists
    xiv. Reorder List
    xv. Copy List with Random Pointer
    xvi. Sort List
    xvii. Insertion Sort List
    xviii. Check if a singly linked list is palindrome
    xix. Delete Node in the Middle of Singly Linked List
13. Binary Tree
    i. Binary Tree Preorder Traversal
    ii. Binary Tree Inorder Traversal
    iii. Binary Tree Postorder Traversal
    iv. Binary Tree Level Order Traversal
    v. Binary Tree Level Order Traversal II
    vi. Maximum Depth of Binary Tree
    vii. Balanced Binary Tree
    viii. Binary Tree Maximum Path Sum
    ix. Lowest Common Ancestor
    x. Invert Binary Tree
    xi. Diameter of a Binary Tree
    xii. Construct Binary Tree from Preorder and Inorder Traversal
    xiii. Construct Binary Tree from Inorder and Postorder Traversal
    xiv. Subtree
    xv. Binary Tree Zigzag Level Order Traversal
    xvi. Binary Tree Serialization
14. Binary Search Tree
    i. Insert Node in a Binary Search Tree

    ii. Validate Binary Search Tree
    iii. Search Range in Binary Search Tree
    iv. Convert Sorted Array to Binary Search Tree
    v. Convert Sorted List to Binary Search Tree
    vi. Binary Search Tree Iterator
15. Exhaustive Search
    i. Subsets
    ii. Unique Subsets
    iii. Permutations
    iv. Unique Permutations
    v. Next Permutation
    vi. Previous Permuation
    vii. Unique Binary Search Trees II
    viii. Permutation Index
    ix. Permutation Index II
    x. Permutation Sequence
    xi. Palindrome Partitioning
    xii. Combinations
    xiii. Combination Sum
    xiv. Combination Sum II
    xv. Minimum Depth of Binary Tree
16. Dynamic Programming
    i. Triangle
    ii. Backpack
    iii. Minimum Path Sum
    iv. Unique Paths
    v. Unique Paths II
    vi. Climbing Stairs
    vii. Jump Game
    viii. Word Break
    ix. Longest Increasing Subsequence
    x. Palindrome Partitioning II
    xi. Longest Common Subsequence
    xii. Edit Distance
    xiii. Jump Game II
    xiv. Best Time to Buy and Sell Stock
    xv. Best Time to Buy and Sell Stock II
    xvi. Best Time to Buy and Sell Stock III
    xvii. Best Time to Buy and Sell Stock IV
    xviii. Distinct Subsequences
    xix. Interleaving String
    xx. Maximum Subarray
    xxi. Maximum Subarray II
    xxii. Longest Increasing Continuous subsequence
    xxiii. Longest Increasing Continuous subsequence II
17. Graph

    i. Topological Sorting
    ii. Word Ladder
18. Data Structure
    i. Implement Queue by Two Stacks
    ii. Min Stack
    iii. Sliding Window Maximum
    iv. Longest Words
    v. Heapify
19. Problem Misc
    i. Nuts and Bolts Problem
    ii. String to Integer
    iii. Insert Interval
    iv. Merge Intervals
    v. Minimum Subarray
    vi. Matrix Zigzag Traversal
    vii. Valid Sudoku
    viii. Add Binary
    ix. Reverse Integer
    x. Gray Code
20. Part III - Contest
21. Google APAC
    i. APAC 2015 Round B
        i. Problem A. Password Attacker
22. Appendix I Interview and Resume
    i. Interview
    ii. Resume
23. 术语表

数据结构与算法/leetcode/lintcode题解

GITTER JOIN CHAT → build passing

简介

本文档为数据结构和算法学习笔记，全文大致分为以下三大部分：

1. Part I为数据结构和算法基础，介绍一些基础的排序/链表/基础算法
2. Part II为 OJ 上的编程题目实战，按题目的内容分章节编写，主要来源为 https://leetcode.com/ 和 http://www.lintcode.com/.
3. Part III 为附录部分，包含如何写简历和其他附加材料

本文参考了很多教材和博客，凡参考过的几乎都给出明确链接，如果不小心忘记了，请不要吝惜你的评论和issue :)

本项目托管在 https://github.com/billryan/algorithm-exercise 由 Gitbook 渲染生成 HTML 页面。你可以在 GitHub 中 star 该项目查看更新，RSS 种子功能正在开发中。

你可以在线或者离线查看/搜索本文档，以下方式任君选择~

- 在线阅读(由 Gitbook 渲染) http://algorithm.yuanbin.me
- 离线阅读: 推送到GitHub后会触发 travis-ci 的编译，相应的部分编译输出提供 GitHub 和 GitCafe 下载。
    1. EPUB: GitHub, Gitbook, GitCafe(中国大陆用户适用) - 适合在 iPhone/iPad/MAC 上离线查看，实测效果极好。
    2. PDF: GitHub, Gitbook, GitCafe(中国大陆用户适用) - 推荐下载GitHub 和 GitCafe 的版本，Gitbook 官方使用的中文字体有点问题。
    3. MOBI: GitHub, Gitbook, GitCafe(中国大陆用户适用) - Kindle 专用，未测试，感觉不适合在 Kindle 上看此类书籍，尽管 Kindle 的屏幕对眼睛很好...

- Google 站内搜索: `keywords site:algorithm.yuanbin.me`
- Swiftype 站内搜索: 可使用网页右下方的 `Search this site` 进行站内搜索

许可证

本作品采用 **知识共享署名-相同方式共享 4.0 国际许可协议** 进行许可。**传播此文档时请注意遵循以上许可协议。** 关于本许可证的更多详情可参考 http://creativecommons.org/licenses/by-sa/4.0/

本着独乐乐不如众乐乐的开源精神，我将自己的算法学习笔记公开和小伙伴们讨论，希望高手们不吝赐教。

多国语言

- English maintained by who?

- 简体中文 maintained by @billryan
- 繁體中文 maintained by @CrossLuna

如何贡献

如果你发现任何有错误的地方或是想更新/翻译本文档，请毫不犹豫地猛击 贡献指南.

捐助

添加这一小节其实是有点诚惶诚恐的，毕竟这本小书目前还很不完善，把捐助信息贴出来不脸红吗？:-(但既然前些天有网友专门发邮件来问这个事，我就大概说下我目前的想法，先以 @billryan 为例，其他 Contributor 后续补充，捐助者后期会单独整理公布，当然这是在征得捐助人同意的前提下进行的。除了在 GitHub 上协助编写文档外，你还可以以下面几种方式回馈各位贡献者:

邮寄明信片

@billryan 喜欢收集各种明信片，来者不拒~ 邮寄的话可以邮寄至 `上海市闵行区上海交通大学闵行校区电院群楼5号楼307` ，这个地址2016 年3月前有效，收件人： `袁斌` 。

送书

除了邮寄明信片，你还可以买本书送给各位贡献者，@billryan 的地址见上节。

支付宝打赏

金额随意。

PayPal

账户名：yuanbin2014(at)gmail.com 付款时选择 friends and family

金额随意。

如何练习算法

虽说练习算法偏向于算法本身，但是好的代码风格还是很有必要的。粗略可分为以下几点：

- 代码块可为三大块：异常处理（空串和边界处理），主体，返回
- 代码风格(**可参考Google的编程语言规范**)
  1. 变量名的命名(有意义的变量名)
  2. 缩进(语句块)
  3. 空格(运算符两边)
  4. 代码可读性(即使if语句只有一句也要加花括号)

- 《代码大全》中给出的参考

而对于实战算法的过程中，我们可以采取如下策略：

1. 总结归类相似题目
2. 找出适合同一类题目的模板程序
3. 对基础题熟练掌握

以下整理了一些最近练习算法的网站资源，和大家共享之。

在线OJ及部分题解

- LeetCode Online Judge - 找工作方面非常出名的一个OJ，每道题都有 discuss 页面，可以看别人分享的代码和讨论，很有参考价值，相应的题解非常多。不过在线代码编辑框不太好用，写着写着框就拉下来了，最近评测速度比 lintcode 快很多，而且做完后可以看自己代码的运行时间分布，首推此 OJ 刷面试相关的题。
- LintCode | Coding interview questions online training system - 和leetcode类似的在线OJ，但是筛选和写代码时比较方便，左边为题目，右边为代码框。还可以在 `source` 处选择 CC150 或者其他来源的题。会根据系统locale选择中文或者英文，可以拿此 OJ 辅助 leetcode 进行练习。
- LeetCode题解 - GitBook - 题解部分十分详细，比较容易理解，但部分题目不全。
- FreeTymeKiyan/LeetCode-Sol-Res - Clean, Understandable Solutions and Resources on LeetCode Online Judge Algorithms Problems.
- soulmachine/leetcode - 含C++和Java两个版本的题解。
- Woodstock Blog - IT，算法及面试。有知识点及类型题总结，特别赞。
- ITint5 | 专注于IT面试 - 文章质量很高，也有部分公司面试题评测。
- Acm之家,专业的ACM学习网站 - 各类题解
- 牛客网-专业IT笔试面试备考平台,最全求职题库,全面提升IT编程能力 - 国内一个IT求职方面的综合性网站，比较适合想在国内求职的看看。感谢某位美女的推荐 :)

其他资源

- 九章算法 - 代码质量大多不错，但是不太全。这家也提供有偿辅导。
- 七月算法 - julyedu.com - july大神主导的在线算法辅导。
- 刷题 | 一亩三分地论坛 - 时不时就会有惊喜放出。
- VisuAlgo - visualising data structures and algorithms through animation - 相当碉堡的数据结构和算法可视化。
- Data Structure Visualization - 同上，非常好的动画演示！！涵盖了常用的各种数据结构/排序/算法。
- 结构之法 算法之道 - 不得不服！
- julycoding/The-Art-Of-Programming-By-July - 程序员面试艺术的电子版
- 程序员面试、算法研究、编程艺术、红黑树、数据挖掘5大系列集锦
- 专栏：算法笔记——《算法设计与分析》 - CSDN上对《算法设计与分析》一书的学习笔记。
- 我的算法学习之路 - Lucida - Google 工程师的算法学习经验分享。

书籍推荐

本节后三项参考自九章微信分享，谢过。

- Algorithm Design (豆瓣)
- The Algorithm Design Manual, 作者还放出了自己上课的视频和slides - Skiena's Audio Lectures，The Algorithm Design Manual (豆瓣)
- 大部头有 *Introduction to Algorithm* 和 TAOCP
- *Cracking The Coding Interview*. 著名的CTCI(又称CC150)，Google, Mircosoft, LinkedIn 前HR离职之后写的书，从很全面的角度剖析了面试的各个环节和题目。除了算法数据结构等题以外，还包含OO Design, Database, System Design, Brain Teaser等类型的题目。**准备北美面试的同学一定要看。**
- 剑指*Offer*。适合国内找工作的同学看看，英文版叫Coding Interviews. 作者是何海涛(Harry He)。Amazon上可以买到。有大概50多题，题目的分析比较全面，会从面试官的角度给出很多的建议和show各种坑。
- 进军硅谷 -- 程序员面试揭秘。有差不多150题。

Part I - Basics

第一节主要总结一些算法要数据结构方面的基础知识，如基本的数据结构和基础算法。

本节主要由以下章节构成。

Reference

- VisuAlgo - visualising data structures and algorithms through animation - 相当碉堡的数据结构和算法可视化。
- Data Structure Visualization - 相当碉堡的动画演示！！ 涵盖了常用的各种数据结构/排序/算法。

Data Structure - 数据结构

本章主要介绍一些基本的数据结构和算法。

String

String 相关的题常出现在面试题中，这里总结下 C++, Java, Python 中字符串常用的方法。

Java

```
String s1 = new String();
String s2 = "billryan";
int s2Len = s2.length();
s2.substring(4, 8); // return "ryan"
StringBuilder sb = new StringBuilder(s2.substring(4, 8));
sb.append("bill");
String s2New = sb.toString(); // return "ryanbill"
// convert String to char array
char[] s2Char = s2.toCharArray();
// char at index 4
char ch = s2.charAt(4); // return 'r'
// find index at first
int index = s2.indexOf('r'); // return 4. if not found, return -1
```

StringBuffer 与 StringBuilder, 前者保证线程安全，后者不是，但单线程下效率高一些，一般使用 StringBuilder.

Linked List - 链表

链表是线性表的一种。线性表是最基本、最简单、也是最常用的一种数据结构。线性表中数据元素之间的关系是一对一的关系，即除了第一个和最后一个数据元素之外，其它数据元素都是首尾相接的。线性表有两种存储方式，一种是顺序存储结构，另一种是链式存储结构。我们常用的数组就是一种典型的顺序存储结构。

相反，链式存储结构就是两个相邻的元素在内存中可能不是相邻的，每一个元素都有一个指针域，指针域一般是存储着到下一个元素的指针。这种存储方式的**优点**是插入和删除的时间复杂度为 O(1)，不会浪费太多内存，添加元素的时候才会申请内存，删除元素会释放内存。缺点是访问的时间复杂度最坏为 O(n)。

顺序表的特性是随机读取，也就是访问一个元素的时间复杂度是O(1)，链式表的特性是插入和删除的时间复杂度为O(1)。

链表就是链式存储的线性表。根据指针域的不同，链表分为单向链表、双向链表、循环链表等等。

编程实现

Java

```
public class ListNode {
    public int val;
    public ListNode next;
    public ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}
```

链表的基本操作

反转链表

链表的基本形式是：`1 -> 2 -> 3 -> null`，反转需要变为 `3 -> 2 -> 1 -> null`。这里要注意：

- 访问某个节点 curt.next 时，要检验 curt 是否为 null。
- 要把反转后的最后一个节点（即反转前的第一个节点）指向 null。

```
public ListNode reverse(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode next = head.next;
        head.next = prev;
        prev = head;
        head = next;
    }
```

```
    return prev;
}
```

删除链表中的某个节点

删除链表中的某个节点一定需要知道这个点的前继节点，所以需要一直有指针指向前继节点。还有一种删除是伪删除，是指复制一个和要删除节点值一样的节点，然后删除，这样就不必知道其真正的前继节点了。

然后只需要把 `prev -> next = prev -> next -> next` 即可。但是由于链表表头可能在这个过程中产生变化，导致我们需要一些特别的技巧去处理这种情况。就是下面提到的 Dummy Node。

链表指针的鲁棒性

综合上面讨论的两种基本操作，链表操作时的鲁棒性问题主要包含两个情况：

- 当访问链表中某个节点 curt.next 时，一定要先判断 curt 是否为 null。
- 全部操作结束后，判断是否有环；若有环，则置其中一端为 null。

Dummy Node

Dummy node 是链表问题中一个重要的技巧，中文翻译叫“哑节点”或者“假人头结点”。

Dummy node 是一个虚拟节点，也可以认为是标杆节点。Dummy node 就是在链表表头 head 前加一个节点指向 head，即 dummy -> head。Dummy node 的使用多针对单链表没有前向指针的问题，保证链表的 head 不会在删除操作中丢失。除此之外，还有一种用法比较少见，就是使用 dummy node 来进行head的删除操作，比如 Remove Duplicates From Sorted List II，一般的方法current = current.next 是无法删除 head 元素的，所以这个时候如果有一个dummy node在head的前面。

所以，当链表的 head 有可能变化（被修改或者被删除）时，使用 dummy node 可以很好的简化代码，最终返回 dummy.next 即新的链表。

快慢指针

快慢指针也是一个可以用于很多问题的技巧。所谓快慢指针中的快慢指的是指针向前移动的步长，每次移动的步长较大即为快，步长较小即为慢，常用的快慢指针一般是在单链表中让快指针每次向前移动2，慢指针则每次向前移动1。快慢两个指针都从链表头开始遍历，于是快指针到达链表末尾的时候慢指针刚好到达中间位置，于是可以得到中间元素的值。快慢指针在链表相关问题中主要有两个应用：

- 快速找出未知长度单链表的中间节点 设置两个指针 `*fast`、`*slow` 都指向单链表的头节点，其中 `*fast` 的移动速度是 `*slow` 的2倍，当 `*fast` 指向末尾节点的时候，`slow` 正好就在中间了。
- 判断单链表是否有环 利用快慢指针的原理，同样设置两个指针 `*fast`、`*slow` 都指向单链表的头节点，其中 `*fast` 的移动速度是 `*slow` 的2倍。如果 `*fast = NULL`，说明该单链表 以 `NULL` 结尾，不是循环链表；如果 `*fast = *slow`，则快指针追上慢指针，说明该链表是循环链表。

Binary Tree - 二叉树

二叉树是每个节点最多有两个子树的树结构，子树有左右之分，二叉树常被用于实现**二叉查找树**和**二叉堆**。

二叉树的第i层至多有 2^{i-1} 个结点；深度为k的二叉树至多有 2^k-1 个结点；对任何一棵二叉树T，如果其终端结点数为 n_0, 度为2的结点数为 n_2, 则 $n_0=n_2+1$。

一棵深度为 k, 且有 2^k-1 个节点称之为**满二叉树**；深度为 k，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中序号为 1 至 n 的节点对应时，称之为**完全二叉树**。完全二叉树中重在节点标号对应。

编程实现

Python

```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left, self.right = None, None
```

C++

```cpp
struct TreeNode {
      int val;
      TreeNode *left;
      TreeNode *right;
      TreeNode(int x) : val(x), left(NULL), right(NULL) {}
  };
```

Java

```java
public class TreeNode {
    public int val;
    public TreeNode left, right;
    public TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
```

树的遍历

从二叉树的根节点出发，节点的遍历分为三个主要步骤：对当前节点进行操作（称为“访问”节点，或者根节点）、遍历左边子节点、遍历右边子节点。访问节点顺序的不同也就形成了不同的遍历方式。需要注意的是树的遍历通常使用递归的方法进行理解和实现，在访问元素时也需要使用递归的思想去理解。实际实现中对于前序和中序遍历可尝试使用递归实现。

按照访问根元素(当前元素)的前后顺序，遍历方式可划分为如下几种：

- 深度优先：先访问子节点，再访问父节点，最后访问第二个子节点。根据根节点相对于左右子节点的访问先后顺序又可细分为以下三种方式。
  1. 前序(pre-order)：先根后左再右
  2. 中序(in-order)：先左后根再右
  3. 后序(post-order)：先左后右再根
- 广度优先：先访问根节点，沿着树的宽度遍历子节点，直到所有节点均被访问为止。

如下图所示，遍历顺序在右侧框中，红色A为根节点。使用递归和整体的思想去分析遍历顺序较为清晰。

二叉树的广度优先遍历和树的前序/中序/后序遍历不太一样，前/中/后序遍历使用递归，也就是栈的思想对二叉树进行遍历，广度优先一般使用队列的思想对二叉树进行遍历。

如果已知中序遍历和前序遍历或者后序遍历，那么就可以完全恢复出原二叉树结构。其中最为关键的是前序遍历中第一个一定是根，而后序遍历最后一个一定是根，中序遍历在得知根节点后又可进一步递归得知左右子树的根节点。但是这种方法也是有适用范围的：元素不能重复！否则无法完成定位。

树类题的复杂度分析

对树相关的题进行复杂度分析时可统计对每个节点被访问的次数，进而求得总的时间复杂度。

Binary Search Tree - 二叉查找树

一颗**二叉查找树(BST)**是一颗二叉树，其中每个节点都含有一个可进行比较的键及相应的值，且每个节点的键都**大于等于左子树中的任意节点的键**，而**小于右子树中的任意节点的键**。

使用中序遍历可得到有序数组，这是二叉查找树的又一个重要特征。

二叉查找树使用的每个节点含有**两个**链接，它是将链表插入的灵活性和有序数组查找的高效性结合起来的

高效符号表实现。

Huffman Compression - 霍夫曼压缩

主要思想：放弃文本文件的普通保存方式：不再使用7位或8位二进制数表示每一个字符，而是**用较少的比特表示出现频率最高的字符，用较多的比特表示出现频率低的字符**。

使用变长编码来表示字符串，势必会导致编解码时码字的唯一性问题，因此需要一种编解码方式唯一的前缀码，而表示前缀码的一种简单方式就是使用单词查找树，其中最优前缀码即为Huffman首创。

以符号F, O, R, G, E, T为例，其出现的频次如以下表格所示。

Symbol	F	O	R	G	E	T
Frequence	2	3	4	4	5	7
Code	000	001	100	101	01	10

则对各符号进行霍夫曼编码的动态演示如下图所示。基本步骤是将出现频率由小到大排列，组成子树后频率相加作为整体再和其他未加入二叉树中的节点频率比较。加权路径长为节点的频率乘以树的深度。

有关霍夫曼编码的具体步骤可参考 Huffman 编码压缩算法 | 酷 壳 - CoolShell.cn 和 霍夫曼编码 - 维基百科，自由的百科全书，清晰易懂。

Queue - 队列

Queue 是一个 FIFO（先进先出）的数据结构，并发中使用较多，可以安全地将对象从一个任务传给另一个任务。

编程实现

Java

Queue 在 Java 中是 Interface, 一种实现是 LinkedList, LinkedList 向上转型为 Queue, Queue 通常不能存储 `null` 元素，否则与 `poll()` 等方法的返回值混淆。

```
Queue<Integer> q = new LinkedList<Integer>();
int qLen = q.size(); // get queue length
```

Methods

0:0	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

优先考虑右侧方法，右侧元素不存在时返回 `null`. 判断非空时使用 `isEmpty()` 方法，继承自 Collection.

Priority Queue - 优先队列

应用程序常常需要处理带有优先级的业务，优先级最高的业务首先得到服务。因此优先队列这种数据结构应运而生。优先队列中的每个元素都有各自的优先级，优先级最高的元素最先得到服务；优先级相同的元素按照其在优先队列中的顺序得到服务。

优先队列可以使用数组或链表实现，从时间和空间复杂度来说，往往用二叉堆来实现。

Java

Java 中提供 `PriorityQueue` 类，该类是 Interface Queue 的另外一种实现，和 `LinkedList` 的区别主要在于排序行为而不是性能，基于 priority heap 实现，非 `synchronized` ，故多线程下应使用 `PriorityBlockingQueue` . 默认为自然序（小根堆），需要其他排序方式可自行实现 `Comparator` 接口，选用合适的构造器初始化。使用迭代器遍历时不保证有序，有序访问时需要使用 `Arrays.sort(pq.toArray())` .

不同方法的时间复杂度：

- enqueuing and dequeuing: `offer` , `poll` , `remove()` and `add` - $O(\log n)$

- Object: `remove(Object)` and `contains(Object)` - $O(n)$
- retrieval: `peek`, `element`, and `size` - $O(1)$.

Deque - 双端队列

双端队列（deque，全名double-ended queue）可以让你在任何一端添加或者移除元素，因此它是一种具有队列和栈性质的数据结构。

Java

Java 在1.6之后提供了 Deque 接口，既可使用 `ArrayDeque` （数组）来实现，也可以使用 `LinkedList` （链表）来实现。前者是一个数组外加首尾索引，后者是双向链表。

```
Deque<Integer> deque = new ArrayDeque<Integer>();
```

Methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

其中 `offerLast` 和 Queue 中的 `offer` 功能相同，都是从尾部插入。

Reference

- 優先佇列 - 维基百科，自由的百科全书
- 双端队列 - 维基百科，自由的百科全书

Heap - 堆

一般情况下，堆通常指的是**二叉堆**，**二叉堆**是一个近似**完全二叉树**的数据结构，**即披着二叉树羊皮的数组，**故使用数组来实现较为便利。子结点的键值或索引总是小于（或者大于）它的父节点，且每个节点的左右子树又是一个**二叉堆**(大根堆或者小根堆)。根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。**常被用作实现优先队列。**

特点

1. **以数组表示，但是以完全二叉树的方式理解。**
2. 唯一能够同时最优地利用空间和时间的方法——最坏情况下也能保证使用 $2N \log N$ 次比较和恒定的额外空间。
3. 在索引从0开始的数组中：
   - 父节点 `i` 的左子节点在位置 `(2*i+1)`
   - 父节点 `i` 的右子节点在位置 `(2*i+2)`
   - 子节点 `i` 的父节点在位置 `floor((i-1)/2)`

堆的基本操作

以大根堆为例，堆的常用操作如下。

1. 最大堆调整（Max_Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
2. 创建最大堆（Build_Max_Heap）：将堆所有数据重新排序
3. 堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

其中步骤1是给步骤2和3用的。

Stack - 栈

栈是一种 LIFO(Last In First Out) 的数据结构，常用方法有添加元素，取栈顶元素，弹出栈顶元素，判断栈是否为空。

编程实现

Java

```
Deque<Integer> stack = new ArrayDeque<Integer>();
s.size(); // size of stack
```

JDK doc 中建议使用 `Deque` 代替 `Stack` 实现栈，因为 `Stack` 继承自 `Vector` ，需要 `synchronized` ，性能略低。

Methods

- `boolean isEmpty()` - 判断栈是否为空，若使用 Stack 类构造则为 empty()
- `E peek()` - 取栈顶元素，不移除
- `E pop()` - 移除栈顶元素并返回该元素
- `E push(E item)` - 向栈顶添加元素

Set

Set 是一种用于保存不重复元素的数据结构。常被用作测试归属性，故其查找的性能十分重要。

编程实现

Java

Set 与 Collection 具有安全一样的接口，通常有 `HashSet`，`TreeSet` 或 `LinkedHashSet` 三种实现。 `HashSet` 基于散列函数实现，无序，查询速度最快； `TreeSet` 基于红-黑树实现，有序。

```
Set<String> hash = new HashSet<String>();
hash.add("billryan");
hash.contains("billryan");
```

在不允许重复元素时可当做哈希表来用。

Map - 哈希表

Map 是一种关联数组的数据结构，也常被称为字典或键值对。

编程实现

Java

Java 的实现中 Map 是一种将对象与对象相关联的设计。常用的实现有 `HashMap` 和 `TreeMap`，`HashMap` 被用来快速访问，而 `TreeMap` 则保证『键』始终有序。Map 可以返回键的 Set, 值的 Collection, 键值对的 Set.

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("bill", 98);
map.put("ryan", 99);
boolean exist = map.containsKey("ryan"); // check key exists in map
int point = map.get("bill"); // get value by key
int point = map.remove("bill") // remove by key, return value
Set<String> set = map.keySet();
// iterate Map
for (Entry<String, Integer> entry : map.entrySet()) {
    // do some thing
}
```

Basics Sorting - 基础排序算法

算法复习——排序

时间限制为1s时，大O为10000000时勉强可行，100,000,000时很悬。

算法分析

1. 时间复杂度-执行时间(比较和交换次数)
2. 空间复杂度-所消耗的额外内存空间
    - 使用小堆栈或表
    - 使用链表或指针、数组索引来代表数据
    - 排序数据的副本

对具有重键的数据(同一组数按不同键多次排序)进行排序时，需要考虑排序方法的稳定性，在非稳定性排序算法中需要稳定性时可考虑加入小索引。

稳定性：如果排序后文件中拥有相同键的项的相对位置不变，这种排序方式是稳定的。

常见的排序算法根据是否需要比较可以分为如下几类：

- Comparison Sorting
    1. Bubble Sort
    2. Selection Sort
    3. Insertion Sort
    4. Shell Sort
    5. Merge Sort
    6. Quck Sort
    7. Heap Sort
- Bucket Sort
- Counting Sort
- Radix Sort

从稳定性角度考虑可分为如下两类：

- 稳定
- 非稳定

Reference

- 常用排序算法总结（性能+代码） - SegmentFault
- Sorting algorithm - Wikipedia, the free encyclopedia - 各类排序算法的「平均、最好、最坏时间复杂度」总结。
- 经典排序算法总结与实现 | Jark's Blog - 基于 Python 的较为清晰的总结。

- 【面经】硅谷前沿Startup面试经验-排序算法总结及快速排序算法代码_九章算法 - 总结了一些常用常问的排序算法。

Bubble Sort - 冒泡排序

核心：**冒泡**，持续比较相邻元素，大的挪到后面，因此大的会逐步往后挪，故称之为冒泡。

6 5 3 1 8 7 2 4

Implementation

Python

```
#!/usr/bin/env python


def bubbleSort(alist):
    for i in xrange(len(alist)):
        print(alist)
        for j in xrange(1, len(alist) - i):
            if alist[j - 1] > alist[j]:
                alist[j - 1], alist[j] = alist[j], alist[j - 1]

    return alist

unsorted_list = [6, 5, 3, 1, 8, 7, 2, 4]
print(bubbleSort(unsorted_list))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        bubbleSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void bubbleSort(int[] array) {
        int len = array.length;
        for (int i = 0; i < len; i++) {
            for (int item : array) {
```

```
                System.out.print(item + " ");
            }
            System.out.println();
            for (int j = 1; j < len - i; j++) {
                if (array[j - 1] > array[j]) {
                    int temp = array[j - 1];
                    array[j - 1] = array[j];
                    array[j] = temp;
                }
            }
        }
    }
}
```

复杂度分析

平均情况与最坏情况均为 $O(n^2)$, 使用了 temp 作为临时交换变量，空间复杂度为 $O(1)$.

Reference

- 冒泡排序 - 维基百科，自由的百科全书

Selection Sort - 选择排序

核心：不断地选择剩余元素中的最小者。

1. 找到数组中最小元素并将其和数组第一个元素交换位置。
2. 在剩下的元素中找到最小元素并将其与数组第二个元素交换，直至整个数组排序。

性质：

- 比较次数=(N-1)+(N-2)+(N-3)+...+2+1~N^2/2
- 交换次数=N
- 运行时间与输入无关
- 数据移动最少

下图来源为 File:Selection-Sort-Animation.gif - IB Computer Science

Implementation

Python

```
#!/usr/bin/env python


def selectionSort(alist):
    for i in xrange(len(alist)):
        print(alist)
        min_index = i
        for j in xrange(i + 1, len(alist)):
            if alist[j] < alist[min_index]:
```

```
                min_index = j
        alist[min_index], alist[i] = alist[i], alist[min_index]
    return alist

unsorted_list = [8, 5, 2, 6, 9, 3, 1, 4, 0, 7]
print(selectionSort(unsorted_list))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{8, 5, 2, 6, 9, 3, 1, 4, 0, 7};
        selectionSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void selectionSort(int[] array) {
        int len = array.length;
        for (int i = 0; i < len; i++) {
            for (int item : array) {
                System.out.print(item + " ");
            }
            System.out.println();
            int min_index = i;
            for (int j = i + 1; j < len; j++) {
                if (array[j] < array[min_index]) {
                    min_index = j;
                }
            }
            int temp = array[min_index];
            array[min_index] = array[i];
            array[i] = temp;
        }
    }
}
```

Reference

- 选择排序 - 维基百科，自由的百科全书
- The Selection Sort — Problem Solving with Algorithms and Data Structures

Insertion Sort - 插入排序

核心：通过构建有序序列，对于未排序序列，从后向前扫描(对于单向链表则只能从前往后遍历)，找到相应位置并插入。实现上通常使用in-place排序(需用到O(1)的额外空间)

1. 从第一个元素开始，该元素可认为已排序
2. 取下一个元素，对已排序数组从后往前扫描
3. 若从排序数组中取出的元素大于新元素，则移至下一位置
4. 重复步骤3，直至找到已排序元素小于或等于新元素的位置
5. 插入新元素至该位置
6. 重复2~5

性质：

- 交换操作和数组中导致的数量相同
- 比较次数>=倒置数量，<=倒置的数量加上数组的大小减一
- 每次交换都改变了两个顺序颠倒的元素的位置，即减少了一对倒置，倒置数量为0时即完成排序。
- 每次交换对应着一次比较，且1到N-1之间的每个i都可能需要一次额外的记录(a[i]未到达数组左端时)
- 最坏情况下需要~N^2/2次比较和~N^2/2次交换，最好情况下需要N-1次比较和0次交换。
- 平均情况下需要~N^2/4次比较和~N^2/4次交换

6 5 3 1 8 7 2 4

Implementation

Python

```
#!/usr/bin/env python


def insertionSort(alist):
    for i, item_i in enumerate(alist):
        print alist
```

```
        index = i
        while index > 0 and alist[index - 1] > item_i:
            alist[index] = alist[index - 1]
            index -= 1

        alist[index] = item_i

    return alist

unsorted_list = [6, 5, 3, 1, 8, 7, 2, 4]
print(insertionSort(unsorted_list))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        insertionSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void insertionSort(int[] array) {
        int len = array.length;
        for (int i = 0; i < len; i++) {
            int index = i, array_i = array[i];
            while (index > 0 && array[index - 1] > array_i) {
                array[index] = array[index - 1];
                index -= 1;
            }
            array[index] = array_i;

            /* print sort process */
            for (int item : array) {
                System.out.print(item + " ");
            }
            System.out.println();
        }
    }
}
```

实现(C++)：

```
template<typename T>
void insertion_sort(T arr[], int len) {
    int i, j;
    T temp;
    for (int i = 1; i < len; i++) {
        temp = arr[i];
        for (int j = i - 1; j >= 0 && arr[j] > temp; j--) {
            a[j + 1] = a[j];
        }
        arr[j + 1] = temp;
```

```
    }
}
```

希尔排序

核心：基于插入排序，使数组中任意间隔为h的元素都是有序的，即将全部元素分为h个区域使用插入排序。其实现可类似于插入排序但使用不同增量。更高效的原因是它权衡了子数组的规模和有序性。

实现(C++):

```
template<typename T>
void shell_sort(T arr[], int len) {
    int gap, i, j;
    T temp;
    for (gap = len >> 1; gap > 0; gap >>= 1)
        for (i = gap; i < len; i++) {
            temp = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap)
                arr[j + gap] = arr[j];
            arr[j + gap] = temp;
        }
}
```

Reference

- 插入排序 - 维基百科，自由的百科全书
- 希尔排序 - 维基百科，自由的百科全书
- The Insertion Sort — Problem Solving with Algorithms and Data Structures

Merge Sort - 归并排序

核心：将两个有序对数组归并成一个更大的有序数组。通常做法为递归排序，并将两个不同的有序数组归并到第三个数组中。

先来看看动图，归并排序是一种典型的分治应用。

6 5 3 1 8 7 2 4

Python

```
#!/usr/bin/env python


class Sort:
    def mergeSort(self, alist):
        if len(alist) <= 1:
            return alist

        mid = len(alist) / 2
        left = self.mergeSort(alist[:mid])
        print("left = " + str(left))
        right = self.mergeSort(alist[mid:])
        print("right = " + str(right))
        return self.mergeSortedArray(left, right)

    #@param A and B: sorted integer array A and B.
    #@return: A new sorted integer array
    def mergeSortedArray(self, A, B):
        sortedArray = []
        l = 0
        r = 0
        while l < len(A) and r < len(B):
            if A[l] < B[r]:
                sortedArray.append(A[l])
                l += 1
            else:
                sortedArray.append(B[r])
                r += 1
        sortedArray += A[l:]
        sortedArray += B[r:]

        return sortedArray

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
```

```
merge_sort = Sort()
print(merge_sort.mergeSort(unsortedArray))
```

原地归并

Java

```
public class MergeSort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        mergeSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    private static void merge(int[] array, int low, int mid, int high) {
        int[] helper = new int[array.length];
        // copy array to helper
        for (int k = low; k <= high; k++) {
            helper[k] = array[k];
        }
        // merge array[low...mid] and array[mid + 1...high]
        int i = low, j = mid + 1;
        for (int k = low; k <= high; k++) {
            // k means current location
            if (i > mid) {
            // no item in left part
                array[k] = helper[j];
                j++;
            } else if (j > high) {
            // no item in right part
                array[k] = helper[i];
                i++;
            } else if (helper[i] > helper[j]) {
            // get smaller item in the right side
                array[k] = helper[j];
                j++;
            } else {
            // get smaller item in the left side
                array[k] = helper[i];
                i++;
            }
        }
    }

    public static void sort(int[] array, int low, int high) {
        if (high <= low) return;
        int mid = low + (high - low) / 2;
        sort(array, low, mid);
        sort(array, mid + 1, high);
        merge(array, low, mid, high);
        for (int item : array) {
            System.out.print(item + " ");
```

```
        }
        System.out.println();
    }

    public static void mergeSort(int[] array) {
        sort(array, 0, array.length - 1);
    }
}
```

时间复杂度为 $O(N \log N)$, 使用了等长的辅助数组，空间复杂度为 $O(N)$。

Reference

- Mergesort - Robert Sedgewick 的大作，非常清晰。

Quick Sort - 快速排序

核心：快排是一种采用分治思想的排序算法，大致分为三个步骤。

1. 定基准——首先随机选择一个元素最为基准
2. 划分区——所有比基准小的元素置于基准左侧，比基准大的元素置于右侧
3. 递归调用——递归地调用此切分过程

out-in-place - 非原地快排

容易实现和理解的一个方法是采用递归，使用 Python 的 list comprehension 实现如下所示：

```
#!/usr/bin/env python


def qsort1(alist):
    print(alist)
    if len(alist) <= 1:
        return alist
    else:
        pivot = alist[0]
        return qsort1([x for x in alist[1:] if x < pivot]) + \
               [pivot] + \
               qsort1([x for x in alist[1:] if x >= pivot])

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
print(qsort1(unsortedArray))
```

输出如下所示：

```
[6, 5, 3, 1, 8, 7, 2, 4]
[5, 3, 1, 2, 4]
[3, 1, 2, 4]
[1, 2]
[]
[2]
[4]
[]
[8, 7]
[7]
[]
[1, 2, 3, 4, 5, 6, 7, 8]
```

『递归 + 非原地排序』的实现虽然简单易懂，但是如此一来『快速排序』便不再是最快的通用排序算法了，因为递归调用过程中非原地排序需要生成新数组，空间复杂度颇高。list comprehension 大法虽然好写，但是用在『快速排序』算法上就不是那么可取了。

复杂度分析

在最好情况下，快速排序的基准元素正好是整个数组的中位数，可以近似为二分，那么最好情况下递归的层数为 $\log n$, 咋看一下每一层的元素个数都是 n, 那么空间复杂度为 $O(n)$ 无疑了，不过这只答对了一半，从结论上来看是对的，但分析方法是错的。

首先来看看什么叫空间复杂度——简单来讲可以认为是程序在运行过程中所占用的存储空间大小。那么对于递归的 out-in-place 调用而言，排除函数调用等栈空间，**最好情况下，每往下递归调用一层，所需要的存储空间是上一层中的一半。完成最底层的调用后即向上返回执行出栈操作，故并不需要保存每层所有元素的值。**所以需要的总的存储空间就是 $\sum_{i=0} \frac{n}{2^i} = 2n$

不是特别理解的可以结合下图的非严格分析和上面 Python 的代码，递归调用的第一层保存8个元素的值，那么第二层调用时实际需要保存的其实仅为4个元素，逐层往下递归，而不是自左向右保存每一层的所有元素。

那么在最坏情况下 out-in-place 需要耗费多少额外空间呢？最坏情况下第 i 层需要 $i-1$ 次交换，故总的空间复杂度：

$\sum_{i=0}^{n}(n-i+1) = O(n^2)$

in-place - 原地快排

one index for partition

先来看一种简单的 in-place 实现，仍然以 `[6, 5, 3, 1, 8, 7, 2, 4]` 为例，结合下图进行分析。以下标 l 和 u 表示数组待排序部分的下界(lower bound)和上界(upper bound)，下标 m 表示遍历到数组第 i 个元素时当前 partition 的索引，基准元素为 t, 即图中的 target.

在遍历到第 i 个元素时，$x[i]$ 有两种可能，第一种是 $x[i] \geq t$, i 自增往后遍历；第二种是 $x[i] < t$, 此时需要将 $x[i]$ 置于前半部分，比较简单的实现为 `swap(x[++m], x[i])`. 直至 `i == u` 时划分阶段结束，分两截递归进行快排。既然说到递归，就不得不提递归的终止条件，容易想到递归的终止步为 `l >= u`, 即索引相等或者交叉时退出。使用 Python 的实现如下所示：

Python

```
#!/usr/bin/env python


def qsort2(alist, l, u):
    print(alist)
    if l >= u:
        return

    m = l
    for i in xrange(l + 1, u + 1):
        if alist[i] < alist[l]:
            m += 1
            alist[m], alist[i] = alist[i], alist[m]
    # swap between m and l after partition, important!
    alist[m], alist[l] = alist[l], alist[m]
    qsort2(alist, l, m - 1)
    qsort2(alist, m + 1, u)
```

```
unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
print(qsort2(unsortedArray, 0, len(unsortedArray) - 1))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        quickSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void quickSort1(int[] array, int l, int u) {
        for (int item : array) {
            System.out.print(item + " ");
        }
        System.out.println();

        if (l >= u) return;
        int m = l;
        for (int i = l + 1; i <= u; i++) {
            if (array[i] < array[l]) {
                m += 1;
                int temp = array[m];
                array[m] = array[i];
                array[i] = temp;
            }
        }
        // swap between array[m] and array[l]
        // put pivot in the mid
        int temp = array[m];
        array[m] = array[l];
        array[l] = temp;

        quickSort1(array, l, m - 1);
        quickSort1(array, m + 1, u);
    }

    public static void quickSort(int[] array) {
        quickSort1(array, 0, array.length - 1);
    }
}
```

容易出错的地方在于当前 partition 结束时未将 i 和 m 交换。比较 `alist[i]` 和 `alist[l]` 时只能使用 `<` 而不是 `<=`！因为只有取 `<` 才能进入收敛条件，`<=` 则可能会出现死循环，因为在 `=` 时第一个元素可能保持不变进而产生死循环。

相应的结果输出为：

```
[6, 5, 3, 1, 8, 7, 2, 4]
[4, 5, 3, 1, 2, 6, 8, 7]
```

```
[2, 3, 1, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Two-way partitioning

对于仅使用一个索引进行 partition 操作的快排对于随机分布数列的效果还是不错的，但若数组本身就已经有序或者相等的情况下，每次划分仅能确定一个元素的最终位置，故最坏情况下的时间复杂度变为 $O(n^2)$. 那么有什么办法尽可能避免这种最坏情况吗？聪明的人类总是能找到更好地解决办法——使用两个索引分别向右向左进行 partition.

先来一张动图看看使用两个索引进行 partition 的过程。

1. 选中 `3` 作为基准
2. `lo` 指针指向元素 `6`，`hi` 指针指向 `4`，移动 `lo` 直至其指向的元素大于等于 `3`，移动 `hi` 直至其指向的元素小于 `3`。找到后交换 `lo` 和 `hi` 指向的元素——交换元素 `6` 和 `2`。
3. `lo` 递增， `hi` 递减，重复步骤2，此时 `lo` 指向元素为 `5`，`hi` 指向元素为 `1`. 交换元素。
4. `lo` 递增， `hi` 递减，发现其指向元素相同，此轮划分结束。递归排序元素 `3` 左右两边的元素。

对上述过程进行适当的抽象：

1. 下标 i 和 j 初始化为待排序数组的两端。
2. 基准元素设置为数组的第一个元素。
3. 执行 partition 操作，大循环内包含两个内循环：
   - 左侧内循环自增 i, 直到遇到**不小于**基准元素的值为止。
   - 右侧内循环自减 j, 直到遇到**不大于**基准元素的值为止。
4. 大循环测试两个下标是否相等或交叉，交换其值。

这样一来对于数组元素均相等的情形下，每次 partition 恰好在中间元素，故共递归调用 $\log n$ 次，每层递归调用进行 partition 操作的比较次数总和近似为 n. 故总计需 $n \log n$ 次比较。[programming_pearls]

Python

```
#!/usr/bin/env python
```

```
def qsort3(alist, l, u):
    print(alist)
    if l >= u:
        return

    t = alist[l]
    i = l + 1
    j = u
    while True:
        while i <= u and alist[i] < t:
            i += 1
        while alist[j] > t:
            j -= 1
        if i > j:
            break
        # swap after make sure i > j
        alist[i], alist[j] = alist[j], alist[i]
    # do not forget swap l and j
    alist[l], alist[j] = alist[j], alist[l]

    qsort3(alist, l, j - 1)
    qsort3(alist, j + 1, u)

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
print(qsort3(unsortedArray, 0, len(unsortedArray) - 1))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        quickSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void quickSort2(int[] array, int l, int u) {
        for (int item : array) {
            System.out.print(item + " ");
        }
        System.out.println();

        if (l >= u) return;
        int pivot = array[l];
        int left = l + 1;
        int right = u;
        while (left <= right) {
            while (left <= right && array[left] < pivot) {
                left++;
            }
            while (array[right] > pivot) {
                right--;
            }
```

```
            if (left > right) break;
            // swap array[left] with array[right] while left <= right
            int temp = array[left];
            array[left] = array[right];
            array[right] = temp;
        }
        /* swap the smaller with pivot */
        int temp = array[right];
        array[right] = array[l];
        array[l] = temp;

        quickSort2(array, l, right - 1);
        quickSort2(array, right + 1, u);
    }

    public static void quickSort(int[] array) {
        quickSort2(array, 0, array.length - 1);
    }
}
```

相应的输出为：

```
[6, 5, 3, 1, 8, 7, 2, 4]
[2, 5, 3, 1, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

从以上3种快排的实现我们可以发现其与『归并排序』的区别主要有如下两点：

1. 归并排序将数组分成两个子数组分别排序，并将有序的子数组归并以将整个数组排序。递归调用发生在处理整个数组之前。
2. 快速排序将一个数组分成两个子数组并对这两个子数组独立地排序，两个子数组有序时整个数组也就有序了。递归调用发生在处理整个数组之后。

Robert Sedgewick 在其网站上对 Quicksort 做了较为完整的介绍，建议去围观下。

Reference

- 快速排序 - 维基百科，自由的百科全书
- Quicksort | Robert Sedgewick
- Programming Pearls Column 11 Sorting - 深入探讨了插入排序和快速排序
- Quicksort Analysis
- > [programming_pearls]. Programming Pearls(第二版修订版) 一书中第11章排序中注明需要 $n \log 2n$ 次比较，翻译有误？ ↩

Heap Sort - 堆排序

堆排序通常基于**二叉堆** 实现，以大根堆为例，堆排序的实现过程分为两个子过程。第一步为取出大根堆的根节点(当前堆的最大值), 由于取走了一个节点，故需要对余下的元素重新建堆。重新建堆后继续取根节点，循环直至取完所有节点，此时数组已经有序。基本思想就是这样，不过实现上还是有些小技巧的。

堆的操作

以大根堆为例，堆的常用操作如下。

1. 最大堆调整（Max_Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
2. 创建最大堆（Build_Max_Heap）：将堆所有数据重新排序
3. 堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

其中步骤1是给步骤2和3用的。

6 5 3 1 8 7 2 4

建堆时可以自顶向下，也可以采取自底向上，以下先采用自底向上的思路分析。我们可以将数组的后半部分节点想象为堆的最下面的那些节点，由于是单个节点，故显然满足二叉堆的定义，于是乎我们就可以从中间节点向上逐步构建二叉堆，每前进一步都保证其后的节点都是二叉堆，这样一来前进到第一个节点时整个数组就是一个二叉堆了。下面用 C++ 实现一个堆的类。

堆排在空间比较小(嵌入式设备和手机)时特别有用，但是因为现代系统往往有较多的缓存，堆排序无法有效利用缓存，数组元素很少和相邻的其他元素比较，故缓存未命中的概率远大于其他在相邻元素间比较的算法。但是在海量数据的排序下又重新发挥了重要作用，因为它在插入操作和删除最大元素的混合动态场景中能保证对数级别的运行时间。TopM

C++

```
#include <iostream>
#include <vector>

using namespace std;
```

```
class HeapSort {
    // get the parent node index
    int parent(int i) {
        return (i - 1) / 2;
    }

    // get the left child node index
    int left(int i) {
        return 2 * i + 1;
    }

    // get the right child node index
    int right(int i) {
        return 2 * i + 2;
    }

    // build max heap
    void build_max_heapify(vector<int> &nums, int heap_size) {
        for (int i = heap_size / 2; i >= 0; --i) {
            max_heapify(nums, i, heap_size);
        }
        print_heap(nums, heap_size);
    }

    // build min heap
    void build_min_heapify(vector<int> &nums, int heap_size) {
        for (int i = heap_size / 2; i >= 0; --i) {
            min_heapify(nums, i, heap_size);
        }
        print_heap(nums, heap_size);
    }

    // adjust the heap to max-heap
    void max_heapify(vector<int> &nums, int k, int len) {
        // int len = nums.size();
        while (k < len) {
            int max_index = k;
            // left leaf node search
            int l = left(k);
            if (l < len && nums[l] > nums[max_index]) {
                max_index = l;
            }
            // right leaf node search
            int r = right(k);
            if (r < len && nums[r] > nums[max_index]) {
                max_index = r;
            }
            // node after k are max-heap already
            if (k == max_index) {
                break;
            }
            // keep the root node the largest
            int temp = nums[k];
            nums[k] = nums[max_index];
            nums[max_index] = temp;
            // adjust not only just current index
            k = max_index;
        }
```

```
    }

    // adjust the heap to min-heap
    void min_heapify(vector<int> &nums, int k, int len) {
        // int len = nums.size();
        while (k < len) {
            int min_index = k;
            // left leaf node search
            int l = left(k);
            if (l < len && nums[l] < nums[min_index]) {
                min_index = l;
            }
            // right leaf node search
            int r = right(k);
            if (r < len && nums[r] < nums[min_index]) {
                min_index = r;
            }
            // node after k are min-heap already
            if (k == min_index) {
                break;
            }
            // keep the root node the largest
            int temp = nums[k];
            nums[k] = nums[min_index];
            nums[min_index] = temp;
            // adjust not only just current index
            k = min_index;
        }
    }

public:
    // heap sort
    void heap_sort(vector<int> &nums) {
        int len = nums.size();
        // init heap structure
        build_max_heapify(nums, len);
        // heap sort
        for (int i = len - 1; i >= 0; --i) {
            // put the largest number int the last
            int temp = nums[0];
            nums[0] = nums[i];
            nums[i] = temp;
            // reconstruct heap
            build_max_heapify(nums, i);
        }
        print_heap(nums, len);
    }

    // print heap between [0, heap_size - 1]
    void print_heap(vector<int> &nums, int heap_size) {
        for (int i = 0; i < heap_size; ++i) {
            cout << nums[i] << ", ";
        }
        cout << endl;
    }
};

int main(int argc, char *argv[])
{
```

```
    int A[] = {19, 1, 10, 14, 16, 4, 7, 9, 3, 2, 8, 5, 11};
    vector<int> nums;
    for (int i = 0; i < sizeof(A) / sizeof(A[0]); ++i) {
        nums.push_back(A[i]);
    }

    HeapSort sort;
    sort.print_heap(nums, nums.size());
    sort.heap_sort(nums);

    return 0;
}
```

复杂度分析

从代码中可以发现堆排最费时间的地方在于构建二叉堆的过程。

上述构建大根堆和小根堆都是自底向上的方法，建堆过程时间复杂度为 $O(2N)$, 堆排过程中重建的时间复杂度为 $O(2N\log N)$. 故总的时间复杂度为 $O(N\log N)$.

先看看建堆的过程，画图分析(比如以8个节点为例)可知在最坏情况下，每次都需要调整之前已经成为堆的节点，那么就意味着有二分之一的节点向下比较了一次，四分之一的节点向下比较了两次，八分之一的节点比较了三次... 等差等比数列求和，具体过程可参考下面的链接。

Reference

- 堆排序 - 维基百科，自由的百科全书
- Priority Queues - Robert Sedgewick 的大作，详解了关于堆的操作。
- 经典排序算法总结与实现 | Jark's Blog - 堆排序讲的很好。
- *Algorithm* - Robert Sedgewick
- 堆排序中建堆过程时间复杂度O(n)怎么来的?
- 《大话数据结构》第9章 排序 9.7 堆排序（上） - 伍迷 - 博客园
- 《大话数据结构》第9章 排序 9.7 堆排序（下） - 伍迷 - 博客园

Bucket Sort

桶排序和归并排序有那么点点类似，也使用了归并的思想。大致步骤如下：

1. 设置一个定量的数组当作空桶。
2. Divide - 从待排序数组中取出元素，将元素按照一定的规则塞进对应的桶子去。
3. 对每个非空桶进行排序，通常可在塞元素入桶时进行插入排序。
4. Conquer - 从非空桶把元素再放回原来的数组中。

Reference

- Bucket Sort Visualization - 动态演示。
- 桶排序 - 维基百科，自由的百科全书

Counting Sort

计数排序，顾名思义，就是对待排序数组按元素进行计数。使用前提是需要先知道待排序数组的元素范围，将这些一定范围的元素置于新数组中，新数组的大小为待排序数组中最大元素与最小元素的差值。

维基上总结的四个步骤如下：

1. 定新数组大小——找出待排序的数组中最大和最小的元素
2. 统计次数——统计数组中每个值为i的元素出现的次数，存入新数组C的第i项
3. 对统计次数逐个累加——对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）
4. 反向填充目标数组——将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1

其中反向填充主要是为了避免重复元素落入新数组的同一索引处。

Reference

- 计数排序 - 维基百科，自由的百科全书 - 中文版的维基感觉比英文版的好理解些。
- Counting Sort Visualization - 动画真心不错~ 结合着看一遍就理解了。

Radix Sort

Basics Algorithm

本章主要介绍一些常用的基本算法，后序章节介绍一些高级算法。

Divide and Conquer - 分治法

在计算机科学中，分治法是一种很重要的算法。分治法即『分而治之』，把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个思想是很多高效算法的基础，如排序算法（快速排序，归并排序）等。

分治法思想

分治法所能解决的问题一般具有以下几个特征：

1. 问题的规模缩小到一定的程度就可以容易地解决。
2. 问题可以分解为若干个规模较小的相同问题，即该问题具有**最优子结构**性质。
3. 利用该问题分解出的子问题的解可以合并为该问题的解。
4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

分治法的三个步骤是：

1. 分解（Divide）：将原问题分解为若干子问题，这些子问题都是原问题规模较小的实例。
2. 解决（Conquer）：递归地求解各子问题。如果子问题规模足够小，则直接求解。
3. 合并（Combine）：将所有子问题的解合并为原问题的解。

分治法的经典题目：

1. 二分搜索
2. 大整数乘法
3. Strassen矩阵乘法
4. 棋盘覆盖
5. 归并排序
6. 快速排序
7. 循环赛日程表
8. 汉诺塔

Basics Miscellaneous

杂项部分，涉及「位操作」等。

Bit Manipulation

位操作有按位与、或、非、左移n位和右移n位等操作。

XOR - 异或

> 异或：相同为0，不同为1。也可用「不进位加法」来理解。

异或操作的一些特点：

```
x ^ 0 = x
x ^ 1s = ~x // 1s = ~0
x ^ (~x) = 1s
x ^ x = 0 // interesting and important!
a ^ b = c => a ^ c = b, b ^ c = a // swap
a ^ b ^ c = a ^ (b ^ c) = (a ^ b) ^ c // associative
```

移位操作

移位操作可近似为乘以/除以2的幂。 `0b0010 * 0b0110` 等价于 `0b0110 << 2`. 下面是一些常见的移位组合操作。

1. 将 `x` 最右边的 `n` 位清零 - `x & (~0 << n)`
2. 获取 `x` 的第 `n` 位值(0或者1) - `x & (1 << n)`
3. 获取 `x` 的第 `n` 位的幂值 - `(x >> n) & 1`
4. 仅将第 `n` 位置为 `1` - `x | (1 << n)`
5. 仅将第 `n` 位置为 `0` - `x & (~(1 << n))`
6. 将 `x` 最高位至第 `n` 位(含)清零 - `x & ((1 << n) - 1)`
7. 将第 `n` 位至第0位(含)清零 - `x & (~((1 << (n + 1)) - 1))`
8. 仅更新第 `n` 位，写入值为 `v`；`v` 为1则更新为1，否则为0 - `mask = ~(1 << n); x = (x & mask) | (v << i)`

实际应用

位图(Bitmap)

位图一般用于替代flag array，节约空间。
一个int型的数组用位图替换后，占用的空间可以缩小到原来的$1/32$.
下面代码定义了一个100万大小的类图，setbit和testbit函数

```
#define N 1000000 // 1 million
#define WORD_LENGTH sizeof(int) * 8 //sizeof返回字节数，乘以8，为int类型总位数

//bits为数组，i控制具体哪位，即i为0~1000000
void setbit(unsigned int* bits, unsigned int i){
  bits[i / WORD_LENGTH] |= 1<<(i % WORD_LENGTH);
```

```
}

int testbit(unsigned int* bits, unsigned int i){
  return bits[i/WORD_LENGTH] & (1<<(i % WORD_LENGTH));
}

unsigned int bits[N/WORD_LENGTH + 1];
```

Reference

- 位运算应用技巧（1） » NoAlGo博客
- 位运算应用技巧（2） » NoAlGo博客
- 位运算简介及实用技巧（一）：基础篇 | Matrix67: The Aha Moments
- *cc150* chapter 8.5 and chapter 9.5
- 《编程珠玑2》
- 《Elementary Algorithms》 Larry LIU Xinyu

Knapsack - 背包问题

在一次抢珠宝店的过程中，抢劫犯只能抢走以下三种珠宝，其重量和价值如下表所述。

Item(jewellery)	Weight	Value
1	6	23
2	3	13
3	4	11

抢劫犯这次过来光顾珠宝店只带了一个最多只能承重17 kg的粉红色小包，于是问题来了，怎样搭配这些不同重量不同价值的珠宝才能不虚此行呢？哎，这年头抢劫也不容易啊...

用数学语言来描述这个问题就是： 背包最多只能承重 W kg, 有 n 种珠宝可供选择，这 n 种珠宝的重量分别为 $\omega_1, \cdots, \omega_n$, 相应的价值为 $v_1, \cdots, v_n$. 问如何选择这些珠宝使得放进包里的珠宝价值最大化?

Knapsack with repetition - 物品重复可用的背包问题

由于这类背包问题中，同一物品可以被多次选择，因此称为Knapsack with repetition, 又称Unbounded knapsack problem(无界背包问题).

动态规划是解决背包问题的有力武器，而在动态规划中，主要的问题之一就是——状态(子问题)是什么？在本题中我们可以从两个方面对原始问题进行化大为小：要么是是更小的背包容量 $\omega \leq W$, 要么尝试更少的珠宝数目(如珠宝 $1, 2, \cdots, j,\ for\ j \leq n$). 这两个状态(子问题)究竟哪个对于解题更为方便，还需进一步论证——能否根据状态(子问题)很方便地写出状态转移方程。

先来看看第一种状态：**在背包容量为 ω 时抢劫犯所能获得的最优值为 $K(\omega)$.** 对应此状态的状态转移方程并不是那么直观，先从 $K(\omega)$ 所包含的信息出发，$K(\omega) > 0$ 时，背包中必然含有某件值钱的珠宝，不妨假设最优值 $K(\omega)$ 包含某珠宝 i, 那么将珠宝 i 从背包中移除后，背包中剩余珠宝的价值加上珠宝 i 的价值即为 $K(\omega)$. 哪尼？这不就是个天然的状态转移方程么？抢劫犯灵机一动，立马想出了如下状态转移方程：

$K(\omega) = F(\omega - \omega_i) + v_i\ (\omega_i \in \Omega)$

其中 $F(\omega - \omega_i)$ 为拿出珠宝 i 后的价值映射函数(用人话来说就是把粉红色小包里剩下的珠宝价值加起来)，取出来的珠宝重量 $\omega_i < \omega$(总不能取出大于背包重量的珠宝吧...), Ω 即为 $K(\omega)$ 中 ω_i 的所有可能取值。想了想好像哪里不对劲，$K(\omega)$ 的转移关系没鼓捣出来，反而新添了个 $F(\omega - \omega_i)$, 真是旧爱未了又添新欢... 别急，再仔细瞅瞅以上等式两端，拿出珠宝 i 后，其价值 v_i 就可以认为是一个定值了，故要想 $K(\omega)$ 为最大值，$F(\omega - \omega_i)$ 也理应是背包容量为 $\omega - \omega_i$ 时的包内珠宝的最大价值，如若不是，则必然存在 $F(\omega - \omega_i) < K(\omega - \omega_i)$, 即有 $K(\omega) = F(\omega - \omega_i) + v_i < K(\omega - \omega_i) + v_i = K'(\omega)$ 与 $K(\omega)$ 为在背包容量为 ω 时的最大值的定义不符，故假设不成立，$F(\omega - \omega_i) = K(\omega - \omega_i)$. 千斤顶终于成功上位——变成了备胎... 新的状态转移方程可改写为： $K(\omega) = K(\omega - \omega_i) + v_i$

嗯，好像还是有哪里不对劲，千斤顶虽然已晋级为备胎，可备胎这个身份实在是不怎么好听，这不还有下标 i 这个标记嘛，我们给抢劫犯想想法子，怎么才能让备胎尽快转正呢？！ 仔细分析发现我们刚才取出d的

价值 v_i 是从已知背包容量为 ω 时取出来的珠宝 i, 重量为 ω_i. 那么到底那几个珠宝才是可能被取出来的呢？答案不得而知，只知道肯定是小于背包容量 ω 中的某一个。既然是这样，我们把所有小于背包容量 ω 的珠宝挨个拿出来比一比不就完了么？但这样一来又有了新的问题：取出来的珠宝 ω_i 不一定是最大值 $K(\omega)$中所包含的珠宝，那假如我们一定要拿出来比一比呢？得到的结果自然是不大于最大值 $K(\omega)$(如果不是，反证法证之), 用数学语言表示就是： $K(\omega) \geq K(\omega - \omega_j) + v_j \ (\omega_j \notin \Omega)$

整理一下思路，用优雅的数学语言来表示就是： $K(\omega) = \max_{i:\ \omega_i \leq \omega}\{K(\omega - \omega_i) + v_i\}$

备胎终于得以登堂入室，警察叔叔，就是她了... 状态转移方程终于完整的找到了，千斤顶窃喜道：皇天不负有心人，我也有转正的一天，蛤蛤蛤...

Knapsack without repetition - 01背包问题

上节讲述的是最原始的背包问题，这节我们探讨条件受限情况下的背包问题。若一件珠宝最多只能带走一件，请问现在抢劫犯该如何做才能使得背包中的珠宝价值总价最大?

显然，无界背包中的状态及状态方程已经不适用于01背包问题，那么我们来比较这两个问题的不同之处，无界背包问题中同一物品可以使用多次，而01背包问题中一个背包仅可使用一次，区别就在这里。我们将 $K(\omega)$ 改为 $K(i, \omega)$ 即可，**新的状态表示前 i 件物品放入一个容量为 ω 的背包可以获得的最大价值。**

现在从以上状态定义出发寻找相应的状态转移方程。$K(i-1, \omega)$为 $K(i, \omega)$ 的子问题，如果不放第 i 件物品，那么问题即转化为「前 $i-1$ 件物品放入容量为 ω 的背包」，此时背包内获得的总价值为 $K(i-1, \omega)$；如果放入第 i 件物品，那么问题即转化为「前 $i-1$ 件物品放入容量为 $\omega - \omega_i$ 的背包」，此时背包内获得的总价值为 $K(i-1, \omega - \omega_i) + v_i$. 新的状态转移方程用数学语言来表述即为：$K(i, \omega) = \max\{K(i-1, \omega), K(i-1, \omega - \omega_i) + v_i\}$

Reference

- Chapter 6.4 Knapsack *Algorithm - S. Dasgupta*
- 0019算法笔记——【动态规划】0-1背包问题 - liufeng_king的专栏
- 崔添翼 § 翼若垂天之云 › 《背包问题九讲》2.0 alpha1

Part II - Coding

本节主要总结一些leetcode等题目的实战经验。

主要有以下章节构成。

String - 字符串

本章主要介绍字符串相关题目。

> 处理字符串操作相关问题时，常见的做法是从字符串尾部开始编辑，从后往前逆向操作。这么做的原因是因为字符串的尾部往往有足够空间，可以直接修改而不用担心覆盖字符串前面的数据。

摘自《程序员面试金典》

strStr

Source

- leetcode: Implement strStr() | LeetCode OJ
- lintcode: lintcode - (13) strstr

```
strstr (a.k.a find sub string), is a useful function in string operation.
You task is to implement this function.

For a given source string and a target string,
you should output the "first" index(from 0) of target string in source string.

If target is not exist in source, just return -1.

Example
If source="source" and target="target", return -1.

If source="abcdabcdefg" and target="bcd", return 1.

Challenge
O(n) time.

Clarification
Do I need to implement KMP Algorithm in an interview?

    - Not necessary. When this problem occurs in an interview,
    the interviewer just want to test your basic implementation ability.
```

题解

对于字符串查找问题，可使用双重for循环解决，效率更高的则为KMP算法。

Java

```java
/**
 * http://www.jiuzhang.com//solutions/implement-strstr
 */
class Solution {
    /**
     * Returns a index to the first occurrence of target in source,
     * or -1  if target is not part of source.
     * @param source string to be scanned.
     * @param target string containing the sequence of characters to match.
     */
    public int strStr(String source, String target) {
        if (source == null || target == null) {
            return -1;
        }
```

```
        int i, j;
        for (i = 0; i < source.length() - target.length() + 1; i++) {
            for (j = 0; j < target.length(); j++) {
                if (source.charAt(i + j) != target.charAt(j)) {
                    break;
                } //if
            } //for j
            if (j == target.length()) {
                return i;
            }
        } //for i

        // did not find the target
        return -1;
    }
}
```

源码分析

1. 边界检查： `source` 和 `target` 有可能是空串。
2. 边界检查之下标溢出：注意变量 `i` 的循环判断条件，如果是单纯的 `i < source.length()` 则在后面的 `source.charAt(i + j)` 时有可能溢出。
3. 代码风格：（1）运算符 `==` 两边应加空格；（2）变量名不要起 `s1``s2` 这类，要有意义，如 `target``source` ；（3）即使if语句中只有一句话也要加大括号，即 `{return -1;}` ；（4）Java 代码的大括号一般在同一行右边，C++ 代码的大括号一般另起一行；（5） `int i, j;` 声明前有一行空格，是好的代码风格。
4. 不要在for的条件中声明 `i` , `j` ，容易在循环外再使用时造成编译错误，错误代码示例：

Another Similar Question

```
/**
 * http://www.jiuzhang.com//solutions/implement-strstr
 */
public class Solution {
    public String strStr(String haystack, String needle) {
        if(haystack == null || needle == null) {
            return null;
        }
        int i, j;
        for(i = 0; i < haystack.length() - needle.length() + 1; i++) {
            for(j = 0; j < needle.length(); j++) {
                if(haystack.charAt(i + j) != needle.charAt(j)) {
                    break;
                }
            }
            if(j == needle.length()) {
                return haystack.substring(i);
            }
        }
        return null;
    }
}
```

Two Strings Are Anagrams

Source

- CC150: (158) Two Strings Are Anagrams

```
Write a method anagram(s,t) to decide if two strings are anagrams or not.

Example
Given s="abcd", t="dcab", return true.

Challenge
O(n) time, O(1) extra space
```

题解1 - hashmap 统计字频

判断两个字符串是否互为变位词，若区分大小写，考虑空白字符时，直接来理解可以认为两个字符串的拥有各不同字符的数量相同。对于比较字符数量的问题常用的方法为遍历两个字符串，统计其中各字符出现的频次，若不等则返回 `false` . 有很多简单字符串类面试题都是此题的变形题。

C++

```cpp
class Solution {
public:
    /**
     * @param s: The first string
     * @param b: The second string
     * @return true or false
     */
    bool anagram(string s, string t) {
        if (s.empty() || t.empty()) {
            return false;
        }
        if (s.size() != t.size()) {
            return false;
        }

        int letterCount[256] = {0};

        for (int i = 0; i != s.size(); ++i) {
            ++letterCount[s[i]];
            --letterCount[t[i]];
        }
        for (int i = 0; i != t.size(); ++i) {
            if (letterCount[t[i]] != 0) {
                return false;
            }
        }
```

```
        return true;
    }
};
```

源码分析

1. 两个字符串长度不等时必不可能为变位词(需要注意题目条件灵活处理)。
2. 初始化含有256个字符的计数器数组。
3. 对字符串 s 自增，字符串 t 递减，再次遍历判断 `letterCount` 数组的值，小于0时返回 `false` .

在字符串长度较长(大于所有可能的字符数)时，还可对第二个 `for` 循环做进一步优化，即 `t.size() > 256` 时，使用256替代 `t.size()` , 使用 `i` 替代 `t[i]` .

复杂度分析

两次遍历字符串，时间复杂度最坏情况下为 $O(2n)$, 使用了额外的数组，空间复杂度 $O(256)$.

题解2 - 排序字符串

另一直接的解法是对字符串先排序，若排序后的字符串内容相同，则其互为变位词。题解1中使用 hashmap 的方法对于比较两个字符串是否互为变位词十分有效，但是在比较多个字符串时，使用 hashmap 的方法复杂度则较高。

C++

```
class Solution {
public:
    /**
     * @param s: The first string
     * @param b: The second string
     * @return true or false
     */
    bool anagram(string s, string t) {
        if (s.empty() || t.empty()) {
            return false;
        }
        if (s.size() != t.size()) {
            return false;
        }

        sort(s.begin(), s.end());
        sort(t.begin(), t.end());

        if (s == t) {
            return true;
        } else {
            return false;
        }
    }
};
```

源码分析

对字符串 s 和 t 分别排序，而后比较是否含相同内容。对字符串排序时可以采用先统计字频再组装成排序后的字符串，效率更高一点。

复杂度分析

C++的 STL 中 sort 的时间复杂度介于 $O(n)$ 和 $O(n^2)$之间，判断 `s == t` 时间复杂度最坏为 $O(n)$.

Reference

- *CC150 Chapter 9.1* 中文版 p109

Compare Strings

Source

- lintcode: (55) Compare Strings

```
Compare two strings A and B, determine whether A contains all of the characters in B.

The characters in string A and B are all Upper Case letters.

Example
For A = "ABCD", B = "ABC", return true.

For A = "ABCD" B = "AABC", return false.
```

题解

题 Two Strings Are Anagrams | Data Structure and Algorithm 的变形题。题目意思是问B中的所有字符是否都在A中，而不是单个字符。比如B="AABC"包含两个「A」，而A="ABCD"只包含一个「A」，故返回false. 做题时注意题意，必要时可向面试官确认。

既然不是类似 strstr 那样的匹配，直接使用两重循环就不太合适了。题目中另外给的条件则是A和B都是全大写单词，理解题意后容易想到的方案就是先遍历 A 和 B 统计各字符出现的频次，然后比较频次大小即可。嗯，祭出万能的哈希表。

C++

```
class Solution {
public:
    /**
     * @param A: A string includes Upper Case letters
     * @param B: A string includes Upper Case letter
     * @return:  if string A contains all of the characters in B return true
     *           else return false
     */
    bool compareStrings(string A, string B) {
        if (A.size() < B.size()) {
            return false;
        }

        const int AlphabetNum = 26;
        int letterCount[AlphabetNum] = {0};
        for (int i = 0; i != A.size(); ++i) {
            ++letterCount[A[i] - 'A'];
        }
        for (int i = 0; i != B.size(); ++i) {
            --letterCount[B[i] - 'A'];
            if (letterCount[B[i] - 'A'] < 0) {
                return false;
```

```
            }
        }

        return true;
    }
};
```

源码解析

1. 异常处理，B 的长度大于 A 时必定返回 `false`, 包含了空串的特殊情况。
2. 使用额外的辅助空间，统计各字符的频次。

复杂度分析

遍历一次 A 字符串，遍历一次 B 字符串，时间复杂度最坏 $O(2n)$, 空间复杂度为 $O(26)$.

Anagrams

Source

- leetcode: Anagrams | LeetCode OJ
- lintcode: (171) Anagrams

```
Given an array of strings, return all groups of strings that are anagrams.

Example
Given ["lint", "intl", "inlt", "code"], return ["lint", "inlt", "intl"].

Given ["ab", "ba", "cd", "dc", "e"], return ["ab", "ba", "cd", "dc"].
Note
All inputs will be in lower-case
```

题解1 - 双重 `for` 循环(TLE)

题 Two Strings Are Anagrams 的升级版，容易想到的方法为使用双重 `for` 循环两两判断字符串数组是否互为变位字符串。但显然此法的时间复杂度较高。还需要 $O(n)$ 的数组来记录字符串是否被加入到最终结果中。

C++

```cpp
class Solution {
public:
    /**
     * @param strs: A list of strings
     * @return: A list of strings
     */
    vector<string> anagrams(vector<string> &strs) {
        if (strs.size() < 2) {
            return strs;
        }

        vector<string> result;
        vector<bool> visited(strs.size(), false);
        for (int s1 = 0; s1 != strs.size(); ++s1) {
            bool has_anagrams = false;
            for (int s2 = s1 + 1; s2 < strs.size(); ++s2) {
                if ((!visited[s2]) && isAnagrams(strs[s1], strs[s2])) {
                    result.push_back(strs[s2]);
                    visited[s2] = true;
                    has_anagrams = true;
                }
            }
            if ((!visited[s1]) && has_anagrams) result.push_back(strs[s1]);
        }
```

```
        return result;
    }

private:
    bool isAnagrams(string &s, string &t) {
        if (s.size() != t.size()) {
            return false;
        }

        const int AlphabetNum = 26;
        int letterCount[AlphabetNum] = {0};
        for (int i = 0; i != s.size(); ++i) {
            ++letterCount[s[i] - 'a'];
            --letterCount[t[i] - 'a'];
        }
        for (int i = 0; i != t.size(); ++i) {
            if (letterCount[t[i] - 'a'] < 0) {
                return false;
            }
        }

        return true;
    }
};
```

源码分析

1. strs 长度小于等于1时直接返回。
2. 使用与 strs 等长的布尔数组表示其中的字符串是否被添加到最终的返回结果中。
3. 双重循环遍历字符串数组，注意去重即可。
4. 私有方法 `isAnagrams` 用于判断两个字符串是否互为变位词。

复杂度分析

私有方法 `isAnagrams` 最坏的时间复杂度为 $O(2L)$, 其中 L 为字符串长度。双重 `for` 循环时间复杂度近似为 $\frac{1}{2}O(n^2)$, n 为给定字符串数组数目。总的时间复杂度近似为 $O(n^2L)$. 使用了含有26个元素的 int 数组，空间复杂度可认为是 $O(1)$.

题解2 - 排序 + hashmap

在题 Two Strings Are Anagrams 中曾介绍过使用排序和 hashmap 两种方法判断变位词。这里我们将这两种方法同时引入！只不过此时的 hashmap 的 key 为字符串，value 为该字符串在 vector 中出现的次数。两次遍历字符串数组，第一次遍历求得排序后的字符串数量，第二次遍历将排序后相同的字符串取出放入最终结果中。

leetcode 上此题的 signature 已经更新，需要将 anagrams 按组输出，稍微麻烦一点点。

C++ - lintcode

```
class Solution {
```

```
public:
    /**
     * @param strs: A list of strings
     * @return: A list of strings
     */
    vector<string> anagrams(vector<string> &strs) {
        unordered_map<string, int> hash;

        for (int i = 0; i < strs.size(); i++) {
            string str = strs[i];
            sort(str.begin(), str.end());
            ++hash[str];
        }

        vector<string> result;
        for (int i = 0; i < strs.size(); i++) {
            string str = strs[i];
            sort(str.begin(), str.end());
            if (hash[str] > 1) {
                result.push_back(strs[i]);
            }
        }

        return result;
    }
};
```

Java - leetcode

```
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> result = new ArrayList<List<String>>();
        if (strs == null) return result;

        // one key to multiple value multiMap
        Map<String, ArrayList<String>> multiMap = new HashMap<String, ArrayList<String>>(
        for (String str : strs) {
            char[] strChar = str.toCharArray();
            Arrays.sort(strChar);
            String strSorted = String.valueOf(strChar);
            if (multiMap.containsKey(strSorted)) {
                ArrayList<String> aList = multiMap.get(strSorted);
                aList.add(str);
                multiMap.put(strSorted, aList);
            } else {
                ArrayList<String> aList = new ArrayList<String>();
                aList.add(str);
                multiMap.put(strSorted, aList);
            }
        }

        // add List group to result
        Set<String> keySet = multiMap.keySet();
        for (String key : keySet) {
            ArrayList<String> aList = multiMap.get(key);
            Collections.sort(aList);
            result.add(aList);
```

```
        }

        return result;
    }
}
```

源码分析

建立 key 为字符串，value 为相应计数器的hashmap, `unordered_map` 为 C++ 11中引入的哈希表数据结构[unordered_map], 这种新的数据结构和之前的 map 有所区别，详见[map-unordered_map]。

第一次遍历字符串数组获得排序后的字符串计数器信息，第二次遍历字符串数组将哈希表中计数器值大于1的字符串取出。

leetcode 中题目 signature 已经有所变化，这里使用一对多的 HashMap 较为合适，使用 ArrayList 作为 value. Java 中对 String 排序可先将其转换为 char[], 排序后再转换为新的 String.

复杂度分析

遍历一次字符串数组，复杂度为 $O(n)$, 对单个字符串排序复杂度近似为 $O(L \log L)$. 两次遍历字符串数组，故总的时间复杂度近似为 $O(nL \log L)$. 使用了哈希表，空间复杂度为 $O(K)$, 其中 K 为排序后不同的字符串个数。

Reference

- [unordered_map]. unordered_map - C++ Reference ↩
- [map-unordered_map]. c++ - Choosing between std::map and std::unordered_map - Stack Overflow ↩
- Anagrams | 九章算法

Longest Common Substring

Source

- lintcode: (79) Longest Common Substring

```
Given two strings, find the longest common substring.
Return the length of it.

Example
Given A="ABCD", B="CBCE", return 2.
Note
The characters in substring should occur continuously in original string.
This is different with subsequence.
```

题解

求最长公共子串，注意「子串」和「子序列」的区别！简单考虑可以使用两根指针索引分别指向两个字符串的当前遍历位置，若遇到相等的字符时则同时向后移动一位。

C++

```
class Solution {
public:
    /**
     * @param A, B: Two string.
     * @return: the length of the longest common substring.
     */
    int longestCommonSubstring(string &A, string &B) {
        if (A.empty() || B.empty()) {
            return 0;
        }

        int lcs = 0, lcs_temp = 0;
        for (int i = 0; i < A.size(); ++i) {
            for (int j = 0; j < B.size(); ++j) {
                lcs_temp = 0;
                while ((i + lcs_temp < A.size()) &&\
                       (j + lcs_temp < B.size()) &&\
                       (A[i + lcs_temp] == B[j + lcs_temp]))
                {
                    ++lcs_temp;
                }

                // update lcs
                if (lcs_temp > lcs) {
                    lcs = lcs_temp;
                }
            }
        }
```

```
        return lcs;
    }
};
```

源码分析

1. 异常处理，空串时返回0.
2. 分别使用 `i` 和 `j` 表示当前遍历的索引处。若当前字符相同时则共同往后移动一位。
3. 没有相同字符时比较此次遍历的 `lcs_temp` 和 `lcs` 大小，更新 `lcs` .
4. 返回 `lcs` .

注意在 `while` 循环中不可直接使用 `++i` 或者 `++j` ，因为有可能会漏解！

复杂度分析

双重 for 循环，最坏时间复杂度约为 $O(mn \cdot lcs)$.

Reference

- Longest Common Substring | 九章算法

Rotate String

Source

- lintcode: (8) Rotate String

```
Given a string and an offset, rotate string by offset. (rotate from left to right)

Example
Given "abcdefg"

for offset=0, return "abcdefg"

for offset=1, return "gabcdef"

for offset=2, return "fgabcde"

for offset=3, return "efgabcd"

...
```

题解

常见的翻转法应用题，仔细观察规律可知翻转的分割点在从数组末尾数起的offset位置。先翻转前半部分，随后翻转后半部分，最后整体翻转。

Python

```python
class Solution:
    """
    param A: A string
    param offset: Rotate string with offset.
    return: Rotated string.
    """
    def rotateString(self, A, offset):
        if A is None or len(A) == 0:
            return A

        offset %= len(A)
        before = A[:len(A) - offset]
        after = A[len(A) - offset:]
        # [::-1] means reverse in Python
        A = before[::-1] + after[::-1]
        A = A[::-1]

        return A
```

C++

```
class Solution {
public:
  /**
     * param A: A string
     * param offset: Rotate string with offset.
     * return: Rotated string.
     */
    string rotateString(string A, int offset) {
        if (A.empty() || A.size() == 0) {
            return A;
        }

        int len = A.size();
        offset %= len;
        reverse(A, 0, len - offset - 1);
        reverse(A, len - offset, len - 1);
        reverse(A, 0, len - 1);
        return A;
    }

private:
    void reverse(string &str, int start, int end) {
        while (start < end) {
            char temp = str[start];
            str[start] = str[end];
            str[end] = temp;
            start++;
            end--;
        }
    }
};
```

Java

```
public class Solution {
    /*
     * param A: A string
     * param offset: Rotate string with offset.
     * return: Rotated string.
     */
    public char[] rotateString(char[] A, int offset) {
        if (A == null || A.length == 0) {
            return A;
        }

        int len = A.length;
        offset %= len;
        reverse(A, 0, len - offset - 1);
        reverse(A, len - offset, len - 1);
        reverse(A, 0, len - 1);

        return A;
    }

    private void reverse(char[] str, int start, int end) {
        while (start < end) {
```

```
            char temp = str[start];
            str[start] = str[end];
            str[end] = temp;
            start++;
            end--;
        }
    }
};
```

源码分析

1. 异常处理，A为空或者其长度为0
2. `offset` 可能超出A的大小，应模 `len` 后再用
3. 三步翻转法

Python 虽没有提供字符串的翻转，但用 slice 非常容易实现，非常 Pythonic!

复杂度分析

翻转一次时间复杂度近似为 $O(n)$, 原地交换，空间复杂度为 $O(1)$. 总共翻转3次，总的时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$.

Reference

- Reverse a string in Python - Stack Overflow

Reverse Words in a String

Source

- lintcode: (53) Reverse Words in a String

```
Given an input string, reverse the string word by word.

For example,
Given s = "the sky is blue",
return "blue is sky the".

Example
Clarification

- What constitutes a word?
A sequence of non-space characters constitutes a word.

- Could the input string contain leading or trailing spaces?
Yes. However, your reversed string should not contain leading or trailing spaces.

- How about multiple spaces between two words?
Reduce them to a single space in the reversed string.
```

题解

1. 由第一个提问可知：题中只有空格字符和非空格字符之分，因此空格字符应为其一关键突破口。
2. 由第二个提问可知：输入的前导空格或者尾随空格在反转后应去掉。
3. 由第三个提问可知：两个单词间的多个空格字符应合并为一个或删除掉。

首先找到各个单词(以空格隔开)，根据题目要求，单词应从后往前依次放入。正向取出比较麻烦，因此可尝试采用逆向思维——先将输入字符串数组中的单词从后往前逆序取出，取出单词后即翻转并append至新字符串数组。在append之前加入空格即可。

C++

```cpp
class Solution {
public:
    /**
     * @param s : A string
     * @return : A string
     */
    string reverseWords(string s) {
        if (s.empty()) {
            return s;
        }

        string s_ret, s_temp;
        string::size_type ix = s.size();
```

```
        while (ix != 0) {
            s_temp.clear();
            while (!isspace(s[--ix])) {
                s_temp.push_back(s[ix]);
                if (ix == 0) {
                    break;
                }
            }
            if (!s_temp.empty()) {
                if (!s_ret.empty()) {
                    s_ret.push_back(' ');
                }
                std::reverse(s_temp.begin(), s_temp.end());
                s_ret.append(s_temp);
            }
        }

        return s_ret;
    }
};
```

源码分析

1. 首先处理异常，s为空时直接返回空。
2. 索引初始值 `ix = s.size()` ，而不是 `ix = s.size() - 1` ，便于处理 `ix == 0` 时的特殊情况。
3. 使用额外空间 `s_ret, s_temp` ，空间复杂度为O(n)， `s_temp` 用于缓存临时的单词以append入 `s_ret` 。
4. 最后返回 `s_ret` 。

空间复杂度为O(1)的解法?

1. 处理异常及特殊情况
2. 处理多个空格及首尾空格
3. 记住单词的头尾指针，翻转之
4. 整体翻转

Valid Palindrome

- tags: [palindrome]

Source

- leetcode: Valid Palindrome | LeetCode OJ
- lintcode: (415) Valid Palindrome

```
Given a string, determine if it is a palindrome,
considering only alphanumeric characters and ignoring cases.

Example
"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is not a palindrome.
Note
Have you consider that the string might be empty?
This is a good question to ask during an interview.
For the purpose of this problem,
we define empty string as valid palindrome.

Challenge
O(n) time without extra memory.
```

题解

字符串的回文判断问题，由于字符串可随机访问，故逐个比较首尾字符是否相等最为便利，即常见的『两根指针』技法。此题忽略大小写，并只考虑字母和数字字符。链表的回文判断总结见 Check if a singly linked list is palindrome.

Python

```python
class Solution:
    # @param {string} s A string
    # @return {boolean} Whether the string is a valid palindrome
    def isPalindrome(self, s):
        if not s:
            return True

        l, r = 0, len(s) - 1

        while l < r:
            # find left alphanumeric character
            if not s[l].isalnum():
                l += 1
                continue
            # find right alphanumeric character
            if not s[r].isalnum():
```

```
                r -= 1
                continue
            # case insensitive compare
            if s[l].lower() == s[r].lower():
                l += 1
                r -= 1
            else:
                return False
        #
        return True
```

C++

```
class Solution {
public:
    /**
     * @param s A string
     * @return Whether the string is a valid palindrome
     */
    bool isPalindrome(string& s) {
        if (s.empty()) return true;

        int l = 0, r = s.size() - 1;
        while (l < r) {
            // find left alphanumeric character
            if (!isalnum(s[l])) {
                ++l;
                continue;
            }
            // find right alphanumeric character
            if (!isalnum(s[r])) {
                --r;
                continue;
            }
            // case insensitive compare
            if (tolower(s[l]) == tolower(s[r])) {
                ++l;
                --r;
            } else {
                return false;
            }
        }

        return true;
    }
};
```

Java

```
public class Solution {
    /**
     * @param s A string
     * @return Whether the string is a valid palindrome
     */
```

```java
    public boolean isPalindrome(String s) {
        if (s == null || s.isEmpty()) return true;

        int l = 0, r = s.length() - 1;
        while (l < r) {
            // find left alphanumeric character
            if (!Character.isLetterOrDigit(s.charAt(l))) {
                l++;
                continue;
            }
            // find right alphanumeric character
            if (!Character.isLetterOrDigit(s.charAt(r))) {
                r--;
                continue;
            }
            // case insensitive compare
            if (Character.toLowerCase(s.charAt(l)) == Character.toLowerCase(s.charAt(r)))
                l++;
                r--;
            } else {
                return false;
            }
        }

        return true;
    }
}
```

源码分析

两步走：

1. 找到最左边和最右边的第一个合法字符(字母或者字符)
2. 一致转换为小写进行比较

字符的判断尽量使用语言提供的 API

复杂度分析

两根指针遍历一次，时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

Longest Palindromic Substring

- tags: [palindrome]

Source

- leetcode: Longest Palindromic Substring | LeetCode OJ
- lintcode: (200) Longest Palindromic Substring

```
Given a string S, find the longest palindromic substring in S.
You may assume that the maximum length of S is 1000,
and there exists one unique longest palindromic substring.

Example
Given the string = "abcdzdcab", return "cdzdc".
Challenge
O(n2) time is acceptable. Can you do it in O(n) time.
```

题解1 - 穷竭搜索

最简单的方案，穷举所有可能的子串，判断子串是否为回文，使用一变量记录最大回文长度，若新的回文超过之前的最大回文长度则更新标记变量并记录当前回文的起止索引，最后返回最长回文子串。

Python

```python
class Solution:
    # @param {string} s input string
    # @return {string} the longest palindromic substring
    def longestPalindrome(self, s):
        if not s:
            return ""

        n = len(s)
        longest, left, right = 0, 0, 0
        for i in xrange(0, n):
            for j in xrange(i + 1, n + 1):
                substr = s[i:j]
                if self.isPalindrome(substr) and len(substr) > longest:
                    longest = len(substr)
                    left, right = i, j
        # construct longest substr
        result = s[left:right]
        return result

    def isPalindrome(self, s):
        if not s:
            return False
        # reverse compare
        return s == s[::-1]
```

C++

```
class Solution {
public:
    /**
     * @param s input string
     * @return the longest palindromic substring
     */
    string longestPalindrome(string& s) {
        string result;
        if (s.empty()) return s;

        int n = s.size();
        int longest = 0, left = 0, right = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j <= n; ++j) {
                string substr = s.substr(i, j - i);
                if (isPalindrome(substr) && substr.size() > longest) {
                    longest = j - i;
                    left = i;
                    right = j;
                }
            }
        }

        result = s.substr(left, right - left);
        return result;
    }

private:
    bool isPalindrome(string &s) {
        int n = s.size();
        for (int i = 0; i < n; ++i) {
            if (s[i] != s[n - i - 1]) return false;
        }
        return true;
    }
};
```

Java

```
public class Solution {
    /**
     * @param s input string
     * @return the longest palindromic substring
     */
    public String longestPalindrome(String s) {
        String result = new String();
        if (s == null || s.isEmpty()) return result;

        int n = s.length();
        int longest = 0, left = 0, right = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j <= n; j++) {
```

```
                String substr = s.substring(i, j);
                if (isPalindrome(substr) && substr.length() > longest) {
                    longest = substr.length();
                    left = i;
                    right = j;
                }
            }
        }

        result = s.substring(left, right);
        return result;
    }

    private boolean isPalindrome(String s) {
        if (s == null || s.isEmpty()) return false;

        int n = s.length();
        for (int i = 0; i < n; i++) {
            if (s.charAt(i) != s.charAt(n - i - 1)) return false;
        }

        return true;
    }
}
```

源码分析

使用 `left`，`right` 作为子串的起止索引，用于最后构造返回结果，避免中间构造字符串以减少开销。

复杂度分析

穷举所有的子串，$O(C_n^2) = O(n^2)$, 每次判断字符串是否为回文，复杂度为 $O(n)$, 故总的时间复杂度为 $O(n^3)$. 故大数据集下可能 TLE. 使用了 `substr` 作为临时子串，空间复杂度为 $O(n)$.

Reference

- Longest Palindromic Substring Part I | LeetCode
- Longest Palindromic Substring Part II | LeetCode

Space Replacement

Source

- lintcode: (212) Space Replacement

```
Write a method to replace all spaces in a string with %20.
The string is given in a characters array, you can assume it has enough space
for replacement and you are given the true length of the string.

Example
Given "Mr John Smith", length = 13.

The string after replacement should be "Mr%20John%20Smith".

Note
If you are using Java or Python, please use characters array instead of string.

Challenge
Do it in-place.
```

题解

根据题意，给定的输入数组长度足够长，将空格替换为 `%20` 后也不会溢出。通常的思维为从前向后遍历，遇到空格即将 `%20` 插入到新数组中，这种方法在生成新数组时很直观，但要求原地替换时就不方便了，这时可联想到插入排序的做法——从后往前遍历，空格处标记下就好了。由于不知道新数组的长度，故首先需要遍历一次原数组，字符串类题中常用方法。

需要注意的是这个题并未说明多个空格如何处理，如果多个连续空格也当做一个空格时稍有不同。

Java

```java
public class Solution {
    /**
     * @param string: An array of Char
     * @param length: The true length of the string
     * @return: The true length of new string
     */
    public int replaceBlank(char[] string, int length) {
        if (string == null) return 0;

        int space = 0;
        for (char c : string) {
            if (c == ' ') space++;
        }

        int r = length + 2 * space - 1;
        for (int i = length - 1; i >= 0; i--) {
            if (string[i] != ' ') {
```

```
                string[r] = string[i];
                r--;
            } else {
                string[r--] = '0';
                string[r--] = '2';
                string[r--] = '%';
            }
        }

        return length + 2 * space;
    }
}
```

源码分析

先遍历一遍求得空格数，得到『新数组』的实际长度，从后往前遍历。

复杂度分析

遍历两次原数组，时间复杂度近似为 $O(n)$, 使用了 `r` 作为标记，空间复杂度 $O(1)$.

Wildcard Matching

Source

- leetcode: Wildcard Matching | LeetCode OJ
- lintcode: (192) Wildcard Matching

```
Implement wildcard pattern matching with support for '?' and '*'.

'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial).

Example
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

题解1 - DFS

字符串的通配实现。'`?`'表示匹配单一字符，'`*`'可匹配任意多字符串(包含零个)。要匹配的字符串设为`s`，模式匹配用的字符串设为`p`，那么如果是普通字符，两个字符串索引向前推进一位即可，如果`p`中的字符是`?`也好办，同上处理，向前推进一位。所以现在的关键就在于如何处理'`*`', 因为`*`可匹配0, 1, 2...个字符，所以遇到`*`时，`s`应该尽可能的向前推进，注意到`p`中`*`后面可能跟有其他普通字符，故`s`向前推进多少位直接与`p`中`*`后面的字符相关。同时此时两个字符串的索引处即成为回溯点，如果后面的字符串匹配不成功，则`s`中的索引向前推进，向前推进的字符串即表示和`p`中`*`匹配的字符个数。

Java

```java
public class Solution {
    /**
     * @param s: A string
     * @param p: A string includes "?" and "*"
     * @return: A boolean
     */
    public boolean isMatch(String s, String p) {
        if (s == null || p == null) return false;
        if (s.length() == 0|| p.length() == 0) return false;

        return helper(s, 0, p, 0);
    }

    private boolean helper(String s, int si, String p, int pj) {
        // index out of range check
```

```
        if (si == s.length() || pj == p.length()) {
            if (si == s.length() && pj == p.length()) {
                return true;
            } else {
                return false;
            }
        }

        if (p.charAt(pj) == '*') {
            // remove coninuous *
            while (p.charAt(pj) == '*') {
                pj++;
                // index out of range check
                if (pj == p.length()) return true;
            }

            // compare remaining part of p after * with s
            while (si < s.length() && !helper(s, si, p, pj)) {
                si++;
            }
            // substring of p equals to s
            return si != s.length();
        } else if (s.charAt(si) == p.charAt(pj) || p.charAt(pj) == '?') {
            return helper(s, si + 1, p, pj + 1);
        } else {
            return false;
        }
    }
}
```

源码分析

其中对 `*` 的处理和递归回溯是这段代码的精华。

复杂度分析

最坏情况下需要不断回溯，时间复杂度 $O(n!) \times O(m!)$, 空间复杂度 $O(1)$(不含栈空间)。

Reference

- Soulmachine 的 leetcode 题解
-

Length of Last Word

Source

- leetcode: Length of Last Word | LeetCode OJ
- lintcode: (422) Length of Last Word

```
Given a string s consists of upper/lower-case alphabets and empty space characters ' ',
return the length of last word in the string.

If the last word does not exist, return 0.

Have you met this question in a real interview? Yes
Example
Given s = "Hello World", return 5.

Note
A word is defined as a character sequence consists of non-space characters only.
```

题解

关键点在于确定最后一个字符串之前的空格，此外还需要考虑末尾空格这一特殊情况，故首先除掉右边的空白字符比较好。

Java

```java
public class Solution {
    /**
     * @param s A string
     * @return the length of last word
     */
    public int lengthOfLastWord(String s) {
        if (s == null | s.isEmpty()) return 0;

        // trim right space
        int begin = 0, end = s.length();
        while (end > 0 && s.charAt(end - 1) == ' ') {
            end--;
        }
        // find the last space
        for (int i = 0; i < end; i++) {
            if (s.charAt(i) == ' ') {
                begin = i + 1;
            }
        }

        return end - begin;
    }
}
```

源码分析

两根指针。

复杂度分析

遍历一次，时间复杂度 $O(n)$.

Count and Say

Source

- leetcode: Count and Say | LeetCode OJ
- lintcode: (420) Count and Say

```
The count-and-say sequence is the sequence of integers beginning as follows:

1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.

11 is read off as "two 1s" or 21.

21 is read off as "one 2, then one 1" or 1211.

Given an integer n, generate the nth sequence.

Example
Given n = 5, return "111221".

Note
The sequence of integers will be represented as a string.
```

题解

题目大意是找第 n 个数(字符串表示)，规则则是对于连续字符串，表示为重复次数+数本身。

Java

```java
public class Solution {
    /**
     * @param n the nth
     * @return the nth sequence
     */
    public String countAndSay(int n) {
        if (n <= 0) return null;

        String s = "1";
        for (int i = 1; i < n; i++) {
            int count = 1;
            StringBuilder sb = new StringBuilder();
            int sLen = s.length();
            for (int j = 0; j < sLen; j++) {
                if (j < sLen - 1 && s.charAt(j) == s.charAt(j + 1)) {
                    count++;
                } else {
                    sb.append(count + "" + s.charAt(j));
                    // reset
```

```
                    count = 1;
                }
            }
            s = sb.toString();
        }

        return s;
    }
}
```

源码分析

字符串是动态生成的，故使用 StringBuilder 更为合适。注意s 初始化为"1", 第一重 for循环中注意循环的次数为 n-1.

复杂度分析

略

Reference

- [leetcode]Count and Say - 喵星人与汪星人

Integer Array - 整型数组

本章主要总结与整型数组相关的题。

Remove Element

Source

- leetcode: Remove Element | LeetCode OJ
- lintcode: (172) Remove Element

```
Given an array and a value, remove all occurrences of that value in place and return the

The order of elements can be changed, and the elements after the new length don't matter.

Example
Given an array [0,4,4,0,0,2,4,4], value=4

return 4 and front four elements of the array is [0,0,0,2]
```

题解1 - 使用容器

入门题，返回删除指定元素后的数组长度，使用容器操作非常简单。以 lintcode 上给出的参数为例，遍历容器内元素，若元素值与给定删除值相等，删除当前元素并往后继续遍历。

C++

```cpp
class Solution {
public:
    /**
     *@param A: A list of integers
     *@param elem: An integer
     *@return: The new length after remove
     */
    int removeElement(vector<int> &A, int elem) {
        for (vector<int>::iterator iter = A.begin(); iter < A.end(); ++iter) {
            if (*iter == elem) {
                iter = A.erase(iter);
                --iter;
            }
        }

        return A.size();
    }
};
```

源码分析

注意在遍历容器内元素和指定欲删除值相等时，需要先自减 `--iter`，因为 `for` 循环会对 `iter` 自增，`A.erase()` 删除当前元素值并返回指向下一个元素的指针，一增一减正好平衡。如果改用 `while` 循

环，则需注意访问数组时是否越界。

复杂度分析

由于vector每次erase的复杂度是$O(n)$，我们遍历整个数组，最坏情况下，每个元素都与要删除的目标元素相等，每次都要删除元素的复杂度高达$O(n^2)$ 观察此方法会如此低效的原因，是因为我们一次只删除一个元素，导致很多没必要的元素交换移动，如果能够将要删除的元素集中处理，则可以大幅增加效率，见题解2。

题解2 - 两根指针

由于题中明确暗示元素的顺序可变，且新长度后的元素不用理会。我们可以使用两根指针分别往前往后遍历，头指针用于指示当前遍历的元素位置，尾指针则用于在当前元素与欲删除值相等时替换当前元素，两根指针相遇时返回尾指针索引——即删除元素后「新数组」的长度。

C++

```
class Solution {
public:
    int removeElement(int A[], int n, int elem) {
        for (int i = 0; i < n; ++i) {
            if (A[i] == elem) {
                A[i] = A[n - 1];
                --i;
                --n;
            }
        }

        return n;
    }
};
```

源码分析

遍历当前数组，`A[i] == elem` 时将数组「尾部(以 n 为长度时的尾部)」元素赋给当前遍历的元素。同时自减 `i` 和 `n`，原因见题解1的分析。需要注意的是 `n` 在遍历过程中可能会变化。

复杂度分析

此方法只遍历一次数组，且每个循环的操作至多也不过仅是常数次，因此时间复杂度是$O(n)$。

Reference

- Remove Element | 九章算法

Zero Sum Subarray

Source

- lintcode: (138) Subarray Sum
- GeeksforGeeks: Find if there is a subarray with 0 sum - GeeksforGeeks

```
Given an integer array, find a subarray where the sum of numbers is zero.
Your code should return the index of the first number and the index of the last number.

Example
Given [-3, 1, 2, -3, 4], return [0, 2] or [1, 3].

Note
There is at least one subarray that it's sum equals to zero.
```

题解1 - 两重 for 循环

题目中仅要求返回一个子串(连续)中和为0的索引，而不必返回所有可能满足题意的解。最简单的想法是遍历所有子串，判断其和是否为0，使用两重循环即可搞定，最坏情况下时间复杂度为 $O(n^2)$, 这种方法显然是极其低效的，极有可能会出现 TLE. 下面就不浪费篇幅贴代码了。

题解2 - 比较子串和(TLE)

两重 for 循环显然是我们不希望看到的解法，那么我们再来分析下题意，题目中的对象是分析子串和，那么我们先从常见的对数组求和出发，$f(i) = \sum_0^i nums[i]$ 表示从数组下标 0 开始至下标 i 的和。子串和为 0，也就意味着存在不同的 i_1 和 i_2 使得 $f(i_1) - f(i_2) = 0$, 等价于 $f(i_1) = f(i_2)$. 思路很快就明晰了，使用一 vector 保存数组中从 0 开始到索引 `i` 的和，在将值 push 进 vector 之前先检查 vector 中是否已经存在，若存在则将相应索引加入最终结果并返回。

C++

```cpp
class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the first number
     *          and the index of the last number
     */
    vector<int> subarraySum(vector<int> nums){
        vector<int> result;

        int curr_sum = 0;
        vector<int> sum_i;
        for (int i = 0; i != nums.size(); ++i) {
```

```
            curr_sum += nums[i];

            if (0 == curr_sum) {
                result.push_back(0);
                result.push_back(i);
                return result;
            }

            vector<int>::iterator iter = find(sum_i.begin(), sum_i.end(), curr_sum);
            if (iter != sum_i.end()) {
                result.push_back(iter - sum_i.begin() + 1);
                result.push_back(i);
                return result;
            }

            sum_i.push_back(curr_sum);
        }

        return result;
    }
};
```

源码分析

使用 `curr_sum` 保存到索引 `i` 处的累加和， `sum_i` 保存不同索引处的和。执行 `sum_i.push_back` 之前先检查 `curr_sum` 是否为0，再检查 `curr_sum` 是否已经存在于 `sum_i` 中。是不是觉得这种方法会比题解1好？错！时间复杂度是一样一样的！根本原因在于 `find` 操作的时间复杂度为线性。与这种方法类似的有哈希表实现，哈希表的查找在理想情况下可认为是 $O(1)$.

复杂度分析

最坏情况下 $O(n^2)$, 实测和题解1中的方法运行时间几乎一致。

题解3 - 哈希表

终于到了祭出万能方法时候了，题解2可以认为是哈希表的雏形，而哈希表利用空间换时间的思路争取到了宝贵的时间资源 :)

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the first number
     *          and the index of the last number
     */
    vector<int> subarraySum(vector<int> nums){
        vector<int> result;
        // curr_sum for the first item, index for the second item
        map<int, int> hash;
```

```
        hash[0] = 0;

        int curr_sum = 0;
        for (int i = 0; i != nums.size(); ++i) {
            curr_sum += nums[i];
            if (hash.find(curr_sum) != hash.end()) {
                result.push_back(hash[curr_sum]);
                result.push_back(i);
                return result;
            } else {
                hash[curr_sum] = i + 1;
            }
        }

        return result;
    }
};
```

源码分析

为了将 `curr_sum == 0` 的情况也考虑在内，初始化哈希表后即赋予 `<0, 0>`. 给 `hash` 赋值时使用 `i + 1`, `push_back` 时则不必再加1.

由于 C++ 中的 `map` 采用红黑树实现，故其并非真正的「哈希表」，C++ 11中引入的 `unordered_map` 用作哈希表效率更高，实测可由1300ms 降至1000ms.

复杂度分析

遍历求和时间复杂度为 $O(n)$, 哈希表检查键值时间复杂度为 $O(\log L)$, 其中 L 为哈希表长度。如果采用 `unordered_map` 实现，最坏情况下查找的时间复杂度为线性，最好为常数级别。

题解4 - 排序

除了使用哈希表，我们还可使用排序的方法找到两个子串和相等的情况。这种方法的时间复杂度主要集中在排序方法的实现。由于除了记录子串和之外还需记录索引，故引入 `pair` 记录索引，最后排序时先按照 `sum` 值来排序，然后再按照索引值排序。如果需要自定义排序规则可参考[sort_pair_second].

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the first number
     *          and the index of the last number
     */
    vector<int> subarraySum(vector<int> nums){
        vector<int> result;
        if (nums.empty()) {
            return result;
        }
```

```
        const int num_size = nums.size();
        vector<pair<int, int> > sum_index(num_size + 1);
        for (int i = 0; i != num_size; ++i) {
            sum_index[i + 1].first = sum_index[i].first + nums[i];
            sum_index[i + 1].second = i + 1;
        }

        sort(sum_index.begin(), sum_index.end());
        for (int i = 1; i < num_size + 1; ++i) {
            if (sum_index[i].first == sum_index[i - 1].first) {
                result.push_back(sum_index[i - 1].second);
                result.push_back(sum_index[i].second - 1);
                return result;
            }
        }

        return result;
    }
};
```

源码分析

没啥好分析的，注意好边界条件即可。这里采用了链表中常用的「dummy」节点方法，`pair` 排序后即为我们需要的排序结果。这种排序的方法需要先求得所有子串和然后再排序，最后还需要遍历排序后的数组，效率自然是比不上哈希表。但是在某些情况下这种方法有一定优势。

复杂度分析

遍历求子串和，时间复杂度为 $O(n)$, 空间复杂度 $O(n)$. 排序时间复杂度近似 $O(n \log n)$, 遍历一次最坏情况下时间复杂度为 $O(n)$. 总的时间复杂度可近似为 $O(n \log n)$. 空间复杂度 $O(n)$.

扩展

这道题的要求是找到一个即可，但是要找出所有满足要求的解呢？Stackoverflow 上有这道延伸题的讨论 [stackoverflow].

另一道扩展题来自 Google 的面试题 - Find subarray with given sum - GeeksforGeeks.

Reference

- [stackoverflow]. algorithm - Zero sum SubArray - Stack Overflow ↩
- [sort_pair_second]. c++ - How do I sort a vector of pairs based on the second element of the pair? - Stack Overflow ↩

Subarray Sum K

Source

- GeeksforGeeks: Find subarray with given sum - GeeksforGeeks

```
Given an nonnegative integer array, find a subarray where the sum of numbers is k.
Your code should return the index of the first number and the index of the last number.

Example
Given [1, 4, 20, 3, 10, 5], sum k = 33, return [2, 4].
```

题解1 - 哈希表

题 Zero Sum Subarray | Data Structure and Algorithm 的升级版，这道题求子串和为 K 的索引。首先我们可以考虑使用时间复杂度相对较低的哈希表解决。前一道题的核心约束条件为 $f(i_1) - f(i_2) = 0$，这道题则变为 $f(i_1) - f(i_2) = k$

C++

```cpp
#include <iostream>
#include <vector>
#include <map>

using namespace std;

class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the first number
     *          and the index of the last number
     */
    vector<int> subarraySum(vector<int> nums, int k){
        vector<int> result;
        // curr_sum for the first item, index for the second item
        // unordered_map<int, int> hash;
        map<int, int> hash;
        hash[0] = 0;

        int curr_sum = 0;
        for (int i = 0; i != nums.size(); ++i) {
            curr_sum += nums[i];
            if (hash.find(curr_sum - k) != hash.end()) {
                result.push_back(hash[curr_sum - k]);
                result.push_back(i);
                return result;
            } else {
```

```
                hash[curr_sum] = i + 1;
            }
        }

        return result;
    }
};

int main(int argc, char *argv[])
{
    int int_array1[] = {1, 4, 20, 3, 10, 5};
    int int_array2[] = {1, 4, 0, 0, 3, 10, 5};
    vector<int> vec_array1;
    vector<int> vec_array2;
    for (int i = 0; i != sizeof(int_array1) / sizeof(int); ++i) {
        vec_array1.push_back(int_array1[i]);
    }
    for (int i = 0; i != sizeof(int_array2) / sizeof(int); ++i) {
        vec_array2.push_back(int_array2[i]);
    }

    Solution solution;
    vector<int> result1 = solution.subarraySum(vec_array1, 33);
    vector<int> result2 = solution.subarraySum(vec_array2, 7);

    cout << "result1 = [" << result1[0] << " ," << result1[1] << "]" << endl;
    cout << "result2 = [" << result2[0] << " ," << result2[1] << "]" << endl;

    return 0;
}
```

源码分析

与 Zero Sum Subarray 题的变化之处有两个地方，第一个是判断是否存在哈希表中时需要使用 `hash.find(curr_sum - k)`，最终返回结果使用 `result.push_back(hash[curr_sum - k]);` 而不是 `result.push_back(hash[curr_sum]);`

复杂度分析

略，见 Zero Sum Subarray | Data Structure and Algorithm

题解2 - 利用单调函数特性

不知道细心的你是否发现这道题的隐含条件——**nonnegative integer array**, 这也就意味着子串和函数 $f(i)$ 为「单调不减」函数。单调函数在数学中可是重点研究的对象，那么如何将这种单调性引入本题中呢？不妨设 $i_2 > i_1$, 题中的解等价于寻找 $f(i_2) - f(i_1) = k$, 则必有 $f(i_2) \geq k$.

我们首先来举个实际例子帮助分析，以整数数组 {1, 4, 20, 3, 10, 5} 为例，要求子串和为33的索引值。首先我们可以构建如下表所示的子串和 $f(i)$.

$f(i)$	1	5	25	28	38

i	0	1	2	3	4

要使部分子串和为33，则要求的第二个索引值必大于等于4，如果索引值再继续往后遍历，则所得的子串和必大于等于38，进而可以推断出索引0一定不是解。那现在怎么办咧？当然是把它扔掉啊！第一个索引值往后递推，直至小于33时又往后递推第二个索引值，于是乎这种技巧又可以认为是「两根指针」。

C++

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the first number
     *          and the index of the last number
     */
    vector<int> subarraySum2(vector<int> &nums, int k){
        vector<int> result;

        int left_index = 0, curr_sum = 0;
        for (int i = 0; i != nums.size(); ++i) {
            while (curr_sum > k) {
                curr_sum -= nums[left_index];
                ++left_index;
            }

            if (curr_sum == k) {
                result.push_back(left_index);
                result.push_back(i - 1);
                return result;
            }
            curr_sum += nums[i];
        }
        return result;
    }
};

int main(int argc, char *argv[])
{
    int int_array1[] = {1, 4, 20, 3, 10, 5};
    int int_array2[] = {1, 4, 0, 0, 3, 10, 5};
    vector<int> vec_array1;
    vector<int> vec_array2;
    for (int i = 0; i != sizeof(int_array1) / sizeof(int); ++i) {
        vec_array1.push_back(int_array1[i]);
    }
    for (int i = 0; i != sizeof(int_array2) / sizeof(int); ++i) {
        vec_array2.push_back(int_array2[i]);
    }

    Solution solution;
```

```
    vector<int> result1 = solution.subarraySum2(vec_array1, 33);
    vector<int> result2 = solution.subarraySum2(vec_array2, 7);

    cout << "result1 = [" << result1[0] << " ," << result1[1] << "]" << endl;
    cout << "result2 = [" << result2[0] << " ," << result2[1] << "]" << endl;

    return 0;
}
```

源码分析

使用 `for` 循环, 在 `curr_sum > k` 时使用 `while` 递减 `curr_sum` , 同时递增左边索引 `left_index` , 最后累加 `curr_sum` 。如果顺序不对就会出现 bug, 原因在于判断子串和是否满足条件时在递增之后(谢谢 @glbrtchen 汇报 bug)。

复杂度分析

看似有两重循环，由于仅遍历一次数组，且索引最多挪动和数组等长的次数。故最终时间复杂度近似为 $O(2n)$, 空间复杂度为 $O(1)$.

Reference

- Find subarray with given sum - GeeksforGeeks

Subarray Sum Closest

Source

- lintcode: (139) Subarray Sum Closest

```
Given an integer array, find a subarray with sum closest to zero.
Return the indexes of the first number and last number.

Example
Given [-3, 1, 1, -3, 5], return [0, 2], [1, 3], [1, 1], [2, 2] or [0, 4]

Challenge
O(nlogn) time
```

题解

题 Zero Sum Subarray | Data Structure and Algorithm 的变形题，由于要求的子串和不一定，故哈希表的方法不再适用，使用解法4 - 排序即可在 $O(n \log n)$ 内解决。具体步骤如下：

1. 首先遍历一次数组求得子串和。
2. 对子串和排序。
3. 逐个比较相邻两项差值的绝对值，返回差值绝对值最小的两项。

C++

```cpp
class Solution {
public:
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the first number
     *          and the index of the last number
     */
    vector<int> subarraySumClosest(vector<int> nums){
        vector<int> result;
        if (nums.empty()) {
            return result;
        }

        const int num_size = nums.size();
        vector<pair<int, int> > sum_index(num_size + 1);

        for (int i = 0; i < num_size; ++i) {
            sum_index[i + 1].first = sum_index[i].first + nums[i];
            sum_index[i + 1].second = i + 1;
        }

        sort(sum_index.begin(), sum_index.end());
```

```
        int min_diff = INT_MAX;
        int closest_index = 1;
        for (int i = 1; i < num_size + 1; ++i) {
            int sum_diff = abs(sum_index[i].first - sum_index[i - 1].first);
            if (min_diff > sum_diff) {
                min_diff = sum_diff;
                closest_index = i;
            }
        }

        int left_index = min(sum_index[closest_index - 1].second,\
                             sum_index[closest_index].second);
        int right_index = -1 + max(sum_index[closest_index - 1].second,\
                                   sum_index[closest_index].second);
        result.push_back(left_index);
        result.push_back(right_index);
        return result;
    }
};
```

源码分析

为避免对单个子串和是否为最小情形的单独考虑，我们可以采取类似链表 dummy 节点的方法规避，简化代码实现。故初始化 `sum_index` 时需要 `num_size + 1` 个。这里为避免 vector 反复扩充空间降低运行效率，使用 `resize` 一步到位。 `sum_index` 即最后结果中 `left_index` 和 `right_index` 等边界可以结合简单例子分析确定。

复杂度分析

1. 遍历一次求得子串和时间复杂度为 $O(n)$, 空间复杂度为 $O(n+1)$.
2. 对子串和排序，平均时间复杂度为 $O(n \log n)$.
3. 遍历排序后的子串和数组，时间复杂度为 $O(n)$.

总的时间复杂度为 $O(n \log n)$, 空间复杂度为 $O(n)$.

扩展

- algorithm - How to find the subarray that has sum closest to zero or a certain value t in O(nlogn) - Stack Overflow

Recover Rotated Sorted Array

Source

- lintcode: (39) Recover Rotated Sorted Array

```
Given a rotated sorted array, recover it to sorted array in-place.

Example
[4, 5, 1, 2, 3] -> [1, 2, 3, 4, 5]

Challenge
In-place, O(1) extra space and O(n) time.

Clarification
What is rotated array:

    - For example, the orginal array is [1,2,3,4], The rotated array of it can be [1,2,3,
```

首先可以想到逐步移位，但是这种方法显然太浪费时间，不可取。下面介绍利器『三步翻转法』，以 `[4, 5, 1, 2, 3]` 为例。

1. 首先找到分割点 `5` 和 `1`
2. 翻转前半部分 `4, 5` 为 `5, 4`，后半部分 `1, 2, 3` 翻转为 `3, 2, 1`。整个数组目前变为 `[5, 4, 3, 2, 1]`
3. 最后整体翻转即可得 `[1, 2, 3, 4, 5]`

由以上3个步骤可知其核心为『翻转』的in-place实现。使用两个指针，一个指头，一个指尾，使用for循环移位交换即可。

Java

```java
public class Solution {
    /**
     * @param nums: The rotated sorted array
     * @return: The recovered sorted array
     */
    public void recoverRotatedSortedArray(ArrayList<Integer> nums) {
        if (nums == null || nums.size() <= 1) {
            return;
        }

        int pos = 1;
        while (pos < nums.size()) { // find the break point
            if (nums.get(pos - 1) > nums.get(pos)) {
                break;
            }
            pos++;
```

```
        }
        myRotate(nums, 0, pos - 1);
        myRotate(nums, pos, nums.size() - 1);
        myRotate(nums, 0, nums.size() - 1);
    }

    private void myRotate(ArrayList<Integer> nums, int left, int right) { // in-place rot
        while (left < right) {
            int temp = nums.get(left);
            nums.set(left, nums.get(right));
            nums.set(right, temp);
            left++;
            right--;
        }
    }
}
```

C++

```
/**
 * forked from
 * http://www.jiuzhang.com/solutions/recover-rotated-sorted-array/
 */
class Solution {
private:
    void reverse(vector<int> &nums, vector<int>::size_type start, vector<int>::size_type
        for (vector<int>::size_type i = start, j = end; i < j; ++i, --j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }

public:
    void recoverRotatedSortedArray(vector<int> &nums) {
        for (vector<int>::size_type index = 0; index != nums.size() - 1; ++index) {
            if (nums[index] > nums[index + 1]) {
                reverse(nums, 0, index);
                reverse(nums, index + 1, nums.size() - 1);
                reverse(nums, 0, nums.size() - 1);

                return;
            }
        }
    }
};
```

源码分析

首先找到分割点，随后分三步调用翻转函数。简单起见可将 `vector<int>::size_type` 替换为 `int`

Product of Array Exclude Itself

Source

- lintcode: (50) Product of Array Exclude Itself
- GeeksforGeeks: A Product Array Puzzle - GeeksforGeeks

```
Given an integers array A.

Define B[i] = A[0] * ... * A[i-1] * A[i+1] * ... * A[n-1], calculate B WITHOUT divide ope

Example
For A=[1, 2, 3], return [6, 3, 2].
```

题解1 - 左右分治

根据题意，有 $result[i] = left[i] \cdot right[i]$, 其中 $left[i] = \prod_{j=0}^{i-1} A[j]$, $right[i] = \prod_{j=i+1}^{n-1} A[j]$. 即将最后的乘积分为两部分求解，首先求得左半部分的值，然后求得右半部分的值。最后将左右两半部分乘起来即为解。

C++

```cpp
class Solution {
public:
    /**
     * @param A: Given an integers array A
     * @return: A long long array B and B[i]= A[0] * ... * A[i-1] * A[i+1] * ... * A[n-1]
     */
    vector<long long> productExcludeItself(vector<int> &nums) {
        const int nums_size = nums.size();
        vector<long long> result(nums_size, 1);
        if (nums.empty() || nums_size == 1) {
            return result;
        }

        vector<long long> left(nums_size, 1);
        vector<long long> right(nums_size, 1);
        for (int i = 1; i != nums_size; ++i) {
            left[i] = left[i - 1] * nums[i - 1];
            right[nums_size - i - 1] = right[nums_size - i] * nums[nums_size - i];
        }
        for (int i = 0; i != nums_size; ++i) {
            result[i] = left[i] * right[i];
        }

        return result;
    }
};
```

源码分析

一次 `for` 循环求出左右部分的连乘积，下标的确定可使用简单例子辅助分析。

复杂度分析

两次 `for` 循环，时间复杂度 $O(n)$. 使用了左右两半部分辅助空间，空间复杂度 $O(2n)$.

题解2 - 原地求积

题解1中使用了左右两个辅助数组，但是仔细瞅瞅其实可以发现完全可以在最终返回结果 `result` 基础上原地计算左右两半部分的积。

C++

```
class Solution {
public:
    /**
     * @param A: Given an integers array A
     * @return: A long long array B and B[i]= A[0] * ... * A[i-1] * A[i+1] * ... * A[n-1]
     */
    vector<long long> productExcludeItself(vector<int> &nums) {
        const int nums_size = nums.size();
        vector<long long> result(nums_size, 1);

        // solve the left part first
        for (int i = 1; i < nums_size; ++i) {
            result[i] = result[i - 1] * nums[i - 1];
        }

        // solve the right part
        long long temp = 1;
        for (int i = nums_size - 1; i >= 0; --i) {
            result[i] *= temp;
            temp *= nums[i];
        }

        return result;
    }
};
```

源码分析

计算左半部分的递推式不用改，计算右半部分的乘积时由于会有左半部分值的干扰，故使用 `temp` 保存连乘的值。注意 `temp` 需要使用 `long long` , 否则会溢出。

复杂度分析

时间复杂度同上，空间复杂度为 $O(1)$.

Partition Array

Source

- (31) Partition Array

```
Given an array nums of integers and an int k, partition the array
(i.e move the elements in "nums") such that:

All elements < k are moved to the left
All elements >= k are moved to the right
Return the partitioning index, i.e the first index i nums[i] >= k.

Example
If nums=[3,2,2,1] and k=2, a valid answer is 1.

Note
You should do really partition in array nums instead of just
counting the numbers of integers smaller than k.

If all elements in nums are smaller than k, then return nums.length

Challenge
Can you partition the array in-place and in O(n)?
```

题解1 - 自左向右

容易想到的一个办法是自左向右遍历，使用 `right` 保存大于等于 k 的索引， `i` 则为当前遍历元素的索引，总是保持 `i >= right` , 那么最后返回的 `right` 即为所求。

C++

```cpp
class Solution {
public:
    int partitionArray(vector<int> &nums, int k) {
        int right = 0;
        const int size = nums.size();
        for (int i = 0; i < size; ++i) {
            if (nums[i] < k && i >= right) {
                int temp = nums[i];
                nums[i] = nums[right];
                nums[right] = temp;
                ++right;
            }
        }

        return right;
    }
};
```

源码分析

自左向右遍历，遇到小于 k 的元素时即和 `right` 索引处元素交换，并自增 `right` 指向下一个元素，这样就能保证 `right` 之前的元素一定小于 k. 注意 `if` 判断条件中 `i >= right` 不能是 `i > right`，否则需要对特殊情况如全小于 k 时的考虑，而且即使考虑了这一特殊情况也可能存在其他 bug. 具体是什么 bug 呢？欢迎提出你的分析意见~

复杂度分析

遍历一次数组，时间复杂度最少为 $O(n)$, 可能需要一定次数的交换。

题解2 - 两根指针

有了解过 Quick Sort 的做这道题自然是分分钟的事，使用左右两根指针 $left, right$ 分别代表小于、大于等于 k 的索引，左右同时开工，直至 $left > right$.

C++

```
class Solution {
public:
    int partitionArray(vector<int> &nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            while (left <= right && nums[left] < k) ++left;
            while (left <= right && nums[right] >= k) --right;
            if (left <= right) {
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
                ++left;
                --right;
            }
        }

        return left;
    }
};
```

源码分析

大循环能正常进行的条件为 $left <= right$, 对于左边索引，向右搜索直到找到小于 k 的索引为止；对于右边索引，则向左搜索直到找到大于等于 k 的索引为止。注意在使用 `while` 循环时务必进行越界检查！

找到不满足条件的索引时即交换其值，并递增 `left`，递减 `right`. 紧接着进行下一次循环。最后返回 `left` 即可，当 `nums` 为空时包含在 `left = 0` 之中，不必单独特殊考虑，所以应返回 `left` 而不是 `right`.

复杂度分析

只需要对整个数组遍历一次，时间复杂度为 $O(n)$, 相比题解1，题解2对全小于 k 的数组效率较高，元素交换次数较少。

Reference

- Partition Array | 九章算法

First Missing Positive

Source

- leetcode: First Missing Positive | LeetCode OJ
- lintcode: (189) First Missing Positive

```
Given an unsorted integer array, find the first missing positive integer.

Example
Given [1,2,0] return 3, and [3,4,-1,1] return 2.

Challenge
Your algorithm should run in O(n) time and uses constant space.
```

题解

容易想到的方案是先排序，然后遍历求得缺的最小整数。排序算法中常用的基于比较的方法时间复杂度的理论下界为 $O(n \log n)$, 不符题目要求。常见的能达到线性时间复杂度的排序算法有 基数排序，计数排序 和 桶排序。

基数排序显然不太适合这道题，计数排序对元素落在一定区间且重复值较多的情况十分有效，且需要额外的 $O(n)$ 空间，对这道题不太合适。最后就只剩下桶排序了，桶排序通常需要按照一定规则将值放入桶中，一般需要额外的 $O(n)$ 空间，咋看一下似乎不太适合在这道题中使用，但是若能设定一定的规则原地交换原数组的值呢？这道题的难点就在于这种规则的设定。

设想我们对给定数组使用桶排序的思想排序，第一个桶放1，第二个桶放2，如果找不到相应的数，则相应的桶的值不变(可能为负值，也可能为其他值)。

那么怎么才能做到原地排序呢？即若 $A[i] = x$, 则将 x 放到它该去的地方 - $A[x-1] = x$, 同时将原来 $A[x-1]$ 地方的值交换给 $A[i]$.

排好序后遍历桶，如果不满足 $f[i] = i + 1$, 那么警察叔叔就是它了！如果都满足条件怎么办？那就返回给定数组大小再加1呗。

C++

```
class Solution {
public:
    /**
     * @param A: a vector of integers
     * @return: an integer
     */
    int firstMissingPositive(vector<int> A) {
        const int size = A.size();

        for (int i = 0; i < size; ++i) {
```

```
            while (A[i] > 0 && A[i] <= size && \
                  (A[i] != i + 1) && (A[i] != A[A[i] - 1])) {
                int temp = A[A[i] - 1];
                A[A[i] - 1] = A[i];
                A[i] = temp;
            }
        }

        for (int i = 0; i < size; ++i) {
            if (A[i] != i + 1) {
                return i + 1;
            }
        }

        return size + 1;
    }
};
```

源码分析

核心代码为那几行交换，但是要很好地处理各种边界条件则要下一番功夫了，要能正常的交换，需满足以下几个条件：

1. `A[i]` 为正数，负数和零都无法在桶中找到生存空间...
2. `A[i] \leq size` 当前索引处的值不能比原数组容量大，大了的话也没用啊，肯定不是缺的第一个正数。
3. `A[i] != i + 1`，都满足条件了还交换个毛线，交换也是自身的值。
4. `A[i] != A[A[i] - 1]`，避免欲交换的值和自身相同，否则有重复值时会产生死循环。

如果满足以上四个条件就可以愉快地交换彼此了，使用 `while` 循环处理，此时 `i` 并不自增，直到将所有满足条件的索引处理完。

注意交换的写法，若写成

```
int temp = A[i];
A[i] = A[A[i] - 1];
A[A[i] - 1] = temp;
```

这又是满满的 bug :(因为在第三行中 `A[i]` 已不再是之前的值，第二行赋值时已经改变，故源码中的写法比较安全。

最后遍历桶排序后的数组，若在数组大小范围内找到不满足条件的解，直接返回，否则就意味着原数组给的元素都是从1开始的连续正整数，返回数组大小加1即可。

复杂度分析

「桶排序」需要遍历一次原数组，考虑到 `while` 循环也需要一定次数的遍历，故时间复杂度至少为 $O(n)$. 最后求索引值最多遍历一次排序后数组，时间复杂度最高为 $O(n)$, 用到了 `temp` 作为中间交换变量，空间复杂度为 $O(1)$.

Reference

- Find First Missing Positive | N00tc0d3r
- LeetCode: First Missing Positive 解题报告 - Yu's Garden - 博客园
- First Missing Positive | 九章算法

2 Sum

Source

- leetcode: Two Sum | LeetCode OJ
- lintcode: (56) 2 Sum

```
Given an array of integers, find two numbers such that they add up to a specific target n

The function twoSum should return indices of the two numbers
such that they add up to the target, where index1 must be less than index2.
Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
```

题解1 - 哈希表

找两数之和是否为 `target`，如果是找数组中一个值为 `target` 该多好啊！遍历一次就知道了，我只想说，too naive... 难道要将数组中所有元素的两两组合都求出来与 `target` 比较吗？时间复杂度显然为 $O(n^2)$，显然不符题目要求。找一个数时直接遍历即可，那么可不可以将两个数之和转换为找一个数呢？我们先来看看两数之和为 `target` 所对应的判断条件—— $x_i + x_j = target$，可进一步转化为 $x_i = target - x_j$，其中 i 和 j 为数组中的下标。一段神奇的数学推理就将找两数之和转化为了找一个数是否在数组中了！可见数学是多么的重要...

基本思路有了，现在就来看看怎么实现，显然我们需要额外的空间(也就是哈希表)来保存已经处理过的 x_j，如果不满足等式条件，那么我们就往后遍历，并把之前的元素加入到哈希表中，如果 `target` 减去当前索引后的值在哈希表中找到了，那么就将哈希表中相应的索引返回，大功告成！

C++

```cpp
class Solution {
public:
    /*
     * @param numbers : An array of Integer
     * @param target : target = numbers[index1] + numbers[index2]
     * @return : [index1+1, index2+1] (index1 < index2)
     */
    vector<int> twoSum(vector<int> &nums, int target) {
        vector<int> result;
        const int length = nums.size();
        if (0 == length) {
            return result;
```

```
        }

        // first value, second index
        unordered_map<int, int> hash(length);
        for (int i = 0; i != length; ++i) {
            if (hash.find(target - nums[i]) != hash.end()) {
                result.push_back(hash[target - nums[i]]);
                result.push_back(i + 1);
                return result;
            } else {
                hash[nums[i]] = i + 1;
            }
        }

        return result;
    }
};
```

源码分析

1. 异常处理。
2. 使用 C++ 11 中的哈希表实现 `unordered_map` 映射值和索引。
3. 找到满足条件的解就返回，找不到就加入哈希表中。注意题中要求返回索引值的含义。

复杂度分析

哈希表用了和数组等长的空间，空间复杂度为 $O(n)$, 遍历一次数组，时间复杂度为 $O(n)$.

Python

```
class Solution:
    """
    @param numbers : An array of Integer
    @param target : target = numbers[index1] + numbers[index2]
    @return : [index1 + 1, index2 + 1] (index1 < index2)
    """
    def twoSum(self, numbers, target):
        hashdict = {}
        for i, item in enumerate(numbers):
            if (target - item) in hashdict:
                return (hashdict[target - item] + 1, i + 1)
            hashdict[item] = i

        return (-1, -1)
```

源码分析

Python 中的 `dict` 就是天然的哈希表，使用 enumerate 可以同时返回索引和值，甚为方便。按题意似乎是要返回 list, 但个人感觉返回 tuple 更为合理。最后如果未找到符合题意的索引，返回 `(-1, -1)` .

题解2 - 排序后使用两根指针

但凡可以用空间换时间的做法，往往也可以使用时间换空间。另外一个容易想到的思路就是先对数组排序，然后使用两根指针分别指向首尾元素，逐步向中间靠拢，直至找到满足条件的索引为止。

C++

```
class Solution {
public:
    /*
     * @param numbers : An array of Integer
     * @param target : target = numbers[index1] + numbers[index2]
     * @return : [index1+1, index2+1] (index1 < index2)
     */
    vector<int> twoSum(vector<int> &nums, int target) {
        vector<int> result;
        const int length = nums.size();
        if (0 == length) {
            return result;
        }

        // first num, second is index
        vector<pair<int, int> > num_index(length);
        // map num value and index
        for (int i = 0; i != length; ++i) {
            num_index[i].first = nums[i];
            num_index[i].second = i + 1;
        }

        sort(num_index.begin(), num_index.end());
        int start = 0, end = length - 1;
        while (start < end) {
            if (num_index[start].first + num_index[end].first > target) {
                --end;
            } else if(num_index[start].first + num_index[end].first == target) {
                int min_index = min(num_index[start].second, num_index[end].second);
                int max_index = max(num_index[start].second, num_index[end].second);
                result.push_back(min_index);
                result.push_back(max_index);
                return result;
            } else {
                ++start;
            }
        }

        return result;
    }
};
```

源码分析

1. 异常处理。
2. 使用 `length` 保存数组的长度，避免反复调用 `nums.size()` 造成性能损失。
3. 使用 `pair` 组合排序前的值和索引，避免排序后找不到原有索引信息。
4. 使用标准库函数排序。

5. 两根指针指头尾，逐步靠拢。

复杂度分析

遍历一次原数组得到 `pair` 类型的新数组，时间复杂度为 $O(n)$, 空间复杂度也为 $O(n)$. 标准库中的排序方法时间复杂度近似为 $O(n \log n)$, 两根指针遍历数组时间复杂度为 $O(n)$.

> lintcode 上的题要求时间复杂度在 $O(n \log n)$ 时，空间复杂度为 $O(1)$, 但问题是排序后索引会乱掉，如果要保存之前的索引，空间复杂度一定是 $O(n)$，所以个人认为不存在较为简洁的 $O(1)$ 实现。如果一定要 $O(n)$ 的空间复杂度，那么只能用暴搜了，此时的时间复杂度为 $O(n^2)$.

3 Sum

Source

- leetcode: 3Sum | LeetCode OJ
- lintcode: (57) 3 Sum

```
Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0?
Find all unique triplets in the array which gives the sum of zero.

Example
For example, given array S = {-1 0 1 2 -1 -4}, A solution set is:

(-1, 0, 1)
(-1, -1, 2)
Note
Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)

The solution set must not contain duplicate triplets.
```

题解1 - 排序 + 哈希表 + 2 Sum

相比之前的 2 Sum, 3 Sum 又多加了一个数，按照之前 2 Sum 的分解为『1 Sum + 1 Sum』的思路，我们同样可以将 3 Sum 分解为『1 Sum + 2 Sum』的问题，具体就是首先对原数组排序，排序后选出第一个元素，随后在剩下的元素中使用 2 Sum 的解法。

Python

```python
class Solution:
    """
    @param numbersbers : Give an array numbersbers of n integer
    @return : Find all unique triplets in the array which gives the sum of zero.
    """
    def threeSum(self, numbers):
        triplets = []
        length = len(numbers)
        if length < 3:
            return triplets

        numbers.sort()
        for i in xrange(length):
            target = 0 - numbers[i]
            # 2 Sum
            hashmap = {}
            for j in xrange(i + 1, length):
                item_j = numbers[j]
                if (target - item_j) in hashmap:
                    triplet = [numbers[i], target - item_j, item_j]
```

```
                    if triplet not in triplets:
                        triplets.append(triplet)
                else:
                    hashmap[item_j] = j

        return triplets
```

源码分析

1. 异常处理，对长度小于3的直接返回。
2. 排序输入数组，有助于提高效率和返回有序列表。
3. 循环遍历排序后数组，先取出一个元素，随后求得 2 Sum 中需要的目标数。
4. 由于本题中最后返回结果不能重复，在加入到最终返回值之前查重。

由于排序后的元素已经按照大小顺序排列，且在2 Sum 中先遍历的元素较小，所以无需对列表内元素再排序。

复杂度分析

排序时间复杂度 $O(n \log n)$, 两重 `for` 循环，时间复杂度近似为 $O(n^2)$，使用哈希表(字典)实现，空间复杂度为 $O(n)$.

目前这段源码为比较简易的实现，leetcode 上的运行时间为500 + ms, 还有较大的优化空间，嗯，后续再进行优化。

C++

```
class Solution {
public:
    vector<vector<int> > threeSum(vector<int> &num)
    {
        vector<vector<int> > result;
        if (num.size() < 3) return result;

        int ans = 0;

        sort(num.begin(), num.end());

        for (int i = 0;i < num.size() - 2; ++i)
        {
            if (i > 0 && num[i] == num[i - 1])
                continue;
            int j = i + 1;
            int k = num.size() - 1;

            while (j < k)
            {
                ans = num[i] + num[j] + num[k];

                if (ans == 0)
                {
                    result.push_back({num[i], num[j], num[k]});
```

```
                    ++j;
                    while (j < num.size() && num[j] == num[j - 1])
                        ++j;
                    --k;
                    while (k >= 0 && num[k] == num[k + 1])
                        --k;
                }
                else if (ans > 0)
                    --k;
                else
                    ++j;
            }
        }

        return result;
    }
};
```

源码分析

同python解法不同，没有使用hash map

```
S = {-1 0 1 2 -1 -4}
排序后:
S = {-4 -1 -1 0 1 2}
      ↑  ↑        ↑
      i  j        k
         →        ←
i每轮只走一步，j和k根据S[i]+S[j]+S[k]=ans和0的关系进行移动，且j只向后走（即S[j]只增大），k只向前走（即
如果ans>0说明S[k]过大，k向前移；如果ans<0说明S[j]过小，j向后移；ans==0即为所求。
至于如何取到所有解，看代码即可理解，不再赘述。
```

复杂度分析

外循环i走了n轮,每轮j和k一共走n-i步，所以时间复杂度为$O(n^2)$。 最终运行时间为52ms

Reference

- 3Sum | 九章算法
- A simply Python version based on 2sum - O(n^2) - Leetcode Discuss

3 Sum Closest

Source

- leetcode: 3Sum Closest | LeetCode OJ
- lintcode: (59) 3 Sum Closest

```
Given an array S of n integers, find three integers in S such that the sum is closest to
Return the sum of the three integers. You may assume that each input would have exactly o

For example, given array S = {-1 2 1 -4}, and target = 1.
The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

题解1 - 排序 + 2 Sum + 两根指针 + 优化过滤

和 3 Sum 的思路接近，首先对原数组排序，随后将3 Sum 的题拆解为『1 Sum + 2 Sum』的题，对于 Closest 的题使用两根指针而不是哈希表的方法较为方便。对于有序数组来说，在查找 Cloest 的值时其实是有较大的优化空间的。

Python

```python
class Solution:
    """
    @param numbers: Give an array numbers of n integer
    @param target : An integer
    @return : return the sum of the three integers, the sum closest target.
    """
    def threeSumClosest(self, numbers, target):
        result = 2**31 - 1
        length = len(numbers)
        if length < 3:
            return result

        numbers.sort()
        larger_count = 0
        for i, item_i in enumerate(numbers):
            start = i + 1
            end = length - 1
            # optimization 1 - filter the smallest sum greater then target
            if start < end:
                sum3_smallest = numbers[start] + numbers[start + 1] + item_i
                if sum3_smallest > target:
                    larger_count += 1
                    if larger_count > 1:
                        return result

            while (start < end):
                sum3 = numbers[start] + numbers[end] + item_i
```

```
                if abs(sum3 - target) < abs(result - target):
                    result = sum3

                # optimization 2 - filter the sum3 closest to target
                sum_flag = 0
                if sum3 > target:
                    end -= 1
                    if sum_flag == -1:
                        break
                    sum_flag = 1
                elif sum3 < target:
                    start += 1
                    if sum_flag == 1:
                        break
                    sum_flag = -1
                else:
                    return result

        return result
```

源码分析

1. leetcode 上不让自己导入 `sys` 包，保险起见就初始化了 `result` 为还算较大的数，作为异常的返回值。
2. 对数组进行排序。
3. 依次遍历排序后的数组，取出一个元素 `item_i` 后即转化为『2 Sum Cloest』问题。『2 Sum Cloest』的起始元素索引为 `i + 1` ，之前的元素不能参与其中。
4. 优化一——由于已经对原数组排序，故遍历原数组时比较最小的三个元素和 `target` 值，若第二次大于 `target` 果断就此罢休，后面的值肯定越来越大。
5. 两根指针求『2 Sum Cloest』，比较 `sum3` 和 `result` 与 `target` 的差值的绝对值，更新 `result` 为较小的绝对值。
6. 再度对『2 Sum Cloest』进行优化，仍然利用有序数组的特点，若处于『一大一小』的临界值时就可以马上退出了，后面的元素与 `target` 之差的绝对值只会越来越大。

复杂度分析

对原数组排序，平均时间复杂度为 $O(n \log n)$, 两重 `for` 循环，由于有两处优化，故最坏的时间复杂度才是 $O(n^2)$, 使用了 `result` 作为临时值保存最接近 `target` 的值，两处优化各使用了一个辅助变量，空间复杂度 $O(1)$.

C++

```
class Solution {
public:
    int threeSumClosest(vector<int> &num, int target)
    {
        if (num.size() <= 3) return accumulate(num.begin(), num.end(), 0);
        sort (num.begin(), num.end());

        int result = 0, n = num.size(), temp;
```

```
        result = num[0] + num[1] + num[2];
        for (int i = 0; i < n - 2; ++i)
        {
            int j = i + 1, k = n - 1;
            while (j < k)
            {
                temp = num[i] + num[j] + num[k];

                if (abs(target - result) > abs(target - temp))
                    result = temp;
                if (result == target)
                    return result;
                ( temp > target ) ? --k : ++j;
            }
        }
        return result;
    }
};
```

源码分析

和前面3Sum解法相似，同理使用i,j,k三个指针进行循环。
区别在于3sum中的target为0，这里新增一个变量用于比较哪组数据与target更为相近

复杂度分析

时间复杂度同理为$O(n^2)$ 运行时间 16ms

Reference

- 3Sum Closest | 九章算法

Remove Duplicates from Sorted Array

Source

- leetcode: Remove Duplicates from Sorted Array | LeetCode OJ
- lintcode: (100) Remove Duplicates from Sorted Array

```
Given a sorted array, remove the duplicates in place
such that each element appear only once and return the new length.

Do not allocate extra space for another array,
you must do this in place with constant memory.

For example,
Given input array A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

Example
```

题解

使用两根指针(下标)，一个指针(下标)遍历数组，另一个指针(下标)只取不重复的数置于原数组中。

C++

```cpp
class Solution {
public:
    /**
     * @param A: a list of integers
     * @return : return an integer
     */
    int removeDuplicates(vector<int> &nums) {
        if (nums.size() <= 1) return nums.size();

        int len = nums.size();
        int newIndex = 0;
        for (int i = 1; i< len; ++i) {
            if (nums[i] != nums[newIndex]) {
                newIndex++;
                nums[newIndex] = nums[i];
            }
        }

        return newIndex + 1;
    }
};
```

Java

```
public class Solution {
    /**
     * @param A: a array of integers
     * @return : return an integer
     */
    public int removeDuplicates(int[] nums) {
        if (nums == null) return -1;
        if (nums.length <= 1) return nums.length;

        int newIndex = 0;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] != nums[newIndex]) {
                newIndex++;
                nums[newIndex] = nums[i];
            }
        }

        return newIndex + 1;
    }
}
```

源码分析

注意最后需要返回的是索引值加1。

复杂度分析

遍历一次数组，时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

Remove Duplicates from Sorted Array II

Source

- leetcode: Remove Duplicates from Sorted Array II | LeetCode OJ
- lintcode: (101) Remove Duplicates from Sorted Array II

```
Follow up for "Remove Duplicates":
What if duplicates are allowed at most twice?

For example,
Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].
Example
```

题解

在上题基础上加了限制条件元素最多可重复出现两次。~~因此可以在原题的基础上添加一变量跟踪元素重复出现的次数，小于指定值时执行赋值操作。但是需要注意的是重复出现次数~~`occurence`~~的初始值(从1开始，而不是0)和reset的时机。~~这种方法比较复杂，谢谢 @meishenme 提供的简洁方法，核心思想仍然是两根指针，只不过此时新索引自增的条件是当前遍历的数组值和『新索引』或者『新索引-1』两者之一不同。

C++

```cpp
class Solution {
public:
    /**
     * @param A: a list of integers
     * @return : return an integer
     */
    int removeDuplicates(vector<int> &nums) {
        if (nums.size() <= 2) return nums.size();

        int len = nums.size();
        int newIndex = 1;
        for (int i = 2; i < len; ++i) {
            if (nums[i] != nums[newIndex] || nums[i] != nums[newIndex - 1]) {
                ++newIndex;
                nums[newIndex] = nums[i];
            }
        }

        return newIndex + 1;
    }
};
```

Java

```
public class Solution {
    /**
     * @param A: a array of integers
     * @return : return an integer
     */
    public int removeDuplicates(int[] nums) {
        if (nums == null) return -1;
        if (nums.length <= 2) return nums.length;

        int newIndex = 1;
        for (int i = 2; i < nums.length; i++) {
            if (nums[i] != nums[newIndex] || nums[i] != nums[newIndex - 1]) {
                newIndex++;
                nums[newIndex] = nums[i];
            }
        }

        return newIndex + 1;
    }
}
```

源码分析

遍历数组时 i 从2开始，newIndex 初始化为1便于分析。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

Merge Sorted Array

Source

- leetcode: Merge Sorted Array | LeetCode OJ
- lintcode: (6) Merge Sorted Array

```
Given two sorted integer arrays A and B, merge B into A as one sorted array.

Example
A = [1, 2, 3, empty, empty], B = [4, 5]

After merge, A will be filled as [1, 2, 3, 4, 5]

Note
You may assume that A has enough space (size that is greater or equal to m + n)
to hold additional elements from B.
The number of elements initialized in A and B are m and n respectively.
```

题解

因为本题有 in-place 的限制，故必须从数组末尾的两个元素开始比较；否则就会产生挪动，一旦挪动就会是 $O(n^2)$ 的。 自尾部向首部逐个比较两个数组内的元素，取较大的置于数组 A 中。由于 A 的容量较 B 大，故最后 `m == 0` 或者 `n == 0` 时仅需处理 B 中的元素，因为 A 中的元素已经在 A 中，无需处理。

Python

```python
class Solution:
    """
    @param A: sorted integer array A which has m elements,
              but size of A is m+n
    @param B: sorted integer array B which has n elements
    @return: void
    """
    def mergeSortedArray(self, A, m, B, n):
        if B is None:
            return A

        index = m + n - 1
        while m > 0 and n > 0:
            if A[m - 1] > B[n - 1]:
                A[index] = A[m - 1]
                m -= 1
            else:
                A[index] = B[n - 1]
                n -= 1
            index -= 1

        # B has elements left
```

```
        while n > 0:
            A[index] = B[n - 1]
            n -= 1
            index -= 1
```

C++

```
class Solution {
public:
    /**
     * @param A: sorted integer array A which has m elements,
     *           but size of A is m+n
     * @param B: sorted integer array B which has n elements
     * @return: void
     */
    void mergeSortedArray(int A[], int m, int B[], int n) {
        int index = m + n - 1;
        while (m > 0 && n > 0) {
            if (A[m - 1] > B[n - 1]) {
                A[index] = A[m - 1];
                --m;
            } else {
                A[index] = B[n - 1];
                --n;
            }
            --index;
        }

        // B has elements left
        while (n > 0) {
            A[index] = B[n - 1];
            --n;
            --index;
        }
    }
};
```

Java

```
class Solution {
    /**
     * @param A: sorted integer array A which has m elements,
     *           but size of A is m+n
     * @param B: sorted integer array B which has n elements
     * @return: void
     */
    public void mergeSortedArray(int[] A, int m, int[] B, int n) {
        if (A == null || B == null) return;

        int index = m + n - 1;
        while (m > 0 && n > 0) {
            if (A[m - 1] > B[n - 1]) {
                A[index] = A[m - 1];
                m--;
```

```
            } else {
                A[index] = B[n - 1];
                n--;
            }
            index--;
        }

        // B has elements left
        while (n > 0) {
            A[index] = B[n - 1];
            n--;
            index--;
        }
    }
}
```

源码分析

第一个 while 只能用条件与。

复杂度分析

最坏情况下需要遍历两个数组中所有元素，时间复杂度为 $O(n)$. 空间复杂度 $O(1)$.

Merge Sorted Array II

Source

- lintcode: (64) Merge Sorted Array II

```
Merge two given sorted integer array A and B into a new sorted integer array.

Example
A=[1,2,3,4]

B=[2,4,5,6]

return [1,2,2,3,4,4,5,6]

Challenge
How can you optimize your algorithm
if one array is very large and the other is very small?
```

题解

上题要求 in-place, 此题要求返回新数组。由于可以生成新数组，故使用常规思路按顺序遍历即可。

Python

```python
class Solution:
    #@param A and B: sorted integer array A and B.
    #@return: A new sorted integer array
    def mergeSortedArray(self, A, B):
        if A is None or len(A) == 0:
            return B
        if B is None or len(B) == 0:
            return A

        C = []
        aLen, bLen = len(A), len(B)
        i, j = 0, 0
        while i < aLen and j < bLen:
            if A[i] < B[j]:
                C.append(A[i])
                i += 1
            else:
                C.append(B[j])
                j += 1

        # A has elements left
        while i < aLen:
            C.append(A[i])
            i += 1
```

```
        # B has elements left
        while j < bLen:
            C.append(B[j])
            j += 1

        return C
```

C++

```
class Solution {
public:
    /**
     * @param A and B: sorted integer array A and B.
     * @return: A new sorted integer array
     */
    vector<int> mergeSortedArray(vector<int> &A, vector<int> &B) {
        if (A.empty()) return B;
        if (B.empty()) return A;

        int aLen = A.size(), bLen = B.size();
        vector<int> C;
        int i = 0, j = 0;
        while (i < aLen && j < bLen) {
            if (A[i] < B[j]) {
                C.push_back(A[i]);
                ++i;
            } else {
                C.push_back(B[j]);
                ++j;
            }
        }

        // A has elements left
        while (i < aLen) {
            C.push_back(A[i]);
            ++i;
        }

        // B has elements left
        while (j < bLen) {
            C.push_back(B[j]);
            ++j;
        }

        return C;
    }
};
```

Java

```
class Solution {
    /**
     * @param A and B: sorted integer array A and B.
     * @return: A new sorted integer array
```

```
     */
    public ArrayList<Integer> mergeSortedArray(ArrayList<Integer> A, ArrayList<Integer> B
        if (A == null || A.isEmpty()) return B;
        if (B == null || B.isEmpty()) return A;

        ArrayList<Integer> C = new ArrayList<Integer>();
        int aLen = A.size(), bLen = B.size();
        int i = 0, j = 0;
        while (i < aLen && j < bLen) {
            if (A.get(i) < B.get(j)) {
                C.add(A.get(i));
                i++;
            } else {
                C.add(B.get(j));
                j++;
            }
        }

        // A has elements left
        while (i < aLen) {
            C.add(A.get(i));
            i++;
        }

        // B has elements left
        while (j < bLen) {
            C.add(B.get(j));
            j++;
        }

        return C;
    }
}
```

源码分析

分三步走，后面分别单独处理剩余的元素。

复杂度分析

遍历 A, B 数组各一次，时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

Challenge

两个倒排列表，一个特别大，一个特别小，如何 Merge？此时应该考虑用一个二分法插入小的，即内存拷贝。

Median

Source

- lintcode: (80) Median

```
Given a unsorted array with integers, find the median of it.

A median is the middle number of the array after it is sorted.

If there are even numbers in the array, return the N/2-th number after sorted.

Example
Given [4, 5, 1, 2, 3], return 3

Given [7, 9, 4, 5], return 5

Challenge
O(n) time.
```

题解

寻找未排序数组的中位数，简单粗暴的方法是先排序后输出中位数索引处的数，但是基于比较的排序算法的时间复杂度为 $O(n \log n)$, 不符合题目要求。线性时间复杂度的排序算法常见有计数排序、桶排序和基数排序，这三种排序方法的空间复杂度均较高，且依赖于输入数据特征（数据分布在有限的区间内），用在这里并不是比较好的解法。

由于这里仅需要找出中位数，即找出数组中前半个长度的较大的数，不需要进行完整的排序，说到这你是不是想到了快速排序了呢？快排的核心思想就是以基准为界将原数组划分为左小右大两个部分，用在这十分合适。快排的实现见 Quick Sort, 由于调用一次快排后基准元素的最终位置是知道的，故递归的终止条件即为当基准元素的位置(索引)满足中位数的条件时(左半部分长度为原数组长度一半)即返回最终结果。由于函数原型中左右最小索引并不总是原数组的最小最大，故需要引入相对位置(长度)也作为其中之一的参数。若左半部分长度偏大，则下一次递归排除右半部分，反之则排除左半部分。

Python

```python
class Solution:
    """
    @param nums: A list of integers.
    @return: An integer denotes the middle number of the array.
    """
    def median(self, nums):
        if not nums:
            return -1
        return self.helper(nums, 0, len(nums) - 1, (1 + len(nums)) / 2)

    def helper(self, nums, l, u, size):
        if l >= u:
```

```
            return nums[u]

        m = l
        for i in xrange(l + 1, u + 1):
            if nums[i] < nums[l]:
                m += 1
                nums[m], nums[i] = nums[i], nums[m]

        # swap between m and l after partition, important!
        nums[m], nums[l] = nums[l], nums[m]

        if m - l + 1 == size:
            return nums[m]
        elif m - l + 1 > size:
            return self.helper(nums, l, m - 1, size)
        else:
            return self.helper(nums, m + 1, u, size - (m - l + 1))
```

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers.
     * @return: An integer denotes the middle number of the array.
     */
    int median(vector<int> &nums) {
        if (nums.empty()) return 0;

        int len = nums.size();
        return helper(nums, 0, len - 1, (len + 1) / 2);
    }

private:
    int helper(vector<int> &nums, int l, int u, int size) {
        // if (l >= u) return nums[u];

        int m = l; // index m to track pivot
        for (int i = l + 1; i <= u; ++i) {
            if (nums[i] < nums[l]) {
                ++m;
                int temp = nums[i];
                nums[i] = nums[m];
                nums[m] = temp;
            }
        }

        // swap with the pivot
        int temp = nums[m];
        nums[m] = nums[l];
        nums[l] = temp;

        if (m - l + 1 == size) {
            return nums[m];
        } else if (m - l + 1 > size) {
            return helper(nums, l, m - 1, size);
        } else {
```

```
            return helper(nums, m + 1, u, size - (m - l + 1));
        }
    }
};
```

Java

```
public class Solution {
    /**
     * @param nums: A list of integers.
     * @return: An integer denotes the middle number of the array.
     */
    public int median(int[] nums) {
        if (nums == null) return -1;

        return helper(nums, 0, nums.length - 1, (nums.length + 1) / 2);
    }

    // l: lower, u: upper, m: median
    private int helper(int[] nums, int l, int u, int size) {
        if (l >= u) return nums[u];

        int m = l;
        for (int i = l + 1; i <= u; i++) {
            if (nums[i] < nums[l]) {
                m++;
                int temp = nums[m];
                nums[m] = nums[i];
                nums[i] = temp;
            }
        }
        // swap between array[m] and array[l]
        // put pivot in the mid
        int temp = nums[m];
        nums[m] = nums[l];
        nums[l] = temp;

        if (m - l + 1 == size) {
            return nums[m];
        } else if (m - l + 1 > size) {
            return helper(nums, l, m - 1, size);
        } else {
            return helper(nums, m + 1, u, size - (m - l + 1));
        }
    }
}
```

源码分析

以相对距离(长度)进行理解，递归终止步的条件一直保持不变(比较左半部分的长度)。

以题目中给出的样例进行分析，`size` 传入的值可为 `(len(nums) + 1) / 2`，终止条件为 `m - l + 1 == size`，含义为基准元素到索引为 `l` 的元素之间(左半部分)的长度(含)与 `(len(nums) + 1) / 2` 相等。若 `m - l + 1 > size`，即左半部分长度偏大，此时递归终止条件并未变化，因为 `l` 的值在下一次递归调用时并未

改变，所以仍保持为 `size`；若 `m - l + 1 < size`，左半部分长度偏小，下一次递归调用右半部分，由于此时左半部分的索引值已变化，故 `size` 应改为下一次在右半部分数组中的终止条件 `size - (m - l + 1)`，含义为原长度 `size` 减去左半部分数组的长度 `m - l + 1`.

复杂度分析

和快排类似，这里也有最好情况与最坏情况，平均情况下，索引 `m` 每次都处于中央位置，即每次递归后需要遍历的数组元素个数减半，故总的时间复杂度为 $O(n(1+1/2+1/4+...))=O(2n)$, 最坏情况下为平方。使用了临时变量，空间复杂度为 $O(1)$, 满足题目要求。

Partition Array by Odd and Even

Source

- lintcode: (373) Partition Array by Odd and Even
- Segregate Even and Odd numbers - GeeksforGeeks

```
Partition an integers array into odd number first and even number second.

Example
Given [1, 2, 3, 4], return [1, 3, 2, 4]

Challenge
Do it in-place.
```

题解

将数组中的奇数和偶数分开，使用『两根指针』的方法最为自然，奇数在前，偶数在后，若不然则交换之。

Java

```java
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: nothing
     */
    public void partitionArray(int[] nums) {
        if (nums == null) return;

        int left = 0, right = nums.length - 1;
        while (left < right) {
            // odd number
            while (left < right && nums[left] % 2 != 0) {
                left++;
            }
            // even number
            while (left < right && nums[right] % 2 == 0) {
                right--;
            }
            // swap
            if (left < right) {
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
            }
        }
    }
}
```

源码分析

注意处理好边界即循环时保证 `left < right`.

复杂度分析

遍历一次数组，时间复杂度为 $O(n)$, 使用了两根指针，空间复杂度 $O(1)$.

Kth Largest Element

Source

- leetcode: Kth Largest Element in an Array | LeetCode OJ
- lintcode: (5) Kth Largest Element

```
Find K-th largest element in an array.

Example
In array [9,3,2,4,8], the 3rd largest element is 4.

In array [1,2,3,4,5], the 1st largest element is 5,
2nd largest element is 4, 3rd largest element is 3 and etc.

Note
You can swap elements in the array

Challenge
O(n) time, O(1) extra memory.
```

题解

找第 K 大数，基于比较的排序的方法时间复杂度为 $O(n)$, 数组元素无区间限定，故无法使用线性排序。由于只是需要找第 K 大数，这种类型的题通常需要使用快排的思想解决。Quick Sort 总结了一些经典模板。这里比较基准值最后的位置的索引值和 K 的大小关系即可递归求解。

Java

```java
class Solution {
    //param k : description of k
    //param numbers : array of numbers
    //return: description of return
    public int kthLargestElement(int k, ArrayList<Integer> numbers) {
        if (numbers == null || numbers.isEmpty()) return -1;

        int result = qSort(numbers, 0, numbers.size() - 1, k);
        return result;
    }

    private int qSort(ArrayList<Integer> nums, int l, int u, int k) {
        // l should not greater than u
        if (l >= u) return nums.get(u);

        // index m of nums
        int m = l;
        for (int i = l + 1; i <= u; i++) {
            if (nums.get(i) > nums.get(l)) {
                m++;
```

```
                Collections.swap(nums, m, i);
            }
        }
        Collections.swap(nums, m, l);

        if (m + 1 == k) {
            return nums.get(m);
        } else if (m + 1 > k) {
            return qSort(nums, l, m - 1, k);
        } else {
            return qSort(nums, m + 1, u, k);
        }
    }
}
```

源码分析

递归的终止条件有两个，一个是左边界的值等于右边界(实际中其实不会有 l > u), 另一个则是索引值 `m + 1 == k`. 这里找的是第 K 大数，故为降序排列，for 循环中使用 `nums.get(i) > nums.get(l)` 而不是小于号。

复杂度分析

最坏情况下需要遍历 $n + n - 1 + ... + 1 = O(n^2)$, 平均情况下 $n + n/2 + n/4 + ... + 1 = O(2n) = O(n)$. 故平均情况时间复杂度为 $O(n)$. 交换数组的值时使用了额外空间，空间复杂度 $O(1)$.

Search - 搜索

本章主要总结二分搜索相关的题。

- 能使用二分搜索的前提是数组已排序。
- 二分查找的使用场景：（1）可转换为find the first/last position of...（2）时间复杂度至少为O(lgn)。
- 递归和迭代的使用场景：能用迭代就用迭代，特别复杂时采用递归。

Binary Search - 二分查找

Source

- lintcode: lintcode - (14) Binary Search

```
Binary search is a famous question in algorithm.

For a given sorted array (ascending order) and a target number, find the first index of t

If the target number does not exist in the array, return -1.

Example
If the array is [1, 2, 3, 3, 4, 5, 10], for given target 3, return 2.

Challenge
If the count of numbers is bigger than MAXINT, can your code work properly?
```

题解

对于已排序升序数组，使用二分查找可满足复杂度要求，注意数组中可能有重复值。

Java

```java
/**
 * 本代码fork自九章算法。没有版权欢迎转发。
 * http://www.jiuzhang.com//solutions/binary-search/
 */
class Solution {
    /**
     * @param nums: The integer array.
     * @param target: Target to find.
     * @return: The first position of target. Position starts from 0.
     */
    public int binarySearch(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0;
        int end = nums.length - 1;
        int mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2; // avoid overflow when (end + start)
            if (target < nums[mid]) {
                end = mid;
            } else if (target > nums[mid]) {
                start = mid;
            } else {
```

```
                end = mid;
            }
        }

        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
            return end;
        }

        return -1;
    }
}
```

源码分析

1. 首先对输入做异常处理，数组为空或者长度为0。
2. 初始化 `start, end, mid` 三个变量，注意mid的求值方法，可以防止两个整型值相加时溢出。
3. **使用迭代而不是递归**进行二分查找，因为工程中递归写法存在潜在溢出的可能。
4. while终止条件应为 `start + 1 < end` 而不是 `start <= end`， `start == end` 时可能出现死循环。**即循环终止条件是相邻或相交元素时退出。**
5. 迭代终止时target应为start或者end中的一个——由上述循环终止条件有两个，具体谁先谁后视题目是找 first position or last position 而定。
6. 赋值语句 `end = mid` 有两个条件是相同的，可以选择写到一块。
7. 配合while终止条件 `start + 1 < end` （相邻即退出）的赋值语句mid永远没有 `+1` 或者 `-1`，这样不会死循环。

Search Insert Position

Source

- lintcode: (60) Search Insert Position

```
Given a sorted array and a target value, return the index if the target is found. If not,

You may assume no duplicates in the array.

Example
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0
```

题解

应该把二分法的问题拆解为 `find the first/last position of...` 的问题。由最原始的二分查找可找到不小于目标整数的最小下标。返回此下标即可。

Java

```java
public class Solution {
    /**
     * param A : an integer sorted array
     * param target :  an integer to be inserted
     * return : an integer
     */
    public int searchInsert(int[] A, int target) {
        if (A == null) {
            return -1;
        }
        if (A.length == 0) {
            return 0;
        }

        int start = 0, end = A.length - 1;
        int mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (A[mid] == target) {
                return mid; // no duplicates, if not `end = target;`
            } else if (A[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
```

```
        }

        if (A[start] >= target) {
            return start;
        } else if (A[end] >= target) {
            return end; // in most cases
        } else {
            return end + 1; // A[end] < target;
        }
    }
}
```

源码分析

要注意例子中的第三个, [1,3,5,6], 7 → 4，即找不到要找的数字的情况，此时应返回数组长度，即代码中最后一个else的赋值语句 `return end + 1;`

Search for a Range

Source

- lintcode: (61) Search for a Range

```
Given a sorted array of integers, find the starting and ending position of a given target

Your algorithm's runtime complexity must be in the order of O(log n).

If the target is not found in the array, return [-1, -1].

Example
Given [5, 7, 7, 8, 8, 10] and target value 8,
return [3, 4].
```

题解

Search for a range 的题目可以拆解为找 first & last position 的题目，即要做两次二分。由上题二分查找可找到满足条件的左边界，因此只需要再将右边界找出即可。注意到在 `(target == nums[mid]` 时赋值语句为 `end = mid` ，将其改为 `start = mid` 即可找到右边界，解毕。

Java

```java
/**
 * 本代码fork自九章算法。没有版权欢迎转发。
 * http://www.jiuzhang.com/solutions/search-for-a-range/
 */
public class Solution {
    /**
     *@param A : an integer sorted array
     *@param target :  an integer to be inserted
     *return : a list of length 2, [index1, index2]
     */
    public ArrayList<Integer> searchRange(ArrayList<Integer> A, int target) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        int start, end, mid;
        result.add(-1);
        result.add(-1);

        if (A == null || A.size() == 0) {
            return result;
        }

        // search for left bound
        start = 0;
        end = A.size() - 1;
        while (start + 1 < end) {
```

```
            mid = start + (end - start) / 2;
            if (A.get(mid) == target) {
                end = mid; // set end = mid to find the minimum mid
            } else if (A.get(mid) > target) {
                end = mid;
            } else {
                start = mid;
            }
        }
        if (A.get(start) == target) {
            result.set(0, start);
        } else if (A.get(end) == target) {
            result.set(0, end);
        } else {
            return result;
        }

        // search for right bound
        start = 0;
        end = A.size() - 1;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (A.get(mid) == target) {
                start = mid; // set start = mid to find the maximum mid
            } else if (A.get(mid) > target) {
                end = mid;
            } else {
                start = mid;
            }
        }
        if (A.get(end) == target) {
            result.set(1, end);
        } else if (A.get(start) == target) {
            result.set(1, start);
        } else {
            return result;
        }

        return result;
        // write your code here
    }
}
```

源码分析

1. 首先对输入做异常处理，数组为空或者长度为0
2. 初始化 `start, end, mid` 三个变量，注意mid的求值方法，可以防止两个整型值相加时溢出
3. **使用迭代而不是递归**进行二分查找
4. while终止条件应为 `start + 1 < end` 而不是 `start <= end` ， `start == end` 时可能出现死循环
5. 先求左边界，迭代终止时先判断 `A.get(start) == target` ，再判断 `A.get(end) == target` ，因为迭代终止时target必取start或end中的一个，而end又大于start，取左边界即为start.
6. 再求右边界，迭代终止时先判断 `A.get(end) == target` ，再判断 `A.get(start) == target`
7. 两次二分查找除了终止条件不同，中间逻辑也不同，即当 `A.get(mid) == target` 如果是左边界（first postion），中间逻辑是 `end = mid` ；若是右边界（last position），中间逻辑是 `start = mid`

8. 两次二分查找中间勿忘记重置 `start, end` 的变量值。

First Bad Version

Source

- lintcode: (74) First Bad Version

```
The code base version is an integer and start from 1 to n. One day, someone commit a bad
You can determine whether a version is bad by the following interface:

Java:
    public VersionControl {
        boolean isBadVersion(int version);
    }
C++:
    class VersionControl {
    public:
        bool isBadVersion(int version);
    };
Python:
    class VersionControl:
        def isBadVersion(version)

Find the first bad version.
Note
You should call isBadVersion as few as possible.

Please read the annotation in code area to get the correct way to call isBadVersion in di

Example
Given n=5

Call isBadVersion(3), get false

Call isBadVersion(5), get true

Call isBadVersion(4), get true

return 4 is the first bad version

Challenge
Do not call isBadVersion exceed O(logn) times.
```

题 Search for a Range 的变形，找出左边界即可。

Java

```
/**
 * public class VersionControl {
 *     public static boolean isBadVersion(int k);
 * }
```

```
 * you can use VersionControl.isBadVersion(k) to judge wether
 * the kth code version is bad or not.
*/
class Solution {
    /**
     * @param n: An integers.
     * @return: An integer which is the first bad version.
     */
    public int findFirstBadVersion(int n) {
        // write your code here
        if (n == 0) {
            return -1;
        }

        int start = 1, end = n, mid;
        while (start + 1 < end) {
            mid = start + (end - start)/2;
            if (VersionControl.isBadVersion(mid) == false) {
                start = mid;
            } else {
                end = mid;
            }
        }

        if (VersionControl.isBadVersion(start) == true) {
            return start;
        } else if (VersionControl.isBadVersion(end) == true) {
            return end;
        } else {
            return -1; // not found
        }
    }
}
```

C++

```
/**
 * class VersionControl {
 *     public:
 *     static bool isBadVersion(int k);
 * }
 * you can use VersionControl::isBadVersion(k) to judge wether
 * the kth code version is bad or not.
*/
class Solution {
public:
    /**
     * @param n: An integers.
     * @return: An integer which is the first bad version.
     */
    int findFirstBadVersion(int n) {
        if (n < 1) {
            return -1;
        }

        int start = 1;
        int end = n;
```

```
        int mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (VersionControl::isBadVersion(mid)) {
                end = mid;
            } else {
                start = mid;
            }
        }

        if (VersionControl::isBadVersion(start)) {
            return start;
        } else if (VersionControl::isBadVersion(end)) {
            return end;
        }

        return -1;  // find no bad version
    }
};
```

源码分析

找左边界和Search for a Range类似，但是最好要考虑到有可能end处也为good version，此部分异常也可放在开始的时候处理。

Python

```
#class VersionControl:
#    @classmethod
#    def isBadVersion(cls, id)
#        # Run unit tests to check whether verison `id` is a bad version
#        # return true if unit tests passed else false.
# You can use VersionControl.isBadVersion(10) to check whether version 10 is a
# bad version.
class Solution:
    """
    @param n: An integers.
    @return: An integer which is the first bad version.
    """
    def findFirstBadVersion(self, n):
        if n < 1:
            return -1

        start, end = 1, n
        while start + 1 < end:
            mid = start + (end - start) / 2
            if VersionControl.isBadVersion(mid):
                end = mid
            else:
                start = mid

        if VersionControl.isBadVersion(start):
            return start
        elif VersionControl.isBadVersion(end):
            return end
```

```
        return -1
```

Search a 2D Matrix

Source

- lintcode: (28) Search a 2D Matrix

```
Write an efficient algorithm that searches for a value in an m x n matrix.

This matrix has the following properties:

    * Integers in each row are sorted from left to right.

    * The first integer of each row is greater than the last integer of the previous row.

Example
Consider the following matrix:

[

    [1, 3, 5, 7],

    [10, 11, 16, 20],

    [23, 30, 34, 50]

]

Given target = 3, return true.

Challenge
O(log(n) + log(m)) time
```

题解 - 一次二分搜索 V.S. 两次二分搜索

一次二分搜索

由于矩阵按升序排列，因此可将二维矩阵转换为一维问题。对原始的二分搜索进行适当改变即可(求行和列)。时间复杂度为 $O(log(mn)) = O(log(m) + log(n))$

两次二分搜索

先按行再按列进行搜索，即两次二分搜索。时间复杂度相同。

以一次二分搜素的方法为例。

Java

```
/**
```

```
 * 本代码由九章算法编辑提供。没有版权欢迎转发。
 * http://www.jiuzhang.com/solutions/search-a-2d-matrix
 */
// Binary Search Once
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0) {
            return false;
        }
        if (matrix[0] == null || matrix[0].length == 0) {
            return false;
        }

        int row = matrix.length, column = matrix[0].length;
        int start = 0, end = row * column - 1;
        int mid, number;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            number = matrix[mid / column][mid % column];
            if (number == target) {
                return true;
            } else if (number < target) {
                start = mid;
            } else {
                end = mid;
            }
        }

        if (matrix[start / column][start % column] == target) {
            return true;
        } else if (matrix[end / column][end % column] == target) {
            return true;
        }

        return false;
    }
}
```

源码分析

仍然可以使用经典的二分搜索模板，注意下标的赋值即可。

1. 首先对输入做异常处理，不仅要考虑到matrix为空串，还要考虑到matrix[0]也为空串。
2. 如果搜索结束时target与start或者end的值均不等时，则必在矩阵的值范围之外，避免了特殊情况的考虑。

第一次A掉这个题用的是分行分列两次搜索，好蠢...

Find Peak Element

Source

- leetcode: Find Peak Element | LeetCode OJ
- lintcode: (75) Find Peak Element

```
There is an integer array which has the following features:

    * The numbers in adjacent positions are different.
    * A[0] < A[1] && A[A.length - 2] > A[A.length - 1].

We define a position P is a peek if A[P] > A[P-1] && A[P] > A[P+1].

Find a peak element in this array. Return the index of the peak.

Note
The array may contains multiple peeks, find any of them.

Example
[1, 2, 1, 3, 4, 5, 7, 6]

return index 1 (which is number 2)  or 6 (which is number 7)

Challenge
Time complexity O(logN)
```

题解1 - lintcode

由时间复杂度的暗示可知应使用二分搜索。首先分析若使用传统的二分搜索，若 `A[mid] > A[mid - 1] && A[mid] < A[mid + 1]` ，则找到一个peak为A[mid]；若 `A[mid - 1] > A[mid]` ，则A[mid]左侧必定存在一个peak，可用反证法证明：若左侧不存在peak，则A[mid]左侧元素必满足 `A[0] > A[1] > ... > A[mid -1] > A[mid]` ，与已知 `A[0] < A[1]` 矛盾，证毕。同理可得若 `A[mid + 1] > A[mid]` ，则A[mid]右侧必定存在一个peak。如此迭代即可得解。

备注：如果本题是找 first/last peak，就不能用二分法了。

Python

```python
class Solution:
    #@param A: An integers list.
    #@return: return any of peek positions.
    def findPeak(self, A):
        if not A:
            return -1

        l, r = 0, len(A) - 1
        while l + 1 < r:
```

```
            mid = l + (r - l) / 2
            if A[mid] < A[mid - 1]:
                r = mid
            elif A[mid] < A[mid + 1]:
                l = mid
            else:
                return mid
        mid = l if A[l] > A[r] else r
        return mid
```

C++

```
class Solution {
public:
    /**
     * @param A: An integers array.
     * @return: return any of peek positions.
     */
    int findPeak(vector<int> A) {
        if (A.size() == 0) return -1;

        int l = 0, r = A.size() - 1;
        while (l + 1 < r) {
            int mid = l + (r - l) / 2;
            if (A[mid] < A[mid - 1]) {
                r = mid;
            } else if (A[mid] < A[mid + 1]) {
                l = mid;
            } else {
                return mid;
            }
        }

        int mid = A[l] > A[r] ? l : r;
        return mid;
    }
};
```

Java

```
class Solution {
    /**
     * @param A: An integers array.
     * @return: return any of peek positions.
     */
    public int findPeak(int[] A) {
        if (A == null || A.length == 0) return -1;

        int l = 0, r = A.length - 1;
        while (l + 1 < r) {
            int mid = l + (r - l) / 2;
            if (A[mid] < A[mid - 1]) {
                r = mid;
            } else if (A[mid] < A[mid + 1]) {
```

```
                l = mid;
            } else {
                return mid;
            }
        }

        int mid = A[l] > A[r] ? l : r;
        return mid;
    }
}
```

题解2 - leetcode

leetcode 上的题和 lintcode 上有细微的变化，题目如下：

```
A peak element is an element that is greater than its neighbors.

Given an input array where num[i] ≠ num[i+1],
find a peak element and return its index.

The array may contain multiple peaks,
in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

For example, in array [1, 2, 3, 1], 3 is a peak element and
your function should return the index number 2.

click to show spoilers.

Note:
Your solution should be in logarithmic complexity.
```

如果一开始做的是 leetcode 上的版本而不是 lintcode 上的话，这道题难度要大一些。有了以上的分析基础再来刷 leetcode 上的这道题就是小 case 了，注意题中的关键提示 `num[-1] = num[n] = -∞`，虽然不像 lintcode 上那么直接，但是稍微变通下也能想到。即 `num[-1] < num[0] && num[n-1] > num[n]`，那么问题来了，这样一来就不能确定峰值一定存在了，因为给定数组为单调序列的话就咩有峰值了，但是实际情况是——题中有负无穷的假设，也就是说在单调序列的情况下，峰值为数组首部或者尾部元素，谁大就是谁了。

Java

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if (nums == null || nums.length == 0) return -1;

        int l = 0, r = nums.length - 1;
        while (l + 1 < r) {
            mid = l + (r - l) / 2;
            if (nums[mid] < nums[mid - 1]) {
                // 1 peak at least in the left side
```

```
                r = mid;
            } else if (nums[mid] < nums[mid + 1]) {
                // 1 peak at least in the right side
                l = mid;
            } else {
                return mid;
            }
        }

        mid = nums[l] > nums[r] ? l : r;
        return mid;
    }
}
```

源码分析

典型的二分法模板应用，需要注意的是需要考虑单调序列的特殊情况。当然也可使用紧凑一点的实现如改写循环条件为 `l < r`，这样就不用考虑单调序列了，见实现2.

复杂度分析

二分法，时间复杂度 $O(\log n)$.

Java - compact implementation[leetcode_discussion]

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1, mid = end / 2;
        while (start < end) {
            if (nums[mid] < nums[mid + 1]) {
                // 1 peak at least in the right side
                start = mid + 1;
            } else {
                // 1 peak at least in the left side
                end = mid;
            }
            mid = start + (end - start) / 2;
        }

        return start;
    }
}
```

C++ 的代码可参考 Java 或者 @xuewei4d 的实现。

> leetcode 和 lintcode 上给的方法名不一样，leetcode 上的为 `findPeakElement` 而 lintcode 上为 `findPeak`，弄混的话会编译错误。

Reference

- [leetcode_discussion]. Java - Binary-Search Solution - Leetcode Discuss ↩

Search in Rotated Sorted Array

对于旋转数组的分析可使用画图的方法，如下图所示，升序数组经旋转后可能为如下两种形式。

Source

- lintcode: (62) Search in Rotated Sorted Array

```
Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise

You may assume no duplicate exists in the array.
Example
For [4, 5, 1, 2, 3] and target=1, return 2

For [4, 5,1, 2, 3] and target=0, return -1
```

题解

对于有序数组，使用二分搜索比较方便。分析题中的数组特点，旋转后初看是乱序数组，但仔细一看其实里面是存在两段有序数组的。因此该题可转化为如何找出旋转数组中的局部有序数组，并使用二分搜索解之。结合实际数组在纸上分析较为方便。

C++

```
/**
 * 本代码fork自
 * http://www.jiuzhang.com/solutions/search-in-rotated-sorted-array/
 */
class Solution {
    /**
     * param A : an integer ratated sorted array
     * param target :  an integer to be searched
```

```
     * return : an integer
     */
public:
    int search(vector<int> &A, int target) {
        if (A.empty()) {
            return -1;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (target == A[mid]) {
                return mid;
            }
            if (A[start] < A[mid]) {
                // situation 1, numbers between start and mid are sorted
                if (A[start] <= target && target < A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else {
                // situation 2, numbers between mid and end are sorted
                if (A[mid] < target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }

        if (A[start] == target) {
            return start;
        }
        if (A[end] == target) {
            return end;
        }
        return -1;
    }
};
```

源码分析

1. 若 `target == A[mid]` ，索引找到，直接返回
2. 寻找局部有序数组，分析 `A[mid]` 和两段有序的数组特点，由于旋转后前面有序数组最小值都比后面有序数组最大值大。故若 `A[start] < A[mid]` 成立，则start与mid间的元素必有序（要么是前一段有序数组，要么是后一段有序数组，还有可能是未旋转数组）。
3. 接着在有序数组 `A[start]~A[mid]` 间进行二分搜索，但能在 `A[start]~A[mid]` 间搜索的前提是 `A[start] <= target <= A[mid]` 。
4. 接着在有序数组 `A[mid]~A[end]` 间进行二分搜索，注意前提条件。
5. 搜索完毕时索引若不是mid或者未满足while循环条件，则测试A[start]或者A[end]是否满足条件。
6. 最后若未找到满足条件的索引，则返回-1.

Java

```
public class Solution {
    /**
     *@param A : an integer rotated sorted array
     *@param target :  an integer to be searched
     *return : an integer
     */
    public int search(int[] A, int target) {
        // write your code here
        if (A == null || A.length == 0) {
            return -1;
        }

        int start = 0, end = A.length - 1, mid = 0;
        while (start + 1 < end) {
            mid = start + (end - start)/2;
            if (A[mid] == target) {
                return mid;
            }
            if (A[start] < A[mid]) {//part 1
                if (A[start] <= target && target <= A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else { //part 2
                if (A[mid] <= target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        } // end while

        if (A[start] == target) {
            return start;
        } else if (A[end] == target) {
            return end;
        } else {
            return -1; // not found
        }
    }
}
```

Search in Rotated Sorted Array II

Source

- lintcode: (63) 搜索旋转排序数组 II

```
跟进“搜索旋转排序数组”，假如有重复元素又将如何？
```

```
是否会影响运行时间复杂度?

如何影响?

为何会影响?

写出一个函数判断给定的目标值是否出现在数组中。

样例
给出[3,4,4,5,7,0,1,2]和target=4，返回 true
```

题解

仔细分析此题和之前一题的不同之处，前一题我们利用 `A[start] < A[mid]` 这一关键信息，而在此题中由于有重复元素的存在，在 `A[start] == A[mid]` 时无法确定有序数组，此时只能依次递增start/递减end以缩小搜索范围，时间复杂度最差变为O(n)。

C++

```
class Solution {
    /**
     * param A : an integer ratated sorted array and duplicates are allowed
     * param target :  an integer to be search
     * return : a boolean
     */
public:
    bool search(vector<int> &A, int target) {
        if (A.empty()) {
            return false;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (target == A[mid]) {
                return true;
            }
            if (A[start] < A[mid]) {
                // situation 1, numbers between start and mid are sorted
                if (A[start] <= target && target < A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else if (A[start] > A[mid]) {
                // situation 2, numbers between mid and end are sorted
                if (A[mid] < target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
```

```
            } else  {
                // increment start
                ++start;
            }
        }

        if (A[start] == target || A[end] == target) {
            return true;
        }
        return false;
    }
};
```

源码分析

在 `A[start] == A[mid]` 时递增start序号即可。

Find Minimum in Rotated Sorted Array

Source

- lintcode: (159) Find Minimum in Rotated Sorted Array

```
Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

Example
Given [4,5,6,7,0,1,2] return 0
```

题解

如前节所述，对于旋转数组的分析可使用画图的方法，如下图所示，升序数组经旋转后可能为如下两种形式。

最小值可能在上图中的两种位置出现，如果仍然使用数组首部元素作为target去比较，则需要考虑图中右侧情况。**使用逆向思维分析，如果使用数组尾部元素分析，则无需图中右侧的特殊情况。**

C++

```
class Solution {
public:
    /**
     * @param num: a rotated sorted array
     * @return: the minimum number in the array
     */
    int findMin(vector<int> &num) {
        if (num.empty()) {
            return -1;
        }
```

```
        vector<int>::size_type start = 0;
        vector<int>::size_type end = num.size() - 1;
        vector<int>::size_type mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (num[mid] < num[end]) {
                end = mid;
            } else {
                start = mid;
            }
        }

        if (num[start] < num[end]) {
            return num[start];
        } else {
            return num[end];
        }
    }
};
```

源码分析

仅需注意使用 `num[end]` 作为判断依据即可，由于题中已给无重复数组的条件，故无需处理 `num[mid] == num[end]` 特殊条件。

Find Minimum in Rotated Sorted Array II

Source

- lintcode: (160) Find Minimum in Rotated Sorted Array II

题解

由于此题输入可能有重复元素，因此在 `num[mid] == num[end]` 时无法使用二分的方法缩小start或者end的取值范围。此时只能使用递增start/递减end逐步缩小范围。

C++

```
class Solution {
public:
    /**
     * @param num: a rotated sorted array
     * @return: the minimum number in the array
     */
    int findMin(vector<int> &num) {
        if (num.empty()) {
            return -1;
        }
```

```
        vector<int>::size_type start = 0;
        vector<int>::size_type end = num.size() - 1;
        vector<int>::size_type mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (num[mid] > num[end]) {
                start = mid;
            } else if (num[mid] < num[end]) {
                end = mid;
            } else {
                --end;
            }
        }

        if (num[start] < num[end]) {
            return num[start];
        } else {
            return num[end];
        }
    }
};
```

Search a 2D Matrix II

Source

- lintcode: (38) Search a 2D Matrix II

```
Write an efficient algorithm that searches for a value in an m x n matrix, return the occ

This matrix has the following properties:

    * Integers in each row are sorted from left to right.

    * Integers in each column are sorted from up to bottom.

    * No duplicate integers in each row or column.

Example
Consider the following matrix:

[

    [1, 3, 5, 7],

    [2, 4, 7, 8],

    [3, 5, 9, 10]

]

Given target = 3, return 2.

Challenge
O(m+n) time and O(1) extra space
```

题解 - 自右上而左下

1. 复杂度要求——O(m+n) time and O(1) extra space，同时输入只满足自顶向下和自左向右的升序，行与行之间不再有递增关系，与上题有较大区别。时间复杂度为线性要求，因此可从元素排列特点出发，从一端走向另一端无论如何都需要m+n步，因此可分析对角线元素。
2. 首先分析如果从左上角开始搜索，由于元素升序为自左向右和自上而下，因此如果target大于当前搜索元素时还有两个方向需要搜索，不太合适。
3. 如果从右上角开始搜索，由于左边的元素一定不大于当前元素，而下面的元素一定不小于当前元素，因此每次比较时均可排除一列或者一行元素（大于当前元素则排除当前行，小于当前元素则排除当前列，由矩阵特点可知），可达到题目要求的复杂度。

在遇到之前没有遇到过的复杂题目时，可先使用简单的数据进行测试去帮助发现规律。

C++

```
class Solution {
public:
    /**
     * @param matrix: A list of lists of integers
     * @param target: An integer you want to search in matrix
     * @return: An integer indicate the total occurrence of target in the given matrix
     */
    int searchMatrix(vector<vector<int> > &matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) {
            return 0;
        }

        const int ROW = matrix.size();
        const int COL = matrix[0].size();

        int row = 0, col = COL - 1;
        int occur = 0;
        while (row < ROW && col >= 0) {
            if (target == matrix[row][col]) {
                ++occur;
                --col;
            } else if (target < matrix[row][col]){
                --col;
            } else {
                ++row;
            }
        }

        return occur;
    }
};
```

Java

```
public class Solution {
    /**
     * @param matrix: A list of lists of integers
     * @param: A number you want to search in the matrix
     * @return: An integer indicate the occurrence of target in the given matrix
     */
    public int searchMatrix(int[][] matrix, int target) {
        int occurence = 0;

        if (matrix == null || matrix.length == 0) {
            return occurence;
        }
        if (matrix[0] == null || matrix[0].length == 0) {
            return occurence;
        }

        int row = matrix.length - 1;
        int column = matrix[0].length - 1;
        int index_row = 0, index_column = column;
        int number;

        if (target < matrix[0][0] || target > matrix[row][column]) {
```

```
            return occurence;
        }

        while (index_row < row + 1 && index_column + 1 > 0) {
            number = matrix[index_row][index_column];
            if (target == number) {
                occurence++;
                index_column--;
            } else if (target < number) {
                index_column--;
            } else if (target > number) {
                index_row++;
            }
        }

        return occurence;
    }
}
```

源码分析

1. 首先对输入做异常处理，不仅要考虑到matrix为空串，还要考虑到matrix[0]也为空串。
2. 注意循环终止条件。
3. 在找出`target`后应继续向左搜索其他可能相等的元素，下方比当前元素大，故排除此列。

Reference

Searching a 2D Sorted Matrix Part II | LeetCode

Median of two Sorted Arrays

Source

- lintcode: (65) Median of two Sorted Arrays

```
There are two sorted arrays A and B of size m and n respectively. Find the median of the

Example
For A = [1,2,3,4,5,6] B = [2,3,4,5], the median is 3.5

For A = [1,2,3] B = [4,5], the median is 3

Challenge
Time Complexity O(logn)
```

题解

何谓"Median"? 由题目意思可得即为两个数组中一半数据比它大，另一半数据比它小的那个数。详见 中位数 - 维基百科，自由的百科全书，题中已有信息两个数组均为有序，题目要求时间复杂度为O(log)，因此应该往二分法上想。

在两个数组中找第k大数->找中位数即为找第k大数的一个特殊情况——第(A.length + B.length) / 2 大数。因此首先需要解决找第k大数的问题。这个联想确实有点牵强...

使用归并的思想逐个比较找出中位数的复杂度为O(n)，显然不符要求，接下来考虑使用二分法。由于是找第k大数，使用二分法则比较A[k/2 - 1]和B[k/2 - 1]，并思考这两个元素和第k大元素的关系。

1. A[k/2 - 1] <= B[k/2 - 1] => A和B合并后的第k大数中必包含A[0]~A[k/2 -1]，可使用归并的思想去理解。
2. 若k/2 - 1超出A的长度，则必取B[0]~B[k/2 - 1]

C++

```cpp
class Solution {
public:
    /**
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    double findMedianSortedArrays(vector<int> A, vector<int> B) {
        if (A.empty() && B.empty()) {
            return 0;
        }

        vector<int> NonEmpty;
        if (A.empty()) {
```

```
            NonEmpty = B;
        }
        if (B.empty()) {
            NonEmpty = A;
        }
        if (!NonEmpty.empty()) {
            vector<int>::size_type len_vec = NonEmpty.size();
            return len_vec % 2 == 0 ?
                    (NonEmpty[len_vec / 2 - 1] + NonEmpty[len_vec / 2]) / 2.0 :
                    NonEmpty[len_vec / 2];
        }

        vector<int>::size_type len = A.size() + B.size();
        if (len % 2 == 0) {
            return ((findKth(A, 0, B, 0, len / 2) + findKth(A, 0, B, 0, len / 2 + 1)) / 2
        } else {
            return findKth(A, 0, B, 0, len / 2 + 1);
        }
        // write your code here
    }

private:
    int findKth(vector<int> &A, vector<int>::size_type A_start, vector<int> &B, vector<in
        if (A_start > A.size() - 1) {
            // all of the element of A are smaller than the kTh number
            return B[B_start + k - 1];
        }
        if (B_start > B.size() - 1) {
            // all of the element of B are smaller than the kTh number
            return A[A_start + k - 1];
        }

        if (k == 1) {
            return A[A_start] < B[B_start] ? A[A_start] : B[B_start];
        }

        int A_key = A_start + k / 2 - 1 < A.size() ?
                    A[A_start + k / 2 - 1] : INT_MAX;
        int B_key = B_start + k / 2 - 1 < B.size() ?
                    B[B_start + k / 2 - 1] : INT_MAX;

        if (A_key > B_key) {
            return findKth(A, A_start, B, B_start + k / 2, k - k / 2);
        } else {
            return findKth(A, A_start + k / 2, B, B_start, k - k / 2);
        }
    }
};
```

源码分析

此题的边界条件较多，不容易直接从代码看清思路。首先分析找k大的辅助程序。

1. 如果 `A_start > A.size() - 1` ，意味着A中无数提供，故仅能从B中取，所以只能是B从 `B_start` 开始的第k个数。下面的B...分析方法类似。

2. k为1时，无需再递归调用，直接返回较小值。
3. 以A为例，取出自 `A_start` 开始的第 `k / 2` 个数，若下标 `A_start + k / 2 - 1 < A.size()` ，则可取此下标对应的元素，否则置为int的最大值，便于后面进行比较，免去了诸多边界条件的判断。
4. 比较 `A_key > B_key` ，取小的折半递归调用findKth。

接下来分析 `findMedianSortedArrays` ：

1. 首先考虑异常情况，A, B都为空，A/B其中一个为空。
2. A+B 的长度为偶数时返回len / 2和 len / 2 + 1的均值，为奇数时则返回len / 2 + 1

Java

```
class Solution {
    /**
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    public double findMedianSortedArrays(int[] A, int[] B) {
        // write your code here
        if (A.length == 0 && B.length == 0) {
            return 0;
        }
        int len = A.length + B.length;
        if (len % 2 == 0) {
            return (findKth(A, 0, B, 0, len/2) + findKth(A, 0, B, 0, len/2+1)) / 2.0;
        } else {
            return findKth(A, 0, B, 0, len/2 + 1);
        }
    }

    //find kth number of two sorted array
    public static int findKth(int[] A, int A_start, int[] B, int B_start, int k) {
        if (A_start >= A.length) {
            return B[B_start + k - 1];
        }
        if (B_start >= B.length) {
            return A[A_start + k - 1];
        }
        if (k == 1) {
            return Math.min(A[A_start], B[B_start]);
        }

        int A_key = (A_start + k/2 - 1 < A.length) // if one array is too short
                    ? A[A_start + k/2 - 1] : Integer.MAX_VALUE; // trick
        int B_key = (B_start + k/2 - 1 < B.length) // if one array is too short
                    ? B[B_start + k/2 - 1] : Integer.MAX_VALUE; // trick

        if (A_key < B_key) {
            return findKth(A, A_start + k/2, B, B_start, k - k/2);
        } else {
            return findKth(A, A_start, B, B_start + k/2, k - k/2);
        }
    }
}
```

源码分析

1. 本题用非递归的方法非常麻烦，递归的方法减少了很多边界的判断。
2. 递归的条件比较重要，可以用极端情况时参数的状况来入手，即看`[]A, []B, k`谁先达到极端情况。
3. 本解法中有一个小技巧，就是当`[]A, []B`中某一个数组太短了，无法取k/2，则返回无穷大，设置了Integer.MAX_VALUE。

reference

- 九章算法 | Median of Two Sorted Arrays
- LeetCode: Median of Two Sorted Arrays 解题报告 - Yu's Garden - 博客园

Sqrt x

Source

- leetcode: Sqrt(x) | LeetCode OJ
- lintcode: (141) Sqrt(x)

题解 - 二分搜索

由于只需要求整数部分，故对于任意正整数 x, 设其整数部分为 k, 显然有 $1 \leq k \leq x$, 求解 k 的值也就转化为了在有序数组中查找满足某种约束条件的元素，显然二分搜索是解决此类问题的良方。

Python

```
class Solution:
    # @param {integer} x
    # @return {integer}
    def mySqrt(self, x):
        if x < 0:
            return -1
        elif x == 0:
            return 0

        start, end = 1, x
        while start + 1 < end:
            mid = start + (end - start) / 2
            if mid**2 == x:
                return mid
            elif mid**2 > x:
                end = mid
            else:
                start = mid

        return start
```

源码分析

1. 异常检测，先处理小于等于0的值。
2. 使用二分搜索的经典模板，注意不能使用 `start < end` , 否则在给定值1时产生死循环。
3. 最后返回平方根的整数部分 `start` .

二分搜索过程很好理解，关键是最后的返回结果还需不需要判断？比如是取 start, end, 还是 mid? 我们首先来分析下二分搜索的循环条件，由 `while` 循环条件 `start + 1 < end` 可知， `start` 和 `end` 只可能有两种关系，一个是 `end == 1 || end ==2` 这一特殊情况，返回值均为1，另一个就是循环终止时 `start` 恰好在 `end` 前一个元素。设值 x 的整数部分为 k, 那么在执行二分搜索的过程中 $start \leq k \leq end$ 关系一直存在，也就是说在没有找到 $mid^2 == x$ 时，循环退出时有 $start < k < end$, 取整的话显然就

是 `start` 了。

复杂度分析

经典的二分搜索，时间复杂度为 $O(\log n)$, 使用了 `start` , `end` , `mid` 变量，空间复杂度为 $O(1)$.

除了使用二分法求平方根近似解之外，还可使用牛顿迭代法进一步提高运算效率，欲知后事如何，请猛戳求平方根sqrt()函数的底层算法效率问题 -- 简明现代魔法，不得不感叹算法的魔力！

Wood Cut

Source

- lintcode: (183) Wood Cut

```
Given n pieces of wood with length L[i] (integer array).
Cut them into small pieces to guarantee you could have equal or more than k pieces with t
What is the longest length you can get from the n pieces of wood?
Given L & k, return the maximum length of the small pieces.

Example
For L=[232, 124, 456], k=7, return 114.

Note
You couldn't cut wood into float length.

Challenge
O(n log Len), where Len is the longest length of the wood.
```

题解 - 二分搜索

这道题要直接想到二分搜索其实不容易，但是看到题中 Challenge 的提示后你大概就能想到往二分搜索上靠了。首先来分析下题意，题目意思是说给出 n 段木材 `L[i]`，将这 n 段木材切分为至少 k 段，这 k 段等长，求能从 n 段原材料中获得的最长单段木材长度。以 k=7 为例，要将 L 中的原材料分为7段，能得到的最大单段长度为114, 232/114 = 2, 124/114 = 1, 456/114 = 4, 2 + 1 + 4 = 7.

理清题意后我们就来想想如何用算法的形式表示出来，显然在计算如 `2`，`1`，`4` 等分片数时我们进行了取整运算，在计算机中则可以使用下式表示： $\sum_{i=1}^{n} \frac{L[i]}{l} \geq k$

其中 l 为单段最大长度，显然有 $1 \leq l \leq max(L[i])$. 单段长度最小为1，最大不可能超过给定原材料中的最大木材长度。

> 注意求和与取整的顺序，是先求 `L[i]/l` 的单个值，而不是先对 `L[i]` 求和。

分析到这里就和题 Sqrt x 差不多一样了，要求的是 l 的最大可能取值，同时 l 可以看做是从有序序列 `[1, max(L[i])]` 的一个元素，典型的二分搜索！

Python

```python
class Solution:
    """
    @param L: Given n pieces of wood with length L[i]
    @param k: An integer
    return: The maximum length of the small pieces.
    """
```

```
    def woodCut(self, L, k):
        if sum(L) < k:
            return 0

        max_len = max(L)
        start, end = 1, max_len
        while start + 1 < end:
            mid = start + (end - start) / 2
            pieces_sum = sum([len_i / mid for len_i in L])
            if pieces_sum < k:
                end = mid
            else:
                start = mid

        # corner case
        if end == 2 and sum([len_i / 2 for len_i in L]) >= k:
            return 2

        return start
```

源码分析

1. 异常处理，若对 L 求和所得长度都小于 k，那么肯定无解。
2. 初始化 `start` 和 `end` , 使用二分搜索。
3. 使用 list comprehension 求 $\sum_{i=1}^{n} \frac{L[i]}{l}$.
4. 若求得的 `pieces_sum` 小于 k，则说明 `mid` 偏大，下一次循环应缩小 `mid` ，对应为将当前 `mid` 赋给 `end` .
5. 与一般的二分搜索不同，即使有 `pieces_sum == k` 也不应立即返回 `mid` , 因为这里使用了取整运算，满足 `pieces_sum == k` 的值不止一个，应取其中最大的 `mid` , 具体实现中可以将 `pieces_sum < k` 写在前面，大于等于的情况直接用 `start = end` 代替。
6. 排除 `end == 2` 之后返回 `start` 即可。

简单对第6条做一些说明，首先需要进行二分搜索的前提是 `sum(L) >= k` 且 `end` 不满足 `end == 1 || end == 2` , `end` 为2时单独考虑即可。

复杂度分析

遍历求和时间复杂度为 $O(n)$, 二分搜索时间复杂度为 $O(\log max(L))$. 故总的时间复杂度为 $O(n \log max(L))$. 空间复杂度 $O(1)$.

Reference

- Wood Cut | 九章算法

Bit Manipulation

位运算的题大多较为灵活，涉及较多的按位与/或/异或等特性。

Reference

- 位运算简介及实用技巧（一）：基础篇 | Matrix67: The Aha Moments
- *cc150* chapter 8.5 and chapter 9.5

Single Number

「找单数」系列题，技巧性较强，需要灵活运用位运算的特性。

Source

- lintcode: (82) Single Number

```
Given 2*n + 1 numbers, every numbers occurs twice except one, find it.

Example
Given [1,2,2,1,3,4,3], return 4

Challenge
One-pass, constant extra space
```

题解

根据题意，共有 `2*n + 1` 个数，且有且仅有一个数落单，要找出相应的「单数」。鉴于有空间复杂度的要求，不可能使用另外一个数组来保存每个数出现的次数，考虑到异或运算的特性，根据 `x ^ x = 0` 和 `x ^ 0 = x` 可将给定数组的所有数依次异或，最后保留的即为结果。

C++

```cpp
class Solution {
public:
    /**
     * @param A: Array of integers.
     * return: The single number.
     */
    int singleNumber(vector<int> &A) {
        if (A.empty()) {
            return -1;
        }
        int result = 0;

        for (vector<int>::iterator iter = A.begin(); iter != A.end(); ++iter) {
            result = result ^ *iter;
        }

        return result;
    }
};
```

源码分析

1. 异常处理(OJ上对于空vector的期望结果为0，但个人认为-1更为合理)

2. 初始化返回结果 `result` 为0，因为 `x ^ 0 = x`

Single Number II

Source

- lintcode: (83) Single Number II

```
Given 3*n + 1 numbers, every numbers occurs triple times except one, find it.

Example
Given [1,1,2,3,3,3,2,2,4,1] return 4

Challenge
One-pass, constant exstra space
```

题解 - 逐位处理

上题 Single Number 用到了二进制中异或的运算特性，这题给出的元素数目为 `3*n + 1`，因此我们很自然地想到如果有种运算能满足「三三运算」为0该有多好！对于三个相同的数来说，其相加的和必然是3的倍数，仅仅使用这一个特性还不足以将单数找出来，我们再来挖掘隐含的信息。以3为例，若使用不进位加法，三个3相加的结果为：

```
0011
0011
0011
----
0033
```

注意到其中的奥义了么？三个相同的数相加，不仅其和能被3整除，其二进制位上的每一位也能被3整除！因此我们只需要一个和 `int` 类型相同大小的数组记录每一位累加的结果即可。时间复杂度约为 $O((3n+1)\cdot sizeof(int)\cdot 8)$

C++ bit by bit

```
class Solution {
public:
    /**
     * @param A : An integer array
     * @return : An integer
     */
    int singleNumberII(vector<int> &A) {
        if (A.empty()) {
            return 0;
        }

        int result = 0, bit_i_sum = 0;

        for (int i = 0; i != 8 * sizeof(int); ++i) {
```

```
            bit_i_sum = 0;
            for (int j = 0; j != A.size(); ++j) {
                // get the *i*th bit of A
                bit_i_sum += ((A[j] >> i) & 1);
            }
            // set the *i*th bit of result
            result |= ((bit_i_sum % 3) << i);
        }

        return result;
    }
};
```

源码解析

1. 异常处理
2. 循环处理返回结果 `result` 的 `int` 类型的每一位，要么自增1，要么保持原值。注意 `i` 最大可取 $8 \cdot sizeof(int) - 1$, 字节数=>位数的转换
3. 对第 `i` 位处理完的结果模3后更新 `result` 的第 `i` 位，由于 `result` 初始化为0，故使用或操作即可完成

Reference

Single Number II - Leetcode Discuss 中抛出了这么一道扩展题：

```
Given an array of integers, every element appears k times except for one. Find that singl
```

@ranmocy 给出了如下经典解：

We need a array `x[i]` with size `k` for saving the bits appears `i` times. For every input number a, generate the new counter by `x[j] = (x[j-1] & a) | (x[j] & ~a)`. Except `x[0] = (x[k] & a) | (x[0] & ~a)`.

In the equation, the first part indicates the the carries from previous one. The second part indicates the bits not carried to next one.

Then the algorithms run in `O(kn)` and the extra space `O(k)`.

Java

```
public class Solution {
    public int singleNumber(int[] A, int k, int l) {
        if (A == null) return 0;
        int t;
        int[] x = new int[k];
        x[0] = ~0;
        for (int i = 0; i < A.length; i++) {
            t = x[k-1];
```

```
            for (int j = k-1; j > 0; j--) {
                x[j] = (x[j-1] & A[i]) | (x[j] & ~A[i]);
            }
            x[0] = (t & A[i]) | (x[0] & ~A[i]);
        }
        return x[l];
    }
}
```

Single Number III

Source

- lintcode: (84) Single Number III

```
Given 2*n + 2 numbers, every numbers occurs twice except two, find them.

Example
Given [1,2,2,3,4,4,5,3] return 1 and 5

Challenge
O(n) time, O(1) extra space.
```

题解

题 Single Number 的 follow up, 不妨设最后两个只出现一次的数分别为 `x1, x2`. 那么遍历数组时根据两两异或的方法可得最后的结果为 `x1 ^ x2`, 如果我们要分别求得 `x1` 和 `x2`, 我们可以根据 `x1 ^ x2 ^ x1 = x2` 求得 `x2`, 同理可得 `x_1`. 那么问题来了，如何得到 `x1` 和 `x2` 呢？看起来似乎是个死循环。大多数人一般也就能想到这一步(比如我...)。

这道题的巧妙之处在于利用 `x1 ^ x2` 的结果对原数组进行了分组，进而将 `x1` 和 `x2` 分开了。具体方法则是利用了 `x1 ^ x2` 不为0的特性，如果 `x1 ^ x2` 不为0，那么 `x1 ^ x2` 的结果必然存在某一二进制位不为0（即为1），我们不妨将最低位的1提取出来，由于在这一二进制位上 `x1` 和 `x2` 必然相异，即 `x1`, `x2` 中相应位一个为0，另一个为1，所以我们可以利用这个最低位的1将 `x1` 和 `x2` 分开。又由于除了 `x1` 和 `x2` 之外其他数都是成对出现，故与最低位的1异或时一定会抵消，十分之精妙！

Java

```java
public class Solution {
    /**
     * @param A : An integer array
     * @return : Two integers
     */
    public List<Integer> singleNumberIII(int[] A) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        if (A == null || A.length == 0) return nums;

        int x1xorx2 = 0;
        for (int i : A) {
            x1xorx2 ^= i;
        }

        // get the last 1 bit of x1xorx2, e.g. 1010 ==> 0010
        int last1Bit = x1xorx2 - (x1xorx2 & (x1xorx2 - 1));
        int single1 = 0, single2 = 0;
        for (int i : A) {
```

```
            if ((last1Bit & i) == 0) {
                single1 ^= i;
            } else {
                single2 ^= i;
            }
        }

        nums.add(single1);
        nums.add(single2);
        return nums;
    }
}
```

源码分析

求一个数二进制位1的最低位方法为 `x1xorx2 - (x1xorx2 & (x1xorx2 - 1))`, 其他位运算的总结可参考 Bit Manipulation。利用 `last1Bit` 可将数组的数分为两组，一组是相应位为0，另一组是相应位为1.

复杂度分析

两次遍历数组，时间复杂度 $O(n)$, 使用了部分额外空间，空间复杂度 $O(1)$.

Reference

- Single Number III 参考程序 Java/C++/Python

O(1) Check Power of 2

Source

- lintcode: (142) O(1) Check Power of 2

```
Using O(1) time to check whether an integer n is a power of 2.

Example
For n=4, return true;

For n=5, return false;

Challenge
O(1) time
```

题解

咋看起来挺简单的一道题目，可之前若是没有接触过神奇的位运算技巧遇到这种题就有点不知从哪入手了，咳咳，我第一次接触到这个题就是在七牛的笔试题中看到的，泪奔 :-(

简单点来考虑可以连除2求余，看最后的余数是否为1，但是这种方法无法在 $O(1)$ 的时间内解出，所以我们必须要想点别的办法了。2的整数幂若用二进制来表示，则其中必只有一个1，其余全是0，那么怎么才能用一个式子把这种特殊的关系表示出来了？传统的位运算如按位与、按位或和按位异或等均无法直接求解，我就不卖关子了，比较下 `x - 1` 和 `x` 的关系试试？以 `x=4` 为例。

```
0100 ==> 4
0011 ==> 3
```

两个数进行按位与就为0了！如果不是2的整数幂则无上述关系，反证法可证之。

Python

```python
class Solution:
    """
    @param n: An integer
    @return: True or false
    """
    def checkPowerOf2(self, n):
        if n < 1:
            return False
        else:
            return (n & (n - 1)) == 0
```

C++

```
class Solution {
public:
    /*
     * @param n: An integer
     * @return: True or false
     */
    bool checkPowerOf2(int n) {
        if (1 > n) {
            return false;
        } else {
            return 0 == (n & (n - 1));
        }
    }
};
```

Java

```
class Solution {
    /*
     * @param n: An integer
     * @return: True or false
     */
    public boolean checkPowerOf2(int n) {
        if (n < 1) {
            return false;
        } else {
            return (n & (n - 1)) == 0;
        }
    }
};
```

源码分析

除了考虑正整数之外，其他边界条件如小于等于0的整数也应考虑在内。在比较0和 `(n & (n - 1))` 的值时，需要用括号括起来避免优先级结合的问题。

复杂度分析

$O(1)$.

扩展

关于2的整数幂还有一道有意思的题，比如 Next Power of 2 - GeeksforGeeks，有兴趣的可以去围观下。

Convert Integer A to Integer B

Source

- CC150, lintcode: (181) Convert Integer A to Integer B

```
Determine the number of bits required to convert integer A to integer B

Example
Given n = 31, m = 14,return 2

(31)10=(11111)2

(14)10=(01110)2
```

题解

比较两个数不同的比特位个数，显然容易想到可以使用异或处理两个整数，相同的位上为0，不同的位上为1，故接下来只需将异或后1的个数求出即可。容易想到的方法是移位后和1按位与得到最低位的结果，使用计数器记录这一结果，直至最后操作数为0时返回最终值。这种方法需要遍历元素的每一位，有咩有更为高效的做法呢？还记得之前做过的 O1 Check Power of 2 吗？`x & (x - 1)` 既然可以检查2的整数次幂，那么如何才能进一步得到所有1的个数呢？——将异或得到的数分拆为若干个2的整数次幂，计算得到有多少个2的整数次幂即可。

以上的分析过程对于正数来说是毫无问题的，但问题就在于如果出现了负数如何破？不确定的时候就来个实例测测看，以-2为例，(-2) & (-2 - 1)的计算如下所示(简单起见这里以8位为准)：

```
 11111110 <==> -2   -2 <==> 11111110
+                          &
 11111111 <==> -1   -3 <==> 11111101
=                          =
 11111101                   11111100
```

细心的你也许发现了对于负数来说，其表现也是我们需要的——`x & (x - 1)` 的含义即为将二进制比特位的值为1的最低位置零。逐步迭代直至最终值为0时返回。

C/C++ 和 Java 中左溢出时会直接将高位丢弃，正好方便了我们的计算，但是在 Python 中就没这么幸运了，因为溢出时会自动转换类型，Orz... 所以使用 Python 时需要对负数专门处理，转换为求其补数中0的个数。

Python

```python
class Solution:
    """
    @param a, b: Two integer
```

```
    return: An integer
    """
    def bitSwapRequired(self, a, b):
        count = 0
        a_xor_b = a ^ b
        neg_flag = False
        if a_xor_b < 0:
            a_xor_b = abs(a_xor_b) - 1
            neg_flag = True
        while a_xor_b > 0:
            count += 1
            a_xor_b &= (a_xor_b - 1)

        # bit_wise = 32
        if neg_flag:
            count = 32 - count
        return count
```

C++

```
class Solution {
public:
    /**
     *@param a, b: Two integer
     *return: An integer
     */
    int bitSwapRequired(int a, int b) {
        int count = 0;
        int a_xor_b = a ^ b;
        while (a_xor_b != 0) {
            ++count;
            a_xor_b &= (a_xor_b - 1);
        }

        return count;
    }
};
```

Java

```
class Solution {
    /**
     *@param a, b: Two integer
     *return: An integer
     */
    public static int bitSwapRequired(int a, int b) {
        int count = 0;
        int a_xor_b = a ^ b;
        while (a_xor_b != 0) {
            ++count;
            a_xor_b &= (a_xor_b - 1);
        }

        return count;
```

```
    }
};
```

源码分析

Python 中 int 溢出时会自动变为 long 类型，故处理负数时需要求补数中0的个数，间接求得原异或得到的数中1的个数。

考虑到负数的可能，C/C++, Java 中循环终止条件为 `a_xor_b != 0`，而不是 `a_xor_b > 0`.

复杂度分析

取决于异或后数中1的个数，`O(max(ones in a ^ b))`.

关于 Python 中位运算的一些坑总结在参考链接中。

Reference

- BitManipulation - Python Wiki
- 5. Expressions — Python 2.7.10rc0 documentation
- Python之位移操作符所带来的困惑 - 旁观者 - 博客园

Factorial Trailing Zeroes

Source

- leetcode: Factorial Trailing Zeroes | LeetCode OJ
- lintcode: (2) Trailing Zeros

```
Write an algorithm which computes the number of trailing zeros in n factorial.

Example
11! = 39916800, so the out should be 2

Challenge
O(log N) time
```

题解1 - Iterative

找阶乘数中末尾的连零数量，容易想到的是找相乘能为10的整数倍的数，如 $2 \times 5, 1 \times 10$ 等，遥想当初做阿里笔试题时遇到过类似的题，当时想着算算5和10的个数就好了，可万万没想到啊，25可以变为两个5相乘！真是蠢死了... 根据数论里面的知识，任何正整数都可以表示为它的质因数的乘积[wikipedia]。所以比较准确的思路应该是计算质因数5和2的个数，取小的即可。质因数2的个数显然要大于5的个数，故只需要计算给定阶乘数中质因数中5的个数即可。原题的问题即转化为求阶乘数中质因数5的个数，首先可以试着分析下100以内的数，再试试100以上的数，聪明的你一定想到了可以使用求余求模等方法 :)

Python

```python
class Solution:
    # @param {integer} n
    # @return {integer}
    def trailingZeroes(self, n):
        if n < 0:
            return -1

        count = 0
        while n > 0:
            n /= 5
            count += n

        return count
```

C++

```cpp
class Solution {
public:
    int trailingZeroes(int n) {
        if (n < 0) {
```

```
            return -1;
        }

        int count = 0;
        for (; n > 0; n /= 5) {
            count += (n / 5);
        }

        return count;
    }
};
```

Java

```
public class Solution {
    public int trailingZeroes(int n) {
        if (n < 0) {
            return -1;
        }

        int count = 0;
        for (; n > 0; n /= 5) {
            count += (n / 5);
        }

        return count;
    }
}
```

源码分析

1. 异常处理，小于0的数返回-1.
2. 先计算5的正整数幂都有哪些，不断使用 n / 5 即可知质因数5的个数。
3. 在循环时使用 `n /= 5` 而不是 `i *= 5`，可有效防止溢出。

> lintcode 和 leetcode 上的方法名不一样，在两个 OJ 上分别提交的时候稍微注意下。

复杂度分析

关键在于 `n /= 5` 执行的次数，时间复杂度 $\log_5 n$，使用了 `count` 作为返回值，空间复杂度 $O(1)$.

题解2 - Recursive

可以使用迭代处理的程序往往用递归，而且往往更为优雅。递归的终止条件为 `n <= 0`.

Python

```
class Solution:
    # @param {integer} n
```

```
    # @return {integer}
    def trailingZeroes(self, n):
        if n == 0:
            return 0
        elif n < 0:
            return -1
        else:
            return n / 5 + self.trailingZeroes(n / 5)
```

C++

```
class Solution {
public:
    int trailingZeroes(int n) {
        if (n == 0) {
            return 0;
        } else if (n < 0) {
            return -1;
        } else {
            return n / 5 + trailingZeroes(n / 5);
        }
    }
};
```

Java

```
public class Solution {
    public int trailingZeroes(int n) {
        if (n == 0) {
            return 0;
        } else if (n < 0) {
            return -1;
        } else {
            return n / 5 + trailingZeroes(n / 5);
        }
    }
}
```

源码分析

这里将负数输入视为异常，返回-1而不是0. 注意使用递归时务必注意收敛和终止条件的返回值。这里递归层数最多不超过 $\log_5 n$, 因此效率还是比较高的。

复杂度分析

递归层数最大为 $\log_5 n$, 返回值均在栈上，可以认为没有使用辅助的堆空间。

Reference

- > wikipedia. Prime factor - Wikipedia, the free encyclopedia ↩
- Count trailing zeroes in factorial of a number - GeeksforGeeks

Unique Binary Search Trees

Source

- leetcode: Unique Binary Search Trees | LeetCode OJ
- lintcode: (163) Unique Binary Search Trees

```
Given n, how many structurally unique BSTs (binary search trees)
that store values 1...n?

Example
Given n = 3, there are a total of 5 unique BST's.

1           3    3       2      1
 \         /    /       / \      \
  3      2     1       1   3      2
 /      /       \                  \
2     1          2                  3
```

题解1 - 两重循环

挺有意思的一道题，与数据结构和动态规划都有点关系。这两天在骑车路上和睡前都一直在想，始终未能找到非常明朗的突破口，直到看到这么一句话——『以i为根节点的树，其左子树由[0, i-1]构成， 其右子树由[i+1, n]构成。』这不就是 BST 的定义嘛！灵活运用下就能找到递推关系了。

容易想到这道题的动态规划状态为 count[n], count[n] 表示到正整数 i 为止的二叉搜索树个数。容易得到 count[1] = 1, 根节点为1，count[2] = 2, 根节点可为1或者2。那么 count[3] 的根节点自然可为1，2，3. 如果以1为根节点，那么根据 BST 的定义，2和3只可能位于根节点1的右边；如果以2为根节点，则1位于左子树，3位于右子树；如果以3为根节点，则1和2必位于3的左子树。

抽象一下，如果以 i 作为根节点，由基本的排列组合知识可知，其唯一 BST 个数为左子树的 BST 个数乘上右子树的 BST 个数。故对于 i 来说，其左子树由[0, i - 1]构成，唯一的 BST 个数为 count[i - 1], 右子树由[i + 1, n] 构成，其唯一的 BST 个数没有左子树直观，但是也有迹可循。对于两组有序数列「1, 2, 3] 和 [4, 5, 6]来说，**这两个有序数列分别组成的 BST 个数必然是一样的，因为 BST 的个数只与有序序列的大小有关，而与具体值没有关系。**所以右子树的 BST 个数为 count[n - i]，于是乎就得到了如下递推关系：

$$count[i] = \sum_{j=0}^{i-1}(count[j] \cdot count[i-j-1])$$

网上有很多用 count[3] 的例子来得到递推关系，恕本人愚笨，在没有从 BST 的定义和有序序列个数与 BST 关系分析的基础上，我是不敢轻易说就能得到如上状态转移关系的。

Python

```python
class Solution:
    # @paramn n: An integer
    # @return: An integer
    def numTrees(self, n):
```

```
        if n < 0:
            return -1

        count = [0] * (n + 1)
        count[0] = 1
        for i in xrange(1, n + 1):
            for j in xrange(i):
                count[i] += count[j] * count[i - j - 1]

        return count[n]
```

C++

```
class Solution {
public:
    /**
     * @paramn n: An integer
     * @return: An integer
     */
    int numTrees(int n) {
        if (n < 0) {
            return -1;
        }

        vector<int> count(n + 1);
        count[0] = 1;
        for (int i = 1; i != n + 1; ++i) {
            for (int j = 0; j != i; ++j) {
                count[i] += count[j] * count[i - j - 1];
            }
        }

        return count[n];
    }
};
```

Java

```
public class Solution {
    /**
     * @paramn n: An integer
     * @return: An integer
     */
    public int numTrees(int n) {
        if (n < 0) {
            return -1;
        }

        int[] count = new int[n + 1];
        count[0] = 1;
        for (int i = 1; i < n + 1; ++i) {
            for (int j = 0; j < i; ++j) {
                count[i] += count[j] * count[i - j - 1];
            }
```

```
        }

        return count[n];
    }
}
```

源码分析

1. 对 n 小于0特殊处理。
2. 初始化大小为 n + 1 的数组，初始值为0，但对 count[0] 赋值为1.
3. 两重 for 循环递推求得 count[i] 的值。
4. 返回 count[n] 的值。

由于需要处理空节点的子树，故初始化 count[0] 为1便于乘法处理。其他值必须初始化为0，因为涉及到累加操作。

复杂度分析

一维数组大小为 n + 1, 空间复杂度为 $O(n+1)$. 两重 for 循环等差数列求和累计约 $n^2/2$, 故时间复杂度为 $O(n^2)$. 此题为 Catalan number 的一种，除了平方时间复杂度的解法外还存在 $O(n)$ 的解法，欲练此功，先戳 Wikipedia 的链接。

Reference

- > fisherlei. 水中的鱼: [LeetCode] Unique Binary Search Trees, Solution ↩
- Unique Binary Search Trees | 九章算法
- Catalan number - Wikipedia, the free encyclopedia

Update Bits

Source

- CTCI: (179) Update Bits

```
Given two 32-bit numbers, N and M, and two bit positions, i and j.
Write a method to set all bits between i and j in N equal to M
(e g , M becomes a substring of N located at i and starting at j)

Example
Given N=(10000000000)2, M=(10101)2, i=2, j=6

return N=(10001010100)2

Note
In the function, the numbers N and M will given in decimal,
you should also return a decimal number.

Challenge
Minimum number of operations?

Clarification
You can assume that the bits j through i have enough space to fit all of M.
That is, if M=10011,
you can assume that there are at least 5 bits between j and i.
You would not, for example, have j=3 and i=2,
because M could not fully fit between bit 3 and bit 2.
```

题解

Cracking The Coding Interview 上的题，题意简单来讲就是使用 M 代替 N 中的第 `i` 位到第 `j` 位。很显然，我们需要借用掩码操作。大致步骤如下：

1. 得到第 `i` 位到第 `j` 位的比特位为0，而其他位均为1的掩码 `mask` 。
2. 使用 `mask` 与 N 进行按位与，清零 N 的第 `i` 位到第 `j` 位。
3. 对 M 右移 `i` 位，将 M 放到 N 中指定的位置。
4. 返回 N | M 按位或的结果。

获得掩码 `mask` 的过程可参考 CTCI 书中的方法，先获得掩码(1111...000...111)的左边部分，然后获得掩码的右半部分，最后左右按位或即为最终结果。

C++

```
class Solution {
public:
    /**
     *@param n, m: Two integer
```

```
     *@param i, j: Two bit positions
     *return: An integer
     */
    int updateBits(int n, int m, int i, int j) {
        int ones = ~0;
        int left = ones << (j + 1);
        int right = ((1 << i) - 1);
        int mask = left | right;

        return (n & mask) | (m << i);
    }
};
```

源码分析

在给定测试数据 `[-521,0,31,31]` 时出现了 WA, 也就意味着目前这段程序是存在 bug 的，此时 `m = 0, i = 31, j = 31` ，仔细瞅瞅到底是哪几行代码有问题？本地调试后发现问题出在 `left` 那一行， `left` 移位后仍然为 `ones` , 这是为什么呢？在 `j` 为31时 `j + 1` 为32，也就是说此时对 `left` 位移的操作已经超出了此时 `int` 的最大位宽！

C++

```
class Solution {
public:
    /**
     *@param n, m: Two integer
     *@param i, j: Two bit positions
     *return: An integer
     */
    int updateBits(int n, int m, int i, int j) {
        int ones = ~0;
        int mask = 0;
        if (j < 31) {
            int left = ones << (j + 1);
            int right = ((1 << i) - 1);
            mask = left | right;
        } else {
            mask = (1 << i) - 1;
        }

        return (n & mask) | (m << i);
    }
};
```

源码分析

使用 `~0` 获得全1比特位，在 `j == 31` 时做特殊处理，即不必求 `left` 。求掩码的右侧1时使用了 `(1 << i) - 1` , 题中有保证第 `i` 位到第 `j` 位足以容纳 M, 故不必做溢出处理。

复杂度分析

时间复杂度和空间复杂度均为 $O(1)$.

C++

```
class Solution {
public:
    /**
     *@param n, m: Two integer
     *@param i, j: Two bit positions
     *return: An integer
     */
    int updateBits(int n, int m, int i, int j) {
        // get the bit width of input integer
        int bitwidth = 8 * sizeof(n);
        int ones = ~0;
        // use unsigned for logical shift
        unsigned int mask = ones << (bitwidth - (j - i + 1));
        mask = mask >> (bitwidth - 1 - j);

        return (n & (~mask)) | (m << i);
    }
};
```

源码分析

之前的实现需要使用 `if` 判断，但实际上还有更好的做法，即先获得 `mask` 的反码，最后取反即可。但这种方法需要提防有符号数，因为 C/C++ 中对有符号数的移位操作为算术移位，也就是说对负数右移时会在前面补零。解决办法可以使用无符号数定义 `mask`.

按题意 int 的位数为32，但考虑到通用性，可以使用 `sizeof` 获得其真实位宽。

复杂度分析

时间复杂度和空间复杂度均为 $O(1)$.

Reference

- c++ - logical shift right on signed data - Stack Overflow
- Update Bits | 九章算法
- *CTCI 5th Chapter 9.5* 中文版 p163

Fast Power

Source

- lintcode: (140) Fast Power

题解

数学题，考察整数求模的一些特性，不知道这个特性的话此题一时半会解不出来，本题中利用的关键特性为：

```
(a * b) % p = ((a % p) * (b % p)) % p
```

即 a 与 b 的乘积模 p 的值等于 a, b 分别模 p 相乘后再模 p 的值，只能帮你到这儿了，不看以下的答案先想想知道此关系后如何解这道题。

首先不太可能先把 a^n 具体值求出来，太大了... 所以利用以上求模公式，可以改写 a^n 为：

$$a^n = a^{n/2} \cdot a^{n/2} = a^{n/4} \cdot a^{n/4} \cdot a^{n/4} \cdot a^{n/4} \cdot = \ldots$$

至此递归模型建立。

Python

```python
class Solution:
    """
    @param a, b, n: 32bit integers
    @return: An integer
    """
    def fastPower(self, a, b, n):
        if n == 1:
            return a % b
        elif n == 0:
            # do not use `1` instead `1 % b` because `b = 1`
            return 1 % b
        elif n < 0:
            return -1

        # (a * b) % p = ((a % p) * (b % p)) % p
        product = self.fastPower(a, b, n / 2)
        product = (product * product) % b
        if n % 2 == 1:
            product = (product * a) % b

        return product
```

C++

```
class Solution {
public:
    /*
     * @param a, b, n: 32bit integers
     * @return: An integer
     */
    int fastPower(int a, int b, int n) {
        if (1 == n) {
            return a % b;
        } else if (0 == n) {
            // do not use 1 instead (1 % b)! b = 1
            return 1 % b;
        } else if (0 > n) {
            return -1;
        }

        // (a * b) % p = ((a % p) * (b % p)) % p
        // use long long to prevent overflow
        long long product = fastPower(a, b, n / 2);
        product = (product * product) % b;
        if (1 == n % 2) {
            product = (product * a) % b;
        }

        // cast long long to int
        return (int) product;
    }
};
```

Java

```
class Solution {
    /*
     * @param a, b, n: 32bit integers
     * @return: An integer
     */
    public int fastPower(int a, int b, int n) {
        if (n == 1) {
            return a % b;
        } else if (n == 0) {
            return 1 % b;
        } else if (n < 0) {
            return -1;
        }

        // (a * b) % p = ((a % p) * (b % p)) % p
        // use long to prevent overflow
        long product = fastPower(a, b, n / 2);
        product = (product * product) % b;
        if (n % 2 == 1) {
            product = (product * a) % b;
        }
```

```
        // cast long to int
        return (int) product;
    }
};
```

源码分析

分三种情况讨论 n 的值，需要特别注意的是 `n == 0`，虽然此时 a^0 的值为1，但是不可直接返回1，因为 `b == 1` 时应该返回0，故稳妥的写法为返回 `1 % b`.

递归模型中，需要注意的是要分 n 是奇数还是偶数，奇数的话需要多乘一个 a, 保存乘积值时需要使用 `long` 型防止溢出，最后返回时强制转换回 `int`。

复杂度分析

使用了临时变量 `product`，空间复杂度为 $O(1)$, 递归层数约为 $\log n$, 时间复杂度为 $O(\log n)$, 栈空间复杂度也为 $O(\log n)$.

Reference

- Lintcode: Fast Power 解题报告 - Yu's Garden - 博客园
- Fast Power 参考程序 Java/C++/Python

Hash Function

Source

- lintcode: (128) Hash Function

```
In data structure Hash, hash function is used to convert a string(or any other type)
into an integer smaller than hash size and bigger or equal to zero.
The objective of designing a hash function is to "hash" the key as unreasonable as possib
A good hash function can avoid collision as less as possible.
A widely used hash function algorithm is using a magic number 33,
consider any string as a 33 based big integer like follow:

hashcode("abcd") = (ascii(a) * 333 + ascii(b) * 332 + ascii(c) *33 + ascii(d)) % HASH_SIZ

                              = (97* 333 + 98 * 332 + 99 * 33 +100) % HASH_SIZE

                              = 3595978 % HASH_SIZE

here HASH_SIZE is the capacity of the hash table
(you can assume a hash table is like an array with index 0 ~ HASH_SIZE-1).

Given a string as a key and the size of hash table, return the hash value of this key.f



Have you met this question in a real interview? Yes
Example
For key="abcd" and size=100, return 78

Clarification
For this problem,
you are not necessary to design your own hash algorithm or
consider any collision issue,
you just need to implement the algorithm as described.
```

题解1

基本实现题，大多数人看到题目的直觉是按照定义来递推不就得了嘛，但其实这里面大有玄机，因为在字符串较长时使用 long 型来计算33的幂会溢出！所以这道题的关键在于如何处理**大整数溢出**。对于整数求模，`(a * b) % m = a % m * b % m` 这个基本公式务必牢记。根据这个公式我们可以大大降低时间复杂度和规避溢出。

Java

```java
class Solution {
    /**
     * @param key: A String you should hash
```

```
     * @param HASH_SIZE: An integer
     * @return an integer
     */
    public int hashCode(char[] key,int HASH_SIZE) {
        if (key == null || key.length == 0) return -1;

        long hashSum = 0;
        for (int i = 0; i < key.length; i++) {
            hashSum += key[i] * modPow(33, key.length - i - 1, HASH_SIZE);
            hashSum %= HASH_SIZE;
        }

        return (int)hashSum;
    }

    private long modPow(int base, int n, int mod) {
        if (n == 0) {
            return 1;
        } else if (n == 1) {
            return base % mod;
        } else if (n % 2 == 0) {
            long temp = modPow(base, n / 2, mod);
            return (temp % mod) * (temp % mod) % mod;
        } else {
            return (base % mod) * modPow(base, n - 1, mod) % mod;
        }
    }
}
```

源码分析

题解1属于较为直观的解法，只不过在计算33的幂时使用了私有方法 `modPow`，这个方法使用了对数级别复杂度的算法，可防止 TLE 的产生。注意两个 int 型数据在相乘时可能会溢出，故对中间结果的存储需要使用 long.

复杂度分析

遍历加求 `modPow`，时间复杂度 $O(n \log n)$, 空间复杂度 $O(1)$. 当然也可以使用哈希表的方法将幂求模的结果保存起来，这样一来空间复杂度就是 $O(n)$, 不过时间复杂度为 $O(n)$.

题解2 - 巧用求模公式

从题解1中我们可以看到其时间复杂度还是比较高的，作为基本库来使用是比较低效的。我们从范例 `hashcode("abc")` 为例进行说明。

$$
\begin{aligned}
hashcode(abc) &= (a \times 33^2 + b \times 33 + c)\%M \\
&= (33(33 \times a + b) + c)\%M \\
&= (33(33(33 \times 0 + a) + b) + c)\%M
\end{aligned}
$$

再根据 $(a \times b)\%M = (a\%M) \times (b\%M)$

从中可以看出使用迭代的方法较容易实现。

Java

```java
class Solution {
    /**
     * @param key: A String you should hash
     * @param HASH_SIZE: An integer
     * @return an integer
     */
    public int hashCode(char[] key,int HASH_SIZE) {
        if (key == null || key.length == 0) return -1;

        long hashSum = 0;
        for (int i = 0; i < key.length; i++) {
            hashSum = 33 * hashSum + key[i];
            hashSum %= HASH_SIZE;
        }

        return (int)hashSum;
    }
}
```

源码分析

精华在 `hashSum = 33 * hashSum + key[i];`

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

Count 1 in Binary

Source

- lintcode: (365) Count 1 in Binary

```
Count how many 1 in binary representation of a 32-bit integer.

Example
Given 32, return 1

Given 5, return 2

Given 1023, return 9

Challenge
If the integer is n bits with m 1 bits. Can you do it in O(m) time?
```

题解

题 O1 Check Power of 2 的进阶版，`x & (x - 1)` 的含义为去掉二进制数中1的最后一位，无论 x 是正数还是负数都成立。

Java

```java
public class Solution {
    /**
     * @param num: an integer
     * @return: an integer, the number of ones in num
     */
    public int countOnes(int num) {
        int count = 0;
        while (num != 0) {
            num = num & (num - 1);
            count++;
        }

        return count;
    }
}
```

源码分析

累加计数器即可。

复杂度分析

这种算法依赖于数中1的个数，时间复杂度为 $O(m)$. 空间复杂度 $O(1)$.

Reference

- Number of 1 bits | LeetCode - 评论中有关于不同算法性能的讨论

Fibonacci

Source

- lintcode: (366) Fibonacci

```
Find the Nth number in Fibonacci sequence.

A Fibonacci sequence is defined as follow:

The first two numbers are 0 and 1.
The i th number is the sum of i-1 th number and i-2 th number.
The first ten numbers in Fibonacci sequence is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

Example
Given 1, return 0

Given 2, return 1

Given 10, return 34

Note
The Nth fibonacci number won't exceed the max value of
signed 32-bit integer in the test cases.
```

题解

斐波那契数列使用递归极其容易实现，其实使用非递归的方法也很容易，不断向前滚动即可。

Java

```java
class Solution {
    /**
     * @param n: an integer
     * @return an integer f(n)
     */
    public int fibonacci(int n) {
        if (n < 0) return -1;
        if (n == 1) return 0;
        if (n == 2) return 1;

        int fn = 0, fn1 = 1, fn2 = 0;
        for (int i = 3; i <= n; i++) {
            fn = fn1 + fn2;
            fn2 = fn1;
            fn1 = fn;
        }
```

```
        return fn;
    }
}
```

源码分析

1. corner cases
2. 初始化 fn, fn1, fn2, 建立地推关系。
3. 注意 fn, fn2, fn1的递推顺序。

复杂度分析

遍历一次，时间复杂度为 $O(n)$, 使用了两个额外变量，空间复杂度为 $O(1)$.

A plus B Problem

Source

- lintcode: (1) A + B Problem

```
Write a function that add two numbers A and B.
You should not use + or any arithmetic operators.


Example
Given a=1 and b=2 return 3

Note
There is no need to read data from standard input stream.
Both parameters are given in function aplusb,
you job is to calculate the sum and return it.
Challenge
Of course you can just return a + b to get accepted.
But Can you challenge not do it like that?
Clarification
Are a and b both 32-bit integers?
Yes.
Can I use bit operation?

Sure you can.
```

题解

不用加减法实现加法，类似数字电路中的全加器，异或求得部分和，相与求得进位，最后将进位作为加法器的输入，典型的递归实现思路。

Java

```java
class Solution {
    /*
     * param a: The first integer
     * param b: The second integer
     * return: The sum of a and b
     */
    public int aplusb(int a, int b) {
        int result = a ^ b;
        int carry = a & b;
        carry <<= 1;
        if (carry != 0) {
            result = aplusb(result, carry);
        }

        return result;
    }
```

```
}
```

源码分析

递归步为进位是否为0，为0时返回。

复杂度分析

取决于进位，近似为 $O(1)$. 使用了部分额外变量，空间复杂度为 $O(1)$.

Print Numbers by Recursion

Source

- lintcode: (371) Print Numbers by Recursion

```
Print numbers from 1 to the largest number with N digits by recursion.

Example
Given N = 1, return [1,2,3,4,5,6,7,8,9].

Given N = 2, return [1,2,3,4,5,6,7,8,9,10,11,12,...,99].

Note
It's pretty easy to do recursion like:

recursion(i) {
    if i > largest number:
        return
    results.add(i)
    recursion(i + 1)
}
however this cost a lot of recursion memory as the recursion depth maybe very large.
Can you do it in another way to recursive with at most N depth?

Challenge
Do it in recursion, not for-loop.
```

题解

从小至大打印 N 位的数列，正如题目中所提供的 `recursion(i)`，解法简单粗暴，但问题在于 N 稍微大一点时栈就溢出了，因为递归深度太深了。能联想到的方法大概有两种，一种是用排列组合的思想去解释，把0~9当成十个不同的数(字符串表示)，塞到 N 个坑位中，这个用 DFS 来解应该是可行的；另一个则是使用数学方法，依次递归递推，比如 N=2 可由 N=1递归而来，具体方法则是乘10进位加法。题中明确要求递归深度最大不超过 N, 故 DFS 方法比较危险。

Java

```java
public class Solution {
    /**
     * @param n: An integer.
     * return : An array storing 1 to the largest number with n digits.
     */
    public List<Integer> numbersByRecursion(int n) {
        List<Integer> result = new ArrayList<Integer>();
        if (n <= 0) {
            return result;
        }
        helper(n, result);
```

```
        return result;
    }

    private void helper(int n, List<Integer> ret) {
        if (n == 0) return;
        helper(n - 1, ret);
        // current base such as 10, 20, 30...
        int base = (int)Math.pow(10, n - 1);
        // get List size before for loop
        int size = ret.size();
        for (int i = 1; i < 10; i++) {
            // add 10, 100, 1000...
            ret.add(i * base);
            for (int j = 0; j < size; j++) {
                // add 11, 12, 13...
                ret.add(ret.get(j) + base * i);
            }
        }
    }
}
```

源码分析

递归步的截止条件 `n == 0`，由于需要根据之前 N-1 位的数字递推，`base` 每次递归一层都需要乘 10，`size` 需要在 `for` 循环之前就确定。

复杂度分析

添加 10^n 个元素，时间复杂度 $O(10^n)$, 空间复杂度 $O(1)$.

Reference

- Lintcode: Print Numbers by Recursion | codesolutiony

Majority Number

Source

- leetcode: Majority Element | LeetCode OJ
- lintcode: (46) Majority Number

```
Given an array of integers, the majority number is
the number that occurs more than half of the size of the array. Find it.

Example
Given [1, 1, 1, 1, 2, 2, 2], return 1

Challenge
O(n) time and O(1) extra space
```

题解

找出现次数超过一半的数，使用哈希表统计不同数字出现的次数，超过二分之一即返回当前数字。这种方法非常简单且容易实现，但会占据过多空间，注意到题中明确表明要找的数会超过二分之一，这里的隐含条件不是那么容易应用。既然某个数超过二分之一，那么用这个数和其他数进行 PK，不同的计数器都减一**（核心在于两两抵消）**，相同的则加1，最后返回计数器大于0的即可。综上，我们需要一辅助数据结构如 `pair<int, int>`.

Java

```java
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: find a  majority number
     */
    public int majorityNumber(ArrayList<Integer> nums) {
        if (nums == null || nums.isEmpty()) return -1;

        // pair<key, count>
        int key = -1, count = 0;
        for (int num : nums) {
            // re-initialize
            if (count == 0) {
                key = num;
                count = 1;
                continue;
            }
            // increment/decrement count
            if (key == num) {
                count++;
            } else {
                count--;
            }
```

```
        }

        return key;
    }
}
```

源码分析

初始化 `count = 0`, 遍历数组时需要先判断 `count == 0` 以重新初始化。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

Majority Number II

Source

- leetcode: Majority Element II | LeetCode OJ
- lintcode: (47) Majority Number II

```
Given an array of integers,
the majority number is the number that occurs more than 1/3 of the size of the array.

Find it.

Example
Given [1, 2, 1, 2, 1, 3, 3], return 1.

Note
There is only one majority number in the array.

Challenge
O(n) time and O(1) extra space.
```

题解

题 Majority Number 的升级版，之前那道题是『两两抵消』，这道题自然则需要『三三抵消』，不过『三三抵消』需要注意不少细节，比如两个不同数的添加顺序和添加条件。

Java

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: The majority number that occurs more than 1/3
     */
    public int majorityNumber(ArrayList<Integer> nums) {
        if (nums == null || nums.isEmpty()) return -1;

        // pair
        int key1 = -1, key2 = -1;
        int count1 = 0, count2 = 0;
        for (int num : nums) {
            if (count1 == 0) {
                key1 = num;
                count1 = 1;
                continue;
            } else if (count2 == 0 && key1 != num) {
                key2 = num;
                count2 = 1;
                continue;
            }
            if (key1 == num) {
```

```
                count1++;
            } else if (key2 == num) {
                count2++;
            } else {
                count1--;
                count2--;
            }
        }

        count1 = 0;
        count2 = 0;
        for (int num : nums) {
            if (key1 == num) {
                count1++;
            } else if (key2 == num) {
                count2++;
            }
        }
        return count1 > count2 ? key1 : key2;
    }
}
```

源码分析

首先处理 `count == 0` 的情况，这里需要注意的是 `count2 == 0 && key1 = num`，不重不漏。最后再次遍历原数组也必不可少，因为由于添加顺序的区别，count1 和 count2的大小只具有相对意义，还需要最后再次比较其真实计数器值。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(2 \times 2) = O(1)$.

Reference

- Majority Number II 参考程序 Java/C++/Python

Majority Number III

Source

- lintcode: (48) Majority Number III

```
Given an array of integers and a number k,
the majority number is the number that occurs more than 1/k of the size of the array.

Find it.

Example
Given [3,1,2,3,2,3,3,4,4,4] and k=3, return 3.

Note
There is only one majority number in the array.

Challenge
O(n) time and O(k) extra space
```

题解

Majority Number II 的升级版，有了前两道题的铺垫，此题的思路已十分明了，对 K-1个数进行相互抵消，这里不太可能使用 key1, key2...等变量，用数组使用上不太方便，且增删效率不高，故使用哈希表较为合适，当哈希表的键值数等于 K 时即进行清理，当然更准备地来讲应该是等于 K-1时清理。故此题的逻辑即为：1. 更新哈希表，若遇哈希表 size == K 时则执行删除操作，最后遍历哈希表取真实计数器值，返回最大的 key.

Java

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: As described
     * @return: The majority number
     */
    public int majorityNumber(ArrayList<Integer> nums, int k) {
        HashMap<Integer, Integer> hash = new HashMap<Integer, Integer>();
        if (nums == null || nums.isEmpty()) return -1;

        // update HashMap
        for (int num : nums) {
            if (!hash.containsKey(num)) {
                hash.put(num, 1);
                if (hash.size() >= k) {
                    removeZeroCount(hash);
                }
            } else {
                hash.put(num, hash.get(num) + 1);
```

```
            }
        }

        // reset
        for (int key : hash.keySet()) {
            hash.put(key, 0);
        }
        for (int key : nums) {
            if (hash.containsKey(key)) {
                hash.put(key, hash.get(key) + 1);
            }
        }

        // find max
        int maxKey = -1, maxCount = 0;
        for (int key : hash.keySet()) {
            if (hash.get(key) > maxCount) {
                maxKey = key;
                maxCount = hash.get(key);
            }
        }

        return maxKey;
    }

    private void removeZeroCount(HashMap<Integer, Integer> hash) {
        Set<Integer> keySet = hash.keySet();
        for (int key : keySet) {
            hash.put(key, hash.get(key) - 1);
        }

        /* solution 1 */
        Iterator<Map.Entry<Integer, Integer>> it = hash.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry<Integer, Integer> entry = it.next();
            if(entry.getValue() == 0) {
                it.remove();
            }
        }

        /* solution 2 */
        // List<Integer> removeList = new ArrayList<>();
        // for (int key : keySet) {
        //     hash.put(key, hash.get(key) - 1);
        //     if (hash.get(key) == 0) {
        //         removeList.add(key);
        //     }
        // }
        // for (Integer key : removeList) {
        //     hash.remove(key);
        // }

        /* solution3 lambda expression for Java8 */
    }
}
```

源码分析

此题的思路不算很难，但是实现起来还是有点难度的，**Java 中删除哈希表时需要考虑线程安全。**

复杂度分析

时间复杂度 $O(n)$, 使用了哈希表，空间复杂度 $O(k)$.

Reference

- Majority Number III 参考程序 Java/C++/Python

Digit Counts

Source

- leetcode: Number of Digit One | LeetCode OJ
- lintcode: (3) Digit Counts

```
Count the number of k's between 0 and n. k can be 0 - 9.

Example
if n=12, k=1 in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
we have FIVE 1's (1, 10, 11, 12)
```

题解

leetcode 上的有点简单，这里以 Lintcode 上的为例进行说明。找出从0至整数 n 中出现数位k的个数，与整数有关的题大家可能比较容易想到求模求余等方法，但其实很多与整数有关的题使用字符串的解法更为便利。将整数 i 分解为字符串，然后遍历之，自增 k 出现的次数即可。

Java

```java
class Solution {
    /*
     * param k : As description.
     * param n : As description.
     * return: An integer denote the count of digit k in 1..n
     */
    public int digitCounts(int k, int n) {
        int count = 0;
        char kChar = (char)(k + '0');
        for (int i = k; i <= n; i++) {
            char[] iChars = Integer.toString(i).toCharArray();
            for (char iChar : iChars) {
                if (kChar == iChar) count++;
            }
        }

        return count;
    }
}
```

源码分析

太简单了，略

复杂度分析

时间复杂度 $O(n \times L)$, L 为n 的最大长度，拆成字符数组，空间复杂度 $O(L)$.

Ugly Number

Source

- leetcode: Ugly Number | LeetCode OJ
- lintcode: (4) Ugly Number

```
Ugly number is a number that only have factors 3, 5 and 7.

Design an algorithm to find the Kth ugly number.
The first 5 ugly numbers are 3, 5, 7, 9, 15 ...

Example
If K=4, return 9.
Challenge
O(K log K) or O(K) time.
```

题解1 - TLE

Lintcode 和 Leetcode 中质数稍微有点区别，这里以 Lintcode 上的版本为例进行说明。寻找第 K 个丑数，丑数在这里的定义是仅能被3，5，7整除。简单粗暴的方法就是挨个检查正整数，数到第 K 个丑数时即停止。

Java

```java
class Solution {
    /**
     * @param k: The number k.
     * @return: The kth prime number as description.
     */
    public long kthPrimeNumber(int k) {
        if (k <= 0) return -1;

        int count = 0;
        long num = 1;
        while (count < k) {
            num++;
            if (isUgly(num)) {
                count++;
            }
        }

        return num;
    }

    private boolean isUgly(long num) {
        while (num % 3 == 0) {
            num /= 3;
        }
```

```
        while (num % 5 == 0) {
            num /= 5;
        }
        while (num % 7 == 0) {
            num /= 7;
        }

        return num == 1;
    }
}
```

源码分析

判断丑数时依次约掉质因数3，5，7，若处理完所有质因数3,5,7后不为1则不是丑数。自增 num 时应在判断是否为丑数之前。

复杂度分析

无法准确分析，但是估计在 $O(n^3)$ 以上。

题解2 - 二分查找

根据丑数的定义，它的质因数只含有3, 5, 7, 那么根据这一点其实可以知道后面的丑数一定可以从前面的丑数乘3,5,7得到。那么可不可以将其递推关系用数学公式表示出来呢?

我大概做了下尝试，首先根据丑数的定义可以写成 $U_k = 3^{x_3} \cdot 5^{x_5} \cdot 7^{x_7}$, 那么 U_{k+1} 和 U_k 的不同则在于 x_3, x_5, x_7 的不同，递推关系为 $U_{k+1} = U_k \cdot \frac{3^{y3} \cdot 5^{y5} \cdot 7^{y7}}{3^{z3} \cdot 5^{z5} \cdot 7^{z7}}$,将这些分数按照从小到大的顺序排列可在 $O(K)$ 的时间内解决，但是问题来了，得到这些具体的 y, z 就需要费不少时间，且人工操作极易漏解。:(所以这种解法只具有数学意义，没有想到好的实现方法。

除了这种找相邻递推关系的方法我们还可以尝试对前面的丑数依次乘3, 5, 7，直至找到比当前最大的丑数大的一个数，对乘积后的三种不同结果取最小值即可得下一个最大的丑数。这种方法需要保存之前的 N 个丑数，由于已按顺序排好，天然的二分法。

Java

```
class Solution {
    /**
     * @param k: The number k.
     * @return: The kth prime number as description.
     */
    public long kthPrimeNumber(int k) {
        if (k <= 0) return -1;

        ArrayList<Long> nums = new ArrayList<Long>();
        nums.add(1L);
        for (int i = 0; i < k; i++) {
            long minNextUgly = Math.min(nextUgly(nums, 3), nextUgly(nums, 5));
            minNextUgly = Math.min(minNextUgly, nextUgly(nums, 7));
```

```
            nums.add(minNextUgly);
        }

        return nums.get(nums.size() - 1);
    }

    private long nextUgly(ArrayList<Long> nums, int factor) {
        int size = nums.size();
        int start = 0, end = size - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (nums.get(mid) * factor > nums.get(size - 1)) {
                end = mid;
            } else {
                start = mid;
            }
        }
        if (nums.get(start) * factor > nums.get(size - 1)) {
            return nums.get(start) * factor;
        }

        return nums.get(end) * factor;
    }
}
```

源码分析

`nextUgly` 根据输入的丑数数组和 factor 决定下一个丑数， `nums.add(1L)` 中1后面需要加 L表示 Long, 否则编译错误。

复杂度分析

找下一个丑数 $O(\log K)$, 循环 K 次，故总的时间复杂度近似 $O(K \log K)$, 使用了数组保存丑数，空间复杂度 $O(K)$.

题解3 - 动态规划

TBD

Reference

- 《剑指 Offer》第五章
- Ugly Numbers - GeeksforGeeks

Plus One

Source

- leetcode: Plus One | LeetCode OJ
- lintcode: (407) Plus One

Problem

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example

Given [1,2,3] which represents 123, return [1,2,4].

Given [9,9,9] which represents 999, return [1,0,0,0].

题解

又是一道两个整数按数位相加的题，自后往前累加，处理下进位即可。这道题中是加1，其实还可以扩展至加2，加3等。

Java

```
public class Solution {
    /**
     * @param digits a number represented as an array of digits
     * @return the result
     */
    public int[] plusOne(int[] digits) {
        return plusDigit(digits, 1);
    }

    private int[] plusDigit(int[] digits, int digit) {
        if (digits == null || digits.length == 0) return null;

        // regard digit(0~9) as carry
        int carry = digit;
        int[] result = new int[digits.length];
        for (int i = digits.length - 1; i >= 0; i--) {
            result[i] = (digits[i] + carry) % 10;
            carry = (digits[i] + carry) / 10;
        }

        // carry == 1
        if (carry == 1) {
            int[] finalResult = new int[result.length + 1];
```

```
            finalResult[0] = 1;
            return finalResult;
        }

        return result;
    }
}
```

源码分析

源码中单独实现了加任何数(0~9)的私有方法，更为通用，对于末尾第一个数，可以将要加的数当做进位处理，这样就不必单独区分最后一位了，十分优雅！

复杂度分析

Java 中需要返回数组，而这个数组在处理之前是不知道大小的，故需要对最后一个进位单独处理。时间复杂度 $O(n)$, 空间复杂度在最后一位有进位时恶化为 $O(n)$, 当然也可以通过两次循环使得空间复杂度为 $O(1)$.

Reference

- Soulmachine 的 leetcode 题解，将要加的数当做进位处理就是从这学到的。

Linked List - 链表

本节包含链表的一些常用操作，如删除、插入和合并等。

常见错误有 遍历链表不向前递推节点，遍历链表前未保存头节点，返回链表节点指针错误。

Remove Duplicates from Sorted List

Source

- leetcode: Remove Duplicates from Sorted List | LeetCode OJ
- lintcode: (112) Remove Duplicates from Sorted List

```
Given a sorted linked list,
delete all duplicates such that each element appear only once.

Example
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.
```

题解

遍历之，遇到当前节点和下一节点的值相同时，删除下一节点，并将当前节点 `next` 值指向下一个节点的 `next`，当前节点首先保持不变，直到相邻节点的值不等时才移动到下一节点。

Python

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def deleteDuplicates(self, head):
        if head is None:
            return None

        node = head
        while node.next is not None:
            if node.val == node.next.val:
                node.next = node.next.next
            else:
                node = node.next

        return head
```

C++

```
/**
 * Definition of ListNode
```

```
 * 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: head node
     */
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == NULL) {
            return NULL;
        }

        ListNode *node = head;
        while (node->next != NULL) {
            if (node->val == node->next->val) {
                ListNode *temp = node->next;
                node->next = node->next->next;
                delete temp;
            } else {
                node = node->next;
            }
        }

        return head;
    }
};
```

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

        ListNode node = head;
        while (node.next != null) {
            if (node.val == node.next.val) {
                node.next = node.next.next;
            } else {
                node = node.next;
            }
```

```
        }

        return head;
    }
}
```

源码分析

1. 首先进行异常处理，判断head是否为NULL
2. 遍历链表， `node->val == node->next->val` 时，保存 `node->next` ，便于后面释放内存(非C/C++无需手动管理内存)
3. 不相等时移动当前节点至下一节点，注意这个步骤必须包含在 `else` 中，否则逻辑较为复杂

`while` 循环处也可使用 `node != null && node->next != null` , 这样就不用单独判断 `head` 是否为空了，但是这样会降低遍历的效率，因为需要判断两处。

复杂度分析

遍历链表一次，时间复杂度为 $O(n)$, 使用了一个中间变量进行遍历，空间复杂度为 $O(1)$.

Reference

- Remove Duplicates from Sorted List 参考程序 | 九章

Remove Duplicates from Sorted List II

Source

- leetcode: Remove Duplicates from Sorted List II | LeetCode OJ
- lintcode: (113) Remove Duplicates from Sorted List II

```
Given a sorted linked list, delete all nodes that have duplicate numbers,
leaving only distinct numbers from the original list.

Example
Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.
```

题解

上题为保留重复值节点的一个，这题删除全部重复节点，看似区别不大，但是考虑到链表头不确定(可能被删除，也可能保留)，因此若用传统方式需要较多的if条件语句。这里介绍一个**处理链表头节点不确定的方法——引入dummy node.**

```
ListNode *dummy = new ListNode(0);
dummy->next = head;
ListNode *node = dummy;
```

引入新的指针变量 `dummy` ，并将其next变量赋值为head，考虑到原来的链表头节点可能被删除，故应该从dummy处开始处理，这里复用了head变量。考虑链表 `A->B->C` ，删除B时，需要处理和考虑的是A和C，将A的next指向C。如果从空间使用效率考虑，可以使用head代替以上的node，含义一样，node比较好理解点。

与上题不同的是，由于此题引入了新的节点 `dummy` ，不可再使用 `node->val == node->next->val` ，原因有二：

1. 此题需要将值相等的节点全部删掉，而删除链表的操作与节点前后两个节点都有关系，故需要涉及三个链表节点。且删除单向链表节点时不能删除当前节点，只能改变当前节点的 `next` 指向的节点。
2. 在判断val是否相等时需先确定 `node->next` 和 `node->next->next` 均不为空，否则不可对其进行取值。

说多了都是泪，先看看我的错误实现：

C++ - Wrong

```
/**
 * Definition of ListNode
 * 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: head node
     */
    ListNode * deleteDuplicates(ListNode *head) {
        if (head == NULL || head->next == NULL) {
            return NULL;
        }

        ListNode *dummy;
        dummy->next = head;
        ListNode *node = dummy;

        while (node->next != NULL && node->next->next != NULL) {
            if (node->next->val == node->next->next->val) {
                int val = node->next->val;
                while (node->next != NULL && val == node->next->val) {
                    ListNode *temp = node->next;
                    node->next = node->next->next;
                    delete temp;
                }
            } else {
                node->next = node->next->next;
            }
        }

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

错因分析

错在什么地方？

1. 节点dummy的初始化有问题，对类的初始化应该使用 `new`
2. 在else语句中 `node->next = node->next->next;` 改写了 `dummy-next` 中的内容，返回的 `dummy-next` 不再是队首元素，而是队尾元素。原因很微妙，应该使用 `node = node->next;` ，node代表节点指针变量，而node->next代表当前节点所指向的下一节点地址。具体分析可自行在纸上画图分析，可对指针和链表的理解又加深不少。

图中上半部分为ListNode的内存示意图，每个框底下为其内存地址。 `dummy` 指针变量本身的地址为ox7fff5d0d2500，其保存着指针变量值为0x7fbe7bc04c50. `head` 指针变量本身的地址为ox7fff5d0d2508，其保存着指针变量值为0x7fbe7bc04c00.

好了，接下来看看正确实现及解析。

Python

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def deleteDuplicates(self, head):
        if head is None:
            return None

        dummy = ListNode(0)
        dummy.next = head
        node = dummy
        while node.next is not None and node.next.next is not None:
            if node.next.val == node.next.next.val:
```

```
                val_prev = node.next.val
                while node.next is not None and node.next.val == val_prev:
                    node.next = node.next.next
            else:
                node = node.next

        return dummy.next
```

C++

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == NULL) return NULL;

        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode *node = dummy;
        while (node->next != NULL && node->next->next != NULL) {
            if (node->next->val == node->next->next->val) {
                int val_prev = node->next->val;
                // remove ListNode node->next
                while (node->next != NULL && val_prev == node->next->val) {
                    ListNode *temp = node->next;
                    node->next = node->next->next;
                    delete temp;
                }
            } else {
                node = node->next;
            }
        }

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

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
```

```java
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode node = dummy;
        while(node.next != null && node.next.next != null) {
            if (node.next.val == node.next.next.val) {
                int val_prev = node.next.val;
                while (node.next != null && node.next.val == val_prev) {
                    node.next = node.next.next;
                }
            } else {
                node = node.next;
            }
        }

        return dummy.next;
    }
}
```

源码分析

1. 首先考虑异常情况，head 为 NULL 时返回 NULL
2. new一个dummy变量， `dummy->next` 指向原链表头。
3. 使用新变量node并设置其为dummy头节点，遍历用。
4. 当前节点和下一节点val相同时先保存当前值，便于while循环终止条件判断和删除节点。注意这一段代码也比较精炼。
5. 最后返回 `dummy->next` ，即题目所要求的头节点。

Python 中也可不使用 `is not None` 判断，但是效率会低一点。

复杂度分析

两根指针(node.next 和 node.next.next)遍历，时间复杂度为 $O(2n)$. 使用了一个 dummy 和中间缓存变量，空间复杂度近似为 $O(1)$.

Reference

- Remove Duplicates from Sorted List II | 九章

Remove Duplicates from Unsorted List

Source

- Remove duplicates from an unsorted linked list - GeeksforGeeks

```
Write a removeDuplicates() function which takes a list and deletes
any duplicate nodes from the list. The list is not sorted.

For example if the linked list is 12->11->12->21->41->43->21,
then removeDuplicates() should convert the list to 12->11->21->41->43.

If temporary buffer is not allowed, how to solve it?
```

题解1 - 两重循环

Remove Duplicates 系列题，之前都是已排序链表，这个题为未排序链表。原题出自 *CTCI* 题2.1。

最容易想到的简单办法就是两重循环删除重复节点了，当前遍历节点作为第一重循环，当前节点的下一节点作为第二重循环。

Python

```python
"""
Definition of ListNode
class ListNode(object):
    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: A ListNode
    @return: A ListNode
    """
    def deleteDuplicates(self, head):
        if head is None:
            return None

        curr = head
        while curr is not None:
            inner = curr
            while inner.next is not None:
                if inner.next.val == curr.val:
                    inner.next = inner.next.next
                else:
                    inner = inner.next
            curr = curr.next

        return head
```

C++

```
/**
 * Definition of ListNode
 * 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: head node
     */
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == NULL) return NULL;

        ListNode *curr = head;
        while (curr != NULL) {
            ListNode *inner = curr;
            while (inner->next != NULL) {
                if (inner->next->val == curr->val) {
                    inner->next = inner->next->next;
                } else {
                    inner = inner->next;
                }
            }
            curr = curr->next;
        }

        return head;
    }
};
```

Java

```
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
```

```java
public class Solution {
    /**
     * @param ListNode head is the head of the linked list
     * @return: ListNode head of linked list
     */
    public static ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

        ListNode curr = head;
        while (curr != null) {
            ListNode inner = curr;
            while (inner.next != null) {
                if (inner.next.val == curr.val) {
                    inner.next = inner.next.next;
                } else {
                    inner = inner.next;
                }
            }
            curr = curr.next;
        }

        return head;
    }
}
```

源码分析

删除链表的操作一般判断 `node.next` 较为合适，循环时注意 `inner = inner.next` 和 `inner.next = inner.next.next` 的区别即可。

复杂度分析

两重循环，时间复杂度为 $O(\frac{1}{2}n^2)$, 空间复杂度近似为 $O(1)$.

题解2 - 万能的 hashtable

使用辅助空间哈希表，节点值作为键，布尔值作为相应的值（是否为布尔值其实无所谓，关键是键）。

Python

```python
"""
Definition of ListNode
class ListNode(object):
    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: A ListNode
    @return: A ListNode
    """
```

```
    def deleteDuplicates(self, head):
        if head is None:
            return None

        hash = {}
        hash[head.val] = True
        curr = head
        while curr.next is not None:
            if hash.has_key(curr.next.val):
                curr.next = curr.next.next
            else:
                hash[curr.next.val] = True
                curr = curr.next

        return head
```

C++

```
/**
 * Definition of ListNode
 * 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: head node
     */
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == NULL) return NULL;

        // C++ 11 use unordered_map
        // unordered_map<int, bool> hash;
        map<int, bool> hash;
        hash[head->val] = true;
        ListNode *curr = head;
        while (curr->next != NULL) {
            if (hash.find(curr->next->val) != hash.end()) {
                ListNode *temp = curr->next;
                curr->next = curr->next->next;
                delete temp;
            } else {
                hash[curr->next->val] = true;
                curr = curr->next;
            }
        }

        return head;
    }
```

```
};
```

Java

```
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode head is the head of the linked list
     * @return: ListNode head of linked list
     */
    public static ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

        ListNode curr = head;
        HashMap<Integer, Boolean> hash = new HashMap<Integer, Boolean>();
        hash.put(curr.val, true);
        while (curr.next != null) {
            if (hash.containsKey(curr.next.val)) {
                curr.next = curr.next.next;
            } else {
                hash.put(curr.next.val, true);
                curr = curr.next;
            }
        }

        return head;
    }
}
```

源码分析

删除链表中某个节点的经典模板在 `while` 循环中体现。

复杂度分析

遍历一次链表，时间复杂度为 $O(n)$, 使用了额外的哈希表，空间复杂度近似为 $O(n)$.

Reference

- Remove duplicates from an unsorted linked list - GeeksforGeeks
- ctci/Question.java at master · gaylemcd/ctci

Partition List

Source

- leetcode: Partition List | LeetCode OJ
- lintcode: (96) Partition List

```
Given a linked list and a value x, partition it such that all nodes
less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes
in each of the two partitions.

For example,
Given 1->4->3->2->5->2->null and x = 3,
return 1->2->2->4->3->5->null.
```

题解

此题出自 *CTCI* 题 2.4，依据题意，是要根据值x对链表进行分割操作，具体是指将所有小于x的节点放到不小于x的节点之前，咋一看和快速排序的分割有些类似，但是这个题的不同之处在于只要求将小于x的节点放到前面，而并不要求对元素进行排序。

这种分割的题使用两路指针即可轻松解决。左边指针指向小于x的节点，右边指针指向不小于x的节点。由于左右头节点不确定，我们可以使用两个dummy节点。

Python

```python
"""
Definition of ListNode
class ListNode(object):

    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of linked list.
    @param x: an integer
    @return: a ListNode
    """
    def partition(self, head, x):
        if head is None:
            return None

        leftDummy = ListNode(0)
        left = leftDummy
        rightDummy = ListNode(0)
```

```
        right = rightDummy
        node = head
        while node is not None:
            if node.val < x:
                left.next = node
                left = left.next
            else:
                right.next = node
                right = right.next
            node = node.next
        # post-processing
        right.next = None
        left.next = rightDummy.next

        return leftDummy.next
```

C++

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        if (head == NULL) return NULL;

        ListNode *leftDummy = new ListNode(0);
        ListNode *left = leftDummy;
        ListNode *rightDummy = new ListNode(0);
        ListNode *right = rightDummy;
        ListNode *node = head;
        while (node != NULL) {
            if (node->val < x) {
                left->next = node;
                left = left->next;
            } else {
                right->next = node;
                right = right->next;
            }
            node = node->next;
        }
        // post-processing
        right->next = NULL;
        left->next = rightDummy->next;

        return leftDummy->next;
    }
};
```

Java

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode partition(ListNode head, int x) {
        if (head == null) return null;

        ListNode leftDummy = new ListNode(0);
        ListNode left = leftDummy;
        ListNode rightDummy = new ListNode(0);
        ListNode right = rightDummy;
        ListNode node = head;
        while (node != null) {
            if (node.val < x) {
                left.next = node;
                left = left.next;
            } else {
                right.next = node;
                right = right.next;
            }
            node = node.next;
        }
        // post-processing
        right.next = null;
        left.next = rightDummy.next;

        return leftDummy.next;
    }
}
```

源码分析

1. 异常处理
2. 引入左右两个dummy节点及left和right左右尾指针
3. 遍历原链表
4. 处理右链表，置 `right->next` 为空，将右链表的头部链接到左链表尾指针的next，返回左链表的头部

复杂度分析

遍历链表一次，时间复杂度近似为 $O(n)$, 使用了两个 dummy 节点及中间变量，空间复杂度近似为 $O(1)$.

Two Lists Sum

Source

- CC150 - (167) Two Lists Sum

```
You have two numbers represented by a linked list, where each node contains a single digi
The digits are stored in reverse order, such that the 1's digit is at the head of the lis
Write a function that adds the two numbers and returns the sum as a linked list.

Example
Given two lists, 3->1->5->null and 5->9->2->null, return 8->0->8->null
```

题解

一道看似简单的进位加法题，实则杀机重重，不信你不看答案自己先做做看。

首先由十进制加法可知应该注意进位的处理，但是这道题仅注意到这点就够了吗？还不够！因为两个链表长度有可能不等长！因此这道题的亮点在于边界和异常条件的处理，来瞅瞅我自认为相对优雅的实现。

C++ - Iteration

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
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) {
        if (NULL == l1 && NULL == l2) {
            return NULL;
        }

        ListNode *sumlist = new ListNode(0);
        ListNode *templist = sumlist;

        int carry = 0;
        while ((NULL != l1) || (NULL != l2) || (0 != carry)) {
            // padding for NULL
            int l1_val = (NULL == l1) ? 0 : l1->val;
```

```
            int l2_val = (NULL == l2) ? 0 : l2->val;

            templist->val = (carry + l1_val + l2_val) % 10;
            carry = (carry + l1_val + l2_val) / 10;

            if (NULL != l1) l1 = l1->next;
            if (NULL != l2) l2 = l2->next;

            // return sumlist before generating new ListNode
            if ((NULL == l1) && (NULL == l2) && (0 == carry)) {
                return sumlist;
            }
            templist->next = new ListNode(0);
            templist = templist->next;
        }

        return sumlist;
    }
};
```

源码分析

1. 迭代能正常进行的条件为 `(NULL != l1) || (NULL != l2) || (0 != carry)`，缺一不可。
2. 对于空指针节点的处理可以用相对优雅的方式处理 - `int l1_val = (NULL == l1) ? 0 : l1->val;`
3. 生成新节点时需要先判断迭代终止条件 - `(NULL == l1) && (NULL == l2) && (0 == carry)`，避免多生成一位数0。

复杂度分析

没啥好分析的，时间和空间复杂度均为 $O(max(L1, L2))$.

C++ - Recursion

除了使用迭代，对于链表类问题也比较适合使用递归实现。

To-be done.

Reference

- *CC150 Chapter 9.2* 题2.5，中文版 p123
- Add two numbers represented by linked lists | Set 1 - GeeksforGeeks

Two Lists Sum Advanced

Source

- CC150 - Add two numbers represented by linked lists | Set 2 - GeeksforGeeks

```
Given two numbers represented by two linked lists, write a function that returns sum list
The sum list is linked list representation of addition of two input numbers.

Example

Input:
  First  List: 5->6->3  // represents number 563
  Second List: 8->4->2  // represents number 842
Output
  Resultant list: 1->4->0->5  // represents number 1405

Challenge

Not allowed to modify the lists.
Not allowed to use explicit extra space.
```

题解1 - 反转链表

在题 Two Lists Sum | Data Structure and Algorithm 的基础上改了下数位的表示方式，前者低位在前，高位在后，这个题的高位在前，低位在后。很自然地可以联想到先将链表反转，而后再使用 Two Lists Sum 的解法。

Reference

- Add two numbers represented by linked lists | Set 2 - GeeksforGeeks

Remove Nth Node From End of List

Source

- lintcode: (174) Remove Nth Node From End of List

```
Given a linked list, remove the nth node from the end of list and return its head.

Note
The minimum number of nodes in list is n.

Example
Given linked list: 1->2->3->4->5->null, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5->null.

Challenge
O(n) time
```

题解

简单题， 使用快慢指针解决此题，需要注意最后删除的是否为头节点。让快指针先走 `n` 步，直至快指针走到终点，找到需要删除节点之前的一个节点，改变 `node->next` 域即可。

C++

```
/**
 * Definition of ListNode
 * 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 n: An integer.
     * @return: The head of linked list.
     */
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        if (NULL == head || n < 0) {
            return NULL;
        }
```

```
        ListNode *preN = head;
        ListNode *tail = head;
        // slow fast pointer
        int index = 0;
        while (index < n) {
            if (NULL == tail) {
                return NULL;
            }
            tail = tail->next;
            ++index;
        }

        if (NULL == tail) {
            return head->next;
        }

        while (tail->next) {
            tail = tail->next;
            preN = preN->next;
        }
        preN->next = preN->next->next;

        return head;
    }
};
```

以上代码单独判断了是否需要删除头节点的情况，在遇到头节点不确定的情况下，引入 `dummy` 节点将会使代码更加优雅，改进的代码如下。

C++ dummy node

```
/**
 * Definition of ListNode
 * 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 n: An integer.
     * @return: The head of linked list.
     */
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        if (NULL == head || n < 1) {
            return NULL;
        }

        ListNode *dummy = new ListNode(0);
```

```
        dummy->next = head;
        ListNode *preDel = dummy;

        for (int i = 0; i != n; ++i) {
            if (NULL == head) {
                return NULL;
            }
            head = head->next;
        }

        while (head) {
            head = head->next;
            preDel = preDel->next;
        }
        preDel->next = preDel->next->next;

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

源码分析

引入 `dummy` 节点后画个图分析下就能确定 `head` 和 `preDel` 的转移关系了。

Linked List Cycle

Source

- leetcode: Linked List Cycle | LeetCode OJ
- lintcode: (102) Linked List Cycle

```
Given a linked list, determine if it has a cycle in it.

Example
Given -21->10->4->5, tail connects to node index 1, return true

Challenge
Follow up:
Can you solve it without using extra space?
```

题解 - 快慢指针

对于带环链表的检测，效率较高且易于实现的一种方式为使用快慢指针。快指针每次走两步，慢指针每次走一步，如果快慢指针相遇(快慢指针所指内存为同一区域)则有环，否则快指针会一直走到 `NULL` 为止退出循环，返回 `false` .

快指针走到 `NULL` 退出循环即可确定此链表一定无环这个很好理解。那么带环的链表快慢指针一定会相遇吗？先来看看下图。

在有环的情况下，最终快慢指针一定都走在环内，加入第 `i` 次遍历时快指针还需要 `k` 步才能追上慢指针，由于快指针比慢指针每次多走一步。那么每遍历一次快慢指针间的间距都会减少1，直至最终相遇。故快慢指针相遇一定能确定该链表有环。

C++

```
/**
```

```
 * Definition of ListNode
 * 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: True if it has a cycle, or false
     */
    bool hasCycle(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return false;
        }

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

        return false;
    }
};
```

源码分析

1. 异常处理，将 `head->next` 也考虑在内有助于简化后面的代码。
2. 慢指针初始化为 `head`，快指针初始化为 `head` 的下一个节点，这是快慢指针初始化的一种方法，有时会简化边界处理，但有时会增加麻烦，比如该题的进阶版。

复杂度分析

1. 在无环时，快指针每次走两步走到尾部节点，遍历的时间复杂度为 $O(n/2)$.
2. 有环时，最坏的时间复杂度近似为 $O(n)$. 最坏情况下链表的头尾相接，此时快指针恰好在慢指针前一个节点，还需 n 次快慢指针相遇。最好情况和无环相同，尾节点出现环。

故总的时间复杂度可近似为 $O(n)$.

Reference

- Linked List Cycle | 九章算法

Linked List Cycle II

Source

- leetcode: Linked List Cycle II | LeetCode OJ
- lintcode: (103) Linked List Cycle II

```
Given a linked list, return the node where the cycle begins. If there is no cycle, return

Example
Given -21->10->4->5, tail connects to node index 1, return node 10

Challenge
Follow up:
Can you solve it without using extra space?
```

题解 - 快慢指针

题 Linked List Cycle | Data Structure and Algorithm 的升级版，题目要求不适用额外空间，则必然还是使用快慢指针解决问题。首先设组成环的节点个数为 r, 链表中节点个数为 n. 首先我们来分析下在链表有环时都能推出哪些特性：

1. 快慢指针第一次相遇时快指针比慢指针多走整数个环, 这个容易理解，相遇问题。
2. 每次相遇都在同一个节点。第一次相遇至第二次相遇，快指针需要比慢指针多走一个环的节点个数，而快指针比慢指针多走的步数正好是慢指针自身移动的步数，故慢指针恰好走了一圈回到原点。

从以上两个容易得到的特性可知，在仅仅知道第一次相遇时的节点还不够，相遇后如果不改变既有策略则必然找不到环的入口。接下来我们分析下如何从第一次相遇的节点走到环的入口节点。还是让我们先从实际例子出发，以下图为例。

`slow` 和 `fast` 节点分别初始化为节点 `1` 和 `2`，假设快慢指针第一次相遇的节点为 `0`，对应于环中的第 `i` 个节点 C_i, 那么此时慢指针正好走了 $n-r-1+i$ 步，快指针则走了 $2\cdot(n-r-1+i)$ 步，且存在[1]: $n-r-1+i+1=l\cdot r$. (之所以在 `i` 后面加1是因为快指针初始化时多走了一步) 快慢指针第一次相遇时慢指针肯定没有走完整个环，且慢指针走的步数即为整数个环节点个数，由性质1和性质2可联合推出。

现在分析下相遇的节点和环的入口节点之间的关联，要从环中第 `i` 个节点走到环的入口节点，则按照顺时针方向移动[2]: $(l\cdot r-i+1)$ 个节点 (l 为某个非负整数) 即可到达。现在来看看式[1]和式[2]间的关系。由式[1]可以推知 $n-r=l\cdot r-i$. 从头节点走到环的入口节点所走的步数可用 $n-r$ 表示，故在快慢指针第一次相遇时让另一节点从头节点出发，慢指针仍从当前位置迭代，第二次相遇时的位置即为环的入口节点！

> 由于此题快指针初始化为头节点的下一个节点，故分析起来稍微麻烦些，且在第一次相遇后需要让慢指针先走一步，否则会出现死循环。

对于该题来说，快慢指针都初始化为头节点会方便很多，故以下代码使用头节点对快慢指针进行初始化。

C++

```
/**
 * Definition of ListNode
 * 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) {
        if (NULL == head || NULL == head->next) {
            return NULL;
        }

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

```
        return NULL;
    }
};
```

源码分析

1. 异常处理。
2. 找第一次相遇的节点。
3. 将 `fast` 置为头节点，并只走一步，直至快慢指针第二次相遇，返回慢指针所指的节点。

复杂度分析

第一次相遇的最坏时间复杂度为 $O(n)$, 第二次相遇的最坏时间复杂度为 $O(n)$. 故总的时间复杂度近似为 $O(n)$, 空间复杂度 $O(1)$.

Reference

- Linked List Cycle II | 九章算法

Reverse Linked List

Source

- leetcode: Reverse Linked List | LeetCode OJ
- lintcode: (35) Reverse Linked List

```
Reverse a linked list.

Example
For linked list 1->2->3, the reversed linked list is 3->2->1

Challenge
Reverse it in-place and in one-pass
```

题解1 - 非递归

联想到同样也可能需要翻转的数组，在数组中由于可以利用下标随机访问，翻转时使用下标即可完成。而在单向链表中，仅仅只知道头节点，而且只能单向往前走，故需另寻出路。分析由 `1->2->3` 变为 `3->2->1` 的过程，由于是单向链表，故只能由1开始遍历，1和2最开始的位置是 `1->2` ，最后变为 `2->1` ，故从这里开始寻找突破口，探讨如何交换1和2的节点。

```
temp = head->next;
head->next = prev;
prev = head;
head = temp;
```

要点在于维护两个指针变量 `prev` 和 `head` , 翻转相邻两个节点之前保存下一节点的值，分析如下图所示：

1. 保存head下一节点
2. 将head所指向的下一节点改为prev
3. 将prev替换为head，波浪式前进
4. 将第一步保存的下一节点替换为head，用于下一次循环

Python

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        prev = None
        curr = head
        while curr is not None:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        # fix head
        head = prev

        return head
```

C++

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        ListNode *prev = NULL;
        ListNode *curr = head;
        while (curr != NULL) {
            ListNode *temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
};
```

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode temp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
}
```

源码分析

题解中基本分析完毕，代码中的prev赋值比较精炼，值得借鉴。

复杂度分析

遍历一次链表，时间复杂度为 $O(n)$, 使用了辅助变量，空间复杂度 $O(1)$.

题解2 - 递归

递归的终止步分三种情况讨论：

1. 原链表为空，直接返回空链表即可。
2. 原链表仅有一个元素，返回该元素。
3. 原链表有两个以上元素，由于是单链表，故翻转需要自尾部向首部逆推。

由尾部向首部逆推时大致步骤为先翻转当前节点和下一节点，然后将当前节点指向的下一节点置空(否则会出现死循环和新生成的链表尾节点不指向空)，如此递归到头节点为止。新链表的头节点在整个递归过程中一直没有变化，逐层向上返回。

Python

```
"""
Definition of ListNode
```

```
class ListNode(object):

    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of the linked list.
    @return: You should return the head of the reversed linked list.
                  Reverse it in-place.
    """
    def reverse(self, head):
        # case1: empty list
        if head is None:
            return head
        # case2: only one element list
        if head.next is None:
            return head
        # case3: reverse from the rest after head
        newHead = self.reverse(head.next)
        # reverse between head and head->next
        head.next.next = head
        # unlink list from the rest
        head.next = None

        return newHead
```

C++

```
/**
 * Definition of ListNode
 *
 * 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 new head of reversed linked list.
     */
    ListNode *reverse(ListNode *head) {
        // case1: empty list
        if (head == NULL) return head;
        // case2: only one element list
        if (head->next == NULL) return head;
        // case3: reverse from the rest after head
        ListNode *newHead = reverse(head->next);
```

```
        // reverse between head and head->next
        head->next->next = head;
        // unlink list from the rest
        head->next = NULL;

        return newHead;
    }
};
```

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverse(ListNode head) {
        // case1: empty list
        if (head == null) return head;
        // case2: only one element list
        if (head.next == null) return head;
        // case3: reverse from the rest after head
        ListNode newHead = reverse(head.next);
        // reverse between head and head->next
        head.next.next = head;
        // unlink list from the rest
        head.next = null;

        return newHead;
    }
}
```

源码分析

case1 和 case2 可以合在一起考虑，case3 返回的为新链表的头节点，整个递归过程中保持不变。

复杂度分析

递归嵌套层数为 $O(n)$, 时间复杂度为 $O(n)$, 空间(不含栈空间)复杂度为 $O(1)$.

Reference

- 全面分析再动手的习惯：链表的反转问题（递归和非递归方式） - 木棉和木槿 - 博客园
- data structures - Reversing a linked list in Java, recursively - Stack Overflow
- 反转单向链表的四种实现（递归与非递归，C++） | 宁心勉学，慎思笃行
- iteratively and recursively Java Solution - Leetcode Discuss

Reverse Linked List II

Source

- lintcode: (36) Reverse Linked List II

```
Reverse a linked list from position m to n.

Note
Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length of list.

Example
Given 1->2->3->4->5->NULL, m = 2 and n = 4, return 1->4->3->2->5->NULL.

Challenge
Reverse it in-place and in one-pass
```

题解

此题在上题的基础上加了位置要求，只翻转指定区域的链表。由于链表头节点不确定，祭出我们的dummy杀器。此题边界条件处理特别tricky，需要特别注意。

1. 由于只翻转指定区域，分析受影响的区域为第m-1个和第n+1个节点
2. 找到第m个节点，使用for循环n-m次，使用上题中的链表翻转方法
3. 处理第m-1个和第n+1个节点
4. 返回dummy->next

C++

```
/**
 * 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 head of linked list.
     * @param m: The start position need to reverse.
     * @param n: The end position need to reverse.
     * @return: The new head of partial reversed linked list.
```

```
     */
    ListNode *reverseBetween(ListNode *head, int m, int n) {
        if (head == NULL || m > n) {
            return NULL;
        }

        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode *node = dummy;

        for (int i = 1; i != m; ++i) {
            if (node == NULL) {
                return NULL;
            } else {
                node = node->next;
            }
        }

        ListNode *premNode = node;
        ListNode *mNode = node->next;
        ListNode *nNode = mNode, *postnNode = nNode->next;
        for (int i = m; i != n; ++i) {
            if (postnNode == NULL) {
                return NULL;
            }

            ListNode *temp = postnNode->next;
            postnNode->next = nNode;
            nNode = postnNode;
            postnNode = temp;
        }
        premNode->next = nNode;
        mNode->next = postnNode;

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

源码分析

1. 处理异常
2. 使用dummy辅助节点
3. 找到premNode——m节点之前的一个节点
4. 以nNode和postnNode进行遍历翻转，注意考虑在遍历到n之前postnNode可能为空
5. 连接premNode和nNode， `premNode->next = nNode;`
6. 连接mNode和postnNode， `mNode->next = postnNode;`

务必注意node 和node->next的区别！！，node指代节点，而 `node->next` 指代节点的下一连接。

Merge Two Sorted Lists

Source

- lintcode: (165) Merge Two Sorted Lists
- leetcode: Merge Two Sorted Lists | LeetCode OJ

```
Merge two sorted linked lists and return it as a new list.
The new list should be made by splicing together the nodes of the first two lists.

Example
Given 1->3->8->11->15->null, 2->null , return 1->2->3->8->11->15->null
```

题解

此题为两个链表的合并，合并后的表头节点不一定，故应联想到使用 `dummy` 节点。链表节点的插入主要涉及节点 `next` 指针值的改变，两个链表的合并操作则涉及到两个节点的 `next` 值变化，若每次合并一个节点都要改变两个节点 `next` 的值且要对 `NULL` 指针做异常处理，势必会异常麻烦。嗯，第一次做这个题时我就是这么想的... 下面看看相对较好的思路。

首先 `dummy` 节点还是必须要用到，除了 `dummy` 节点外还引入一个 `lastNode` 节点充当下一次合并时的头节点。在 `l1` 或者 `l2` 的某一个节点为空指针 `NULL` 时，退出 `while` 循环，并将非空链表的头部链接到 `lastNode->next` 中。

C++

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *lastNode = dummy;
        while ((NULL != l1) && (NULL != l2)) {
            if (l1->val < l2->val) {
                lastNode->next = l1;
                l1 = l1->next;
            } else {
                lastNode->next = l2;
                l2 = l2->next;
            }
```

```
            lastNode = lastNode->next;
        }

        // do not forget this line!
        lastNode->next =  (NULL != l1) ? l1 : l2;

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

源码分析

1. 异常处理，包含在 `dummy->next` 中。
2. 引入 `dummy` 和 `lastNode` 节点，此时 `lastNode` 指向的节点为 `dummy`
3. 对非空l1,l2循环处理，将l1/l2的较小者链接到 `lastNode->next` ，往后递推 `lastNode`
4. 最后处理l1/l2中某一链表为空退出while循环，将非空链表头链接到 `lastNode->next`
5. 返回 `dummy->next` ，即最终的首指针

注意 `lastNode` 的递推并不影响 `dummy->next` 的值，因为 `lastNode` 和 `dummy` 是两个不同的指针变量。

> 链表的合并为常用操作，务必非常熟练，以上的模板非常精炼，有两个地方需要记牢。1. 循环结束条件中为条件与操作；2. 最后处理 `lastNode->next` 指针的值。

复杂度分析

最好情况下，一个链表为空，时间复杂度为 $O(1)$. 最坏情况下， `lastNode` 遍历两个链表中的每一个节点，时间复杂度为 $O(l1+l2)$. 空间复杂度近似为 $O(1)$.

Reference

- Merge Two Sorted Lists | 九章算法

Merge k Sorted Lists

Source

- leetcode: Merge k Sorted Lists | LeetCode OJ
- lintcode: (104) Merge k Sorted Lists

题解1 - 选择归并(TLE)

参考 Merge Two Sorted Lists | Data Structure and Algorithm 中对两个有序链表的合并方法，这里我们也可以采用从 k 个链表中选择其中最小值的节点链接到 `lastNode->next` (和选择排序思路有点类似)，同时该节点所在的链表表头节点往后递推一个。直至 `lastNode` 遍历完 k 个链表的所有节点，此时表头节点均为 `NULL`，返回 `dummy->next`.

这种方法非常简单直接，但是时间复杂度较高，容易出现 TLE.

C++

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

class Solution {
public:
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.empty()) {
            return NULL;
        }

        ListNode *dummy = new ListNode(INT_MAX);
        ListNode *last = dummy;

        while (true) {
            int count = 0;
            int index = -1, tempVal = INT_MAX;
            for (int i = 0; i != lists.size(); ++i) {
                if (NULL == lists[i]) {
```

```
                    ++count;
                    if (count == lists.size()) {
                        last->next = NULL;
                        return dummy->next;
                    }
                    continue;
                }

                // choose the min value in non-NULL ListNode
                if (NULL != lists[i] && lists[i]->val <= tempVal) {
                    tempVal = lists[i]->val;
                    index = i;
                }
            }

            last->next = lists[index];
            last = last->next;
            lists[index] = lists[index]->next;
        }
    }
};
```

源码分析

1. 由于头节点不定，我们使用 `dummy` 节点。
2. 使用 `last` 表示每次归并后的新链表末尾节点。
3. `count` 用于累计链表表头节点为 `NULL` 的个数，若与 vector 大小相同则代表所有节点均已遍历完。
4. `tempVal` 用于保存每次比较 vector 中各链表表头节点中的最小值，`index` 保存本轮选择归并过程中最小值对应的链表索引，用于循环结束前递推该链表表头节点。

复杂度分析

由于每次 `for` 循环只能选择出一个最小值，总的时间复杂度最坏情况下为 $O(k \cdot \sum_{i=1}^{k} l_i)$. 空间复杂度近似为 $O(1)$.

题解2 - 迭代调用 `Merge Two Sorted Lists` (TLE)

鉴于题解1时间复杂度较高，题解2中我们可以反复利用时间复杂度相对较低的 Merge Two Sorted Lists | Data Structure and Algorithm. 即先合并链表1和2，接着将合并后的新链表再与链表3合并，如此反复直至 vector 内所有链表均已完全合并[soulmachine]。

C++

```
/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
```

```
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.empty()) {
            return NULL;
        }

        ListNode *head = lists[0];
        for (int i = 1; i != lists.size(); ++i) {
            head = merge2Lists(head, lists[i]);
        }

        return head;
    }

private:
    ListNode *merge2Lists(ListNode *left, ListNode *right) {
        ListNode *dummy = new ListNode(0);
        ListNode *last = dummy;

        while (NULL != left && NULL != right) {
            if (left->val < right->val) {
                last->next = left;
                left = left->next;
            } else {
                last->next = right;
                right = right->next;
            }
            last = last->next;
        }

        last->next = (NULL != left) ? left : right;

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

源码分析

实现合并两个链表的子方法后就没啥难度了， `mergeKLists` 中左半部分链表初始化为 `lists[0]` , `for` 循环后迭代归并 `head` 和 `lists[i]` .

复杂度分析

合并两个链表时最差时间复杂度为 $O(l_1 + l_2)$, 那么在以上的实现中总的时间复杂度可近似认为是

$l_1 + l_1 + l_2 + ... + l_1 + l_2 + ... + l_k = O(\sum_{i=1}^{k}(k-i) \cdot l_i)$. 比起题解1复杂度是要小一点，但量级上仍然差不太多。实际运行时间也证明了这一点，题解2的运行时间差不多时题解1的一半。那么还有没有进一步降低时间复杂度的可能呢？当然是有的，且看下题分解...

题解3 - 二分调用 `Merge Two Sorted Lists`

题解2中 `merge2Lists` 优化空间不大，那咱们就来看看 `mergeKLists` 中的 `for` 循环，仔细观察可得知第 `i` 个链表 l_i 被遍历了 $k-i$ 次，如果我们使用二分法对其进行归并呢？从中间索引处进行二分归并后，每个链表参与合并的次数变为 $\log k$, 故总的时间复杂度可降至 $\log k \cdot \sum_{i=1}^{k} l_i$. 优化幅度较大。

C++

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

class Solution {
public:
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.empty()) {
            return NULL;
        }

        return helper(lists, 0, lists.size() - 1);
    }

private:
    ListNode *helper(vector<ListNode *> &lists, int start, int end) {
        if (start == end) {
            return lists[start];
        } else if (start + 1 == end) {
            return merge2Lists(lists[start], lists[end]);
        }

        ListNode *left = helper(lists, start, start + (end - start) / 2);
        ListNode *right = helper(lists, start + (end - start) / 2 + 1, end);

        return merge2Lists(left, right);
    }
```

```
    ListNode *merge2Lists(ListNode *left, ListNode *right) {
        ListNode *dummy = new ListNode(0);
        ListNode *last = dummy;

        while (NULL != left && NULL != right) {
            if (left->val < right->val) {
                last->next = left;
                left = left->next;
            } else {
                last->next = right;
                right = right->next;
            }
            last = last->next;
        }
        last->next = (NULL != left) ? left : right;

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

源码分析

由于需要建立二分递归模型，另建一私有方法 `helper` 引入起止位置较为方便。下面着重分析 `helper` 。

1. 分两种边界条件处理，分别是 `start == end` 和 `start + 1 == end` . 虽然第二种边界条件可以略去，但是加上会节省递归调用的栈空间。
2. 使用分治思想理解 `helper` , `left` 和 `right` 的边界处理建议先分析几个简单例子，做到不重不漏。
3. 注意 `merge2Lists` 中传入的参数，为 `lists[start]` 而不是 `start` ...

在 `mergeKLists` 中调用 `helper` 时传入的 `end` 参数为 `lists.size() - 1` ，而不是 `lists.size()` .

复杂度分析

题解中已分析过，最坏的时间复杂度为 $\log k \cdot \sum_{i=1}^{k} l_i$, 空间复杂度近似为 $O(1)$.

优化后的运行时间显著减少！由题解2中的500+ms 减至40ms 以内。

Reference

- soulmachine. soulmachine的LeetCode 题解 ↩

Reorder List

Source

- leetcode: Reorder List | LeetCode OJ
- lintcode: (99) Reorder List

```
Given a singly linked list L: L0→L1→…→Ln-1→Ln,
reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…

You must do this in-place without altering the nodes' values.


Example
For example,
Given 1->2->3->4->null, reorder it to 1->4->2->3->null.
```

题解1 - 链表长度(TLE)

直观角度来考虑，如果把链表视为数组来处理，那么我们要做的就是依次将下标之和为 `n` 的两个节点链接到一块儿，使用两个索引即可解决问题，一个索引指向 `i` , 另一个索引则指向其之后的第 `n - 2*i` 个节点(对于链表来说实际上需要获取的是其前一个节点), 直至第一个索引大于第二个索引为止即处理完毕。

既然依赖链表长度信息，那么要做的第一件事就是遍历当前链表获得其长度喽。获得长度后即对链表进行遍历，小心处理链表节点的断开及链接。用这种方法会提示 TLE，也就是说还存在较大的优化空间！

C++ - TLE

```cpp
/**
 * Definition of ListNode
 * 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) {
        if (NULL == head || NULL == head->next || NULL == head->next->next) {
            return;
```

```
        }

        ListNode *last = head;
        int length = 0;
        while (NULL != last) {
            last = last->next;
            ++length;
        }

        last = head;
        for (int i = 1; i < length - i; ++i) {
            ListNode *beforeTail = last;
            for (int j = i; j < length - i; ++j) {
                beforeTail = beforeTail->next;
            }

            ListNode *temp = last->next;
            last->next = beforeTail->next;
            last->next->next = temp;
            beforeTail->next = NULL;
            last = temp;
        }
    }
};
```

源码分析

1. 异常处理，对于节点数目在两个以内的无需处理。
2. 遍历求得链表长度。
3. 遍历链表，第一个索引处的节点使用 `last` 表示，第二个索引处的节点的前一个节点使用 `beforeTail` 表示。
4. 处理链表的链接与断开，迭代处理下一个 `last` 。

复杂度分析

1. 遍历整个链表获得其长度，时间复杂度为 $O(n)$.
2. 双重 `for` 循环的时间复杂度为 $(n-2)+(n-4)+...+2=O(\frac{1}{2}\cdot n^2)$.
3. 总的时间复杂度可近似认为是 $O(n^2)$, 空间复杂度为常数。

> 使用这种方法务必注意 `i` 和 `j` 的终止条件，若取 `i < length + 1 - i` , 则在处理最后两个节点时会出现环，且尾节点会被删掉。在对节点进行遍历时务必注意保留头节点的信息！

题解2 - 反转链表后归并

既然题解1存在较大的优化空间，那我们该从哪一点出发进行优化呢？擒贼先擒王，题解1中时间复杂度最高的地方在于双重 `for` 循环，在对第二个索引进行遍历时， `j` 每次都从 `i` 处开始遍历，要是 `j` 能从链表尾部往前遍历该有多好啊！这样就能大大降低时间复杂度了，可惜本题的链表只是单向链表... 有什么特技可以在单向链表中进行反向遍历吗？还真有——反转链表！一语惊醒梦中人。

C++

```
/**
 * Definition of ListNode
 * 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) {
        if (NULL == head || NULL == head->next || NULL == head->next->next) {
            return;
        }

        ListNode *middle = findMiddle(head);
        ListNode *right = reverse(middle->next);
        middle->next = NULL;

        merge(head, right);
    }

private:
    void merge(ListNode *left, ListNode *right) {
        ListNode *dummy = new ListNode(0);
        while (NULL != left && NULL != right) {
            dummy->next = left;
            left = left->next;
            dummy = dummy->next;
            dummy->next = right;
            right = right->next;
            dummy = dummy->next;
        }

        dummy->next = (NULL != left) ? left : right;
        //delete dummy; /* bug, delete the tail node */
    }

    ListNode *reverse(ListNode *head) {
        ListNode *newHead = NULL;
        while (NULL != head) {
            ListNode *temp = head->next;
            head->next = newHead;
            newHead = head;
            head = temp;
        }

        return newHead;
    }
```

```
    ListNode *findMiddle(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return head;
        }

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

        return slow;
    }
};
```

源码分析

相对于题解1，题解2更多地利用了链表的常用操作如反转、找中点、合并。

1. 找中点：我在九章算法模板的基础上增加了对 `head->next` 的异常检测，增强了鲁棒性。
2. 反转：非常精炼的模板，记牢！
3. 合并：也可使用九章提供的模板，思想是一样的，需要注意 `left`，`right` 和 `dummy` 三者的赋值顺序，不能更改任何一步。
4. 对于 `new` 出的内存如何释放？代码中注释掉的为错误方法，你知道为什么吗？

复杂度分析

找中点一次，时间复杂度近似为 $O(n)$. 反转链表一次，时间复杂度近似为 $O(n/2)$. 合并左右链表一次，时间复杂度近似为 $O(n/2)$. 故总的时间复杂度为 $O(n)$.

Reference

- Reorder List | 九章算法

Copy List with Random Pointer

Source

- leetcode: Copy List with Random Pointer | LeetCode OJ
- lintcode: (105) Copy List with Random Pointer

```
A linked list is given such that each node contains an additional random pointer
which could point to any node in the list or null.

Return a deep copy of the list.
```

题解1 - 哈希表(两次遍历)

首先得弄懂深拷贝的含义，深拷贝可不是我们平时见到的对基本类型的变量赋值那么简单，深拷贝常常用于对象的克隆。这道题要求**深度拷贝**一个带有 random 指针的链表，random 可能指向空，也可能指向链表中的任意一个节点。

对于通常的单向链表，我们依次遍历并根据原链表的值生成新节点即可，原链表的所有内容便被复制了一份。但由于此题中的链表不只是有 next 指针，还有一个随机指针，故除了复制通常的 next 指针外还需维护新链表中的随机指针。容易混淆的地方在于原链表中的随机指针指向的是原链表中的节点，深拷贝则要求将随机指针指向新链表中的节点。

所有类似的**深度拷贝**题目的传统做法，都是维护一个 `hash table` 。即先按照复制一个正常链表的方式复制，复制的时候把复制的结点做一个 `hash table` ，以旧结点为 key，新节点为 value。这么做的目的是为了第二遍扫描的时候我们按照这个哈希表把结点的 random 指针接上。

原链表和深拷贝之后的链表如下：

```
|------------|             |------------|
|            v       ===>  |            v
1  --> 2 --> 3 --> 4       1' --> 2'--> 3'--> 4'
```

深拷贝步骤如下；

1. 根据 next 指针新建链表
2. 维护新旧节点的映射关系
3. 拷贝旧链表中的 random 指针
4. 更新新链表中的 random 指针

其中1, 2, 3 可以合并在一起。

一图胜千文

Python

```
# Definition for singly-linked list with a random pointer.
# class RandomListNode:
#     def __init__(self, x):
#         self.label = x
#         self.next = None
#         self.random = None
class Solution:
    # @param head: A RandomListNode
    # @return: A RandomListNode
    def copyRandomList(self, head):
        dummy = RandomListNode(0)
        curNode = dummy
        randomMap = {}

        while head is not None:
            # link newNode to new List
            newNode = RandomListNode(head.label)
            curNode.next = newNode
            # map old node head to newNode
            randomMap[head] = newNode
            # copy old node random pointer
            newNode.random = head.random
            #
            head = head.next
            curNode = curNode.next

        # re-mapping old random node to new node
        curNode = dummy.next
        while curNode is not None:
            if curNode.random is not None:
                curNode.random = randomMap[curNode.random]
            curNode = curNode.next

        return dummy.next
```

C++

```
/**
 * 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:
    /**
     * @param head: The head of linked list with a random pointer.
     * @return: A new head of a deep copy of the list.
     */
    RandomListNode *copyRandomList(RandomListNode *head) {
        if (head == NULL) return NULL;

        RandomListNode *dummy = new RandomListNode(0);
        RandomListNode *curNode = dummy;
        unordered_map<RandomListNode *, RandomListNode *> randomMap;
        while(head != NULL) {
            // link newNode to new List
            RandomListNode *newNode = new RandomListNode(head->label);
            curNode->next = newNode;
            // map old node head to newNode
            randomMap[head] = newNode;
            // copy old node random pointer
            newNode->random = head->random;

            head = head->next;
            curNode = curNode->next;
        }

        // re-mapping old random node to new node
        curNode = dummy->next;
        while (curNode != NULL) {
            if (curNode->random != NULL) {
                curNode->random = randomMap[curNode->random];
            }
            curNode = curNode->next;
        }

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

Java

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
```

```
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    /**
     * @param head: The head of linked list with a random pointer.
     * @return: A new head of a deep copy of the list.
     */
    public RandomListNode copyRandomList(RandomListNode head) {
        if (head == null) return null;

        RandomListNode dummy = new RandomListNode(0);
        RandomListNode curNode = dummy;
        HashMap<RandomListNode, RandomListNode> randomMap = new HashMap<RandomListNode, R
        while (head != null) {
            // link newNode to new List
            RandomListNode newNode = new RandomListNode(head.label);
            curNode.next = newNode;
            // map old node head to newNode
            randomMap.put(head, newNode);
            // copy old node random pointer
            newNode.random = head.random;
            //
            head = head.next;
            curNode = curNode.next;
        }

        // re-mapping old random node to new node
        curNode = dummy.next;
        while(curNode != null) {
            if (curNode.random != null) {
                curNode.random = randomMap.get(curNode.random);
            }
            curNode = curNode.next;
        }

        return dummy.next;
    }
}
```

源码分析

1. 只需要一个 `dummy` 存储新的拷贝出来的链表头，以用来第二次遍历时链接 random 指针。所以第一句异常检测可有可无。
2. 第一次链接时勿忘记同时拷贝 random 指针，但此时的 random 指针并没有真正“链接”上，实际上是链接到了原始链表的 node 上。
3. 第二次遍历是为了把原始链表的被链接的 node 映射到新链表中的 node，从而完成真正“链接”。

复杂度分析

总共要进行两次扫描，所以时间复杂度是 $O(2n) = O(n)$, 在链表较长时可能会 TLE(比如 Python). 空间上需要一个哈希表来做结点的映射，所以空间复杂度也是 $O(n)$.

题解2 - 哈希表(一次遍历)

从题解1 的分析中我们可以看到对于 random 指针我们是在第二次遍历时单独处理的，那么在借助哈希表的情况下有没有可能一次遍历就完成呢？我们回想一下题解1 中random 节点的处理，由于在第一次遍历完之前 random 所指向的节点是不知道到底是指向哪一个节点，故我们在将 random 指向的节点加入哈希表之前判断一次就好了(是否已经生成，避免对同一个值产生两个不同的节点)。由于 random 节点也在第一次遍历加入哈希表中，故生成新节点时也需要判断哈希表中是否已经存在。

Python

```
# Definition for singly-linked list with a random pointer.
# class RandomListNode:
#     def __init__(self, x):
#         self.label = x
#         self.next = None
#         self.random = None
class Solution:
    # @param head: A RandomListNode
    # @return: A RandomListNode
    def copyRandomList(self, head):
        dummy = RandomListNode(0)
        curNode = dummy
        hash_map = {}

        while head is not None:
            # link newNode to new List
            if head in hash_map.keys():
                newNode = hash_map[head]
            else:
                newNode = RandomListNode(head.label)
                hash_map[head] = newNode
            curNode.next = newNode
            # map old node head to newNode
            hash_map[head] = newNode
            # copy old node random pointer
            if head.random is not None:
                if head.random in hash_map.keys():
                    newNode.random = hash_map[head.random]
                else:
                    newNode.random = RandomListNode(head.random.label)
                    hash_map[head.random] = newNode.random
            #
            head = head.next
            curNode = curNode.next

        return dummy.next
```

C++

```
/**
 * 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:
    /**
     * @param head: The head of linked list with a random pointer.
     * @return: A new head of a deep copy of the list.
     */
    RandomListNode *copyRandomList(RandomListNode *head) {
        RandomListNode *dummy = new RandomListNode(0);
        RandomListNode *curNode = dummy;
        unordered_map<RandomListNode *, RandomListNode *> hash_map;
        while(head != NULL) {
            // link newNode to new List
            RandomListNode *newNode = NULL;
            if (hash_map.count(head) > 0) {
                newNode = hash_map[head];
            } else {
                newNode = new RandomListNode(head->label);
                hash_map[head] = newNode;
            }
            curNode->next = newNode;
            // re-mapping old random node to new node
            if (head->random != NULL) {
                if (hash_map.count(head->random) > 0) {
                    newNode->random = hash_map[head->random];
                } else {
                    newNode->random = new RandomListNode(head->random->label);
                    hash_map[head->random] = newNode->random;
                }
            }

            head = head->next;
            curNode = curNode->next;
        }

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

Java

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    /**
     * @param head: The head of linked list with a random pointer.
     * @return: A new head of a deep copy of the list.
```

```
     */
    public RandomListNode copyRandomList(RandomListNode head) {
        RandomListNode dummy = new RandomListNode(0);
        RandomListNode curNode = dummy;
        HashMap<RandomListNode, RandomListNode> hash_map = new HashMap<RandomListNode, Ra
        while (head != null) {
            // link newNode to new List
            RandomListNode newNode = null;
            if (hash_map.containsKey(head)) {
                newNode = hash_map.get(head);
            } else {
                newNode = new RandomListNode(head.label);
                hash_map.put(head, newNode);
            }
            curNode.next = newNode;
            // re-mapping old random node to new node
            if (head.random != null) {
                if (hash_map.containsKey(head.random)) {
                    newNode.random = hash_map.get(head.random);
                } else {
                    newNode.random = new RandomListNode(head.random.label);
                    hash_map.put(head.random, newNode.random);
                }
            }
            //
            head = head.next;
            curNode = curNode.next;
        }

        return dummy.next;
    }
}
```

源码分析

随机指针指向节点不定，故加入哈希表之前判断一下 key 是否存在即可。C++ 中 C++ 11 引入的 unordered_map 较 map 性能更佳，使用 count 判断 key 是否存在比 find 开销小一点，因为 find 需要构造 iterator。

复杂度分析

遍历一次原链表，判断哈希表中 key 是否存在，故时间复杂度为 $O(n)$, 空间复杂度为 $O(n)$.

题解3 - 间接使用哈希表

上面的解法很显然，需要额外的空间。这个额外的空间是由 `hash table` 的维护造成的。因为当我们访问一个结点时可能它的 random 指针指向的结点还没有访问过，结点还没有创建，所以需要用 `hash table` 的额外线性空间维护。

但我们可以通过链表原来结构中的 `next` 指针来替代 `hash table` 做哈希。假设有如下链表：

```
|------------|
|            v
1  --> 2 --> 3 --> 4
```

节点1的 random 指向了3。首先我们可以通过 next 遍历链表，依次拷贝节点，并将其添加到原节点后面，如下：

```
|--------------------------|
|                          v
1  --> 1' --> 2 --> 2' --> 3 --> 3' --> 4 --> 4'
       |                  ^
       |------------------|
```

因为我们只是简单的复制了 random 指针，所以新的节点的 random 指向的仍然是老的节点，譬如上面的1和1'都是指向的3。

调整新的节点的 random 指针，对于上面例子来说，我们需要将1'的 random 指向3'，其实也就是原先 random 指针的next节点。

```
|--------------------------|
|                          v
1  --> 1' --> 2 --> 2' --> 3 --> 3' --> 4 --> 4'
       |                        ^
       |------------------------|
```

最后，拆分链表，就可以得到深度拷贝的链表了。

总结起来，实际我们对链表进行了三次扫描，第一次扫描对每个结点进行复制，然后把复制出来的新节点接在原结点的 next 指针上，也就是让链表变成一个重复链表，就是新旧更替；第二次扫描中我们把旧结点的随机指针赋给新节点的随机指针，因为新结点都跟在旧结点的下一个，所以赋值比较简单，就是 `node->next->random = node->random->next` ，其中 `node->next` 就是新结点，因为第一次扫描我们就是把新结点接在旧结点后面。现在我们把结点的随机指针都接好了，最后一次扫描我们把链表拆成两个，第一个还原原链表，而第二个就是我们要求的复制链表。因为现在链表是旧新更替，只要把每隔两个结点分别相连，对链表进行分割即可。

Python

```python
# Definition for singly-linked list with a random pointer.
# class RandomListNode:
#     def __init__(self, x):
#         self.label = x
#         self.next = None
#         self.random = None
class Solution:
    # @param head: A RandomListNode
    # @return: A RandomListNode
    def copyRandomList(self, head):
        if head is None:
```

```
            return None

        curr = head
        # step1: generate new List with node
        while curr is not None:
            newNode = RandomListNode(curr.label)
            newNode.next = curr.next
            curr.next = newNode
            curr = curr.next.next

        # step2: copy random pointer
        curr = head
        while curr is not None:
            if curr.random is not None:
                curr.next.random = curr.random.next
            curr = curr.next.next
        # step3: split original and new List
        newHead = head.next
        curr = head
        while curr is not None:
            newNode = curr.next
            curr.next = curr.next.next
            if newNode.next is not None:
                newNode.next = newNode.next.next
            curr = curr.next

        return newHead
```

C++

```
/**
 * 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:
    /**
     * @param head: The head of linked list with a random pointer.
     * @return: A new head of a deep copy of the list.
     */
    RandomListNode *copyRandomList(RandomListNode *head) {
        if (head == NULL) return NULL;

        RandomListNode *curr = head;
        // step1: generate new List with node
        while (curr != NULL) {
            RandomListNode *newNode = new RandomListNode(curr->label);
            newNode->next = curr->next;
            curr->next = newNode;
            //
            curr = curr->next->next;
        }
        // step2: copy random
```

```
        curr = head;
        while (curr != NULL) {
            if (curr->random != NULL) {
                curr->next->random = curr->random->next;
            }
            curr = curr->next->next;
        }
        // step3: split original and new List
        RandomListNode *newHead = head->next;
        curr = head;
        while (curr != NULL) {
            RandomListNode *newNode = curr->next;
            curr->next = curr->next->next;
            curr = curr->next;
            if (newNode->next != NULL) {
                newNode->next = newNode->next->next;
            }
        }

        return newHead;
    }
};
```

Java

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    /**
     * @param head: The head of linked list with a random pointer.
     * @return: A new head of a deep copy of the list.
     */
    public RandomListNode copyRandomList(RandomListNode head) {
        if (head == null) return null;

        RandomListNode curr = head;
        // step1: generate new List with node
        while (curr != null) {
            RandomListNode newNode = new RandomListNode(curr.label);
            newNode.next = curr.next;
            curr.next = newNode;
            //
            curr = curr.next.next;
        }
        // step2: copy random pointer
        curr = head;
        while (curr != null) {
            if (curr.random != null) {
                curr.next.random = curr.random.next;
            }
            curr = curr.next.next;
```

```
        }
        // step3: split original and new List
        RandomListNode newHead = head.next;
        curr = head;
        while (curr != null) {
            RandomListNode newNode = curr.next;
            curr.next = curr.next.next;
            curr = curr.next;
            if (newNode.next != null) {
                newNode.next = newNode.next.next;
            }
        }

        return newHead;
    }
}
```

源码分析

注意指针使用前需要判断是否非空，迭代时注意是否前进两步，即 `.next.next`

复杂度分析

总共进行三次线性扫描，所以时间复杂度是 $O(n)$。但不再需要额外空间的 `hash table`，所以空间复杂度是 $O(1)$。

Reference

- Copy List with Random Pointer - siddontang's leetcode Solution Book
- Copy List with Random Pointer 参考程序 Java/C++/Python
- Copy List with Random Pointer - Code Ganker

Sort List

Source

- leetcode: Sort List | LeetCode OJ
- lintcode: (98) Sort List

```
Sort a linked list in O(n log n) time using constant space complexity.
```

题解1 - 归并排序(链表长度求中间节点)

链表的排序操作，对于常用的排序算法，能达到 $O(n \log n)$的复杂度有快速排序(平均情况)，归并排序，堆排序。快速排序不一定能保证其时间复杂度一定满足要求，归并排序和堆排序都能满足复杂度的要求。在数组排序中，归并排序通常需要使用 $O(n)$ 的额外空间，也有原地归并的实现，代码写起来略微麻烦一点。但是对于链表这种非随机访问数据结构，所谓的「排序」不过是指针 `next` 值的变化而已，主要通过指针操作，故仅需要常数级别的额外空间，满足题意。堆排序通常需要构建二叉树，在这道题中不太适合。

既然确定使用归并排序，我们就来思考归并排序实现的几个要素。

1. 按长度等分链表，归并虽然不严格要求等分，但是等分能保证线性对数的时间复杂度。由于链表不能随机访问，故可以先对链表进行遍历求得其长度。
2. 合并链表，细节已在 Merge Two Sorted Lists | Data Structure and Algorithm 中详述。

在按长度等分链表时进行「后序归并」——先求得左半部分链表的表头，再求得右半部分链表的表头，最后进行归并操作。

由于递归等分链表的操作需要传入链表长度信息，故需要另建一辅助函数。新鲜出炉的源码如下。

```
/**
 * Definition of ListNode
 * 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: You should return the head of the sorted linked list,
                    using constant space complexity.
     */
    ListNode *sortList(ListNode *head) {
```

```
        if (NULL == head) {
            return NULL;
        }

        // get the length of List
        int len = 0;
        ListNode *node = head;
        while (NULL != node) {
            node = node->next;
            ++len;
        }

        return sortListHelper(head, len);
    }

private:
    ListNode *sortListHelper(ListNode *head, const int length) {
        if ((NULL == head) || (0 >= length)) {
            return head;
        }

        ListNode *midNode = head;

        int count = 1;
        while (count < length / 2) {
            midNode = midNode->next;
            ++count;
        }

        ListNode *rList = sortListHelper(midNode->next, length - length / 2);
        midNode->next = NULL;
        ListNode *lList = sortListHelper(head, length / 2);

        return mergeList(lList, rList);
    }

    ListNode *mergeList(ListNode *l1, ListNode *l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *lastNode = dummy;
        while ((NULL != l1) && (NULL != l2)) {
            if (l1->val < l2->val) {
                lastNode->next = l1;
                l1 = l1->next;
            } else {
                lastNode->next = l2;
                l2 = l2->next;
            }

            lastNode = lastNode->next;
        }

        lastNode->next = (NULL != l1) ? l1 : l2;

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

源码分析

1. 归并子程序没啥好说的了，见 Merge Two Sorted Lists | Data Structure and Algorithm.
2. 在递归处理链表长度时，分析方法和 Convert Sorted List to Binary Search Tree | Data Structure and Algorithm 一致，**`count` 表示遍历到链表中间时表头指针需要移动的节点数。**在纸上分析几个简单例子后即可确定，由于这个题需要的是「左右」而不是二叉搜索树那道题需要三分——「左中右」，故将 `count` 初始化为1更为方便，左半部分链表长度为 `length / 2`，这两个值的确定最好是先用纸笔分析再视情况取初值，不可死记硬背。
3. 找到中间节点后首先将其作为右半部分链表处理，然后将其 `next` 值置为 `NULL`，否则归并子程序无法正确求解。这里需要注意的是 `midNode` 是左半部分的最后一个节点，`midNode->next` 才是链表右半部分的起始节点。
4. 递归模型中**左、右、合并**三者的顺序可以根据分治思想确定，即先找出左右链表，最后进行归并(因为归并排序的前提是两个子链表各自有序)。

复杂度分析

遍历求得链表长度，时间复杂度为 $O(n)$, 「折半取中」过程中总共有 $\log(n)$ 层，每层找中点需遍历 $n/2$ 个节点，故总的时间复杂度为 $n/2 \cdot O(\log n)$ (折半取中), 每一层归并排序的时间复杂度介于 $O(n/2)$ 和 $O(n)$之间，故总的时间复杂度为 $O(n \log n)$, 空间复杂度为常数级别，满足题意。

题解2 - 归并排序(快慢指针求中间节点)

除了遍历链表求得总长外，还可使用看起来较为巧妙的技巧如「快慢指针」，快指针每次走两步，慢指针每次走一步，最后慢指针所指的节点即为中间节点。使用这种特技的关键之处在于如何正确确定快慢指针的起始位置。

C++

```
/**
 * Definition of ListNode
 * 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: You should return the head of the sorted linked list,
                    using constant space complexity.
     */
    ListNode *sortList(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return head;
```

```
        }

        ListNode *midNode = findMiddle(head);
        ListNode *rList = sortList(midNode->next);
        midNode->next = NULL;
        ListNode *lList = sortList(head);

        return mergeList(lList, rList);
    }

private:
    ListNode *findMiddle(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return head;
        }

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

        return slow;
    }

    ListNode *mergeList(ListNode *l1, ListNode *l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *lastNode = dummy;
        while ((NULL != l1) && (NULL != l2)) {
            if (l1->val < l2->val) {
                lastNode->next = l1;
                l1 = l1->next;
            } else {
                lastNode->next = l2;
                l2 = l2->next;
            }

            lastNode = lastNode->next;
        }

        lastNode->next = (NULL != l1) ? l1 : l2;

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

源码分析

1. 异常处理不仅考虑了 `head` , 还考虑了 `head->next` , 可减少辅助程序中的异常处理。
2. 使用快慢指针求中间节点时，将 `fast` 初始化为 `head->next` 可有效避免无法分割两个节点如 `1->2->null` [fast_slow_pointer]。
    - 求中点的子程序也可不做异常处理，但前提是主程序 `sortList` 中对 `head->next` 做了检测。
3. 最后进行 `merge` 归并排序。

> 在递归和迭代程序中，需要尤其注意终止条件的确定，以及循环语句中变量的自增，以防出现死循环

> 或访问空指针。

复杂度分析

同上。

Reference

- Sort List | 九章算法
- > [fast_slow_pointer]. LeetCode: Sort List 解题报告 - Yu's Garden - 博客园 ↩

Insertion Sort List

Source

- leetcode: Insertion Sort List | LeetCode OJ
- lintcode: (173) Insertion Sort List

```
Sort a linked list using insertion sort.

Example
Given 1->3->2->0->null, return 0->1->2->3->null.
```

题解1 - 从首到尾遍历

插入排序常见的实现是针对数组的，如前几章总的的 Insertion Sort，但这道题中的排序的数据结构为单向链表，故无法再从后往前遍历比较值的大小了。好在天无绝人之路，我们还可以**从前往后依次遍历比较和交换**。

由于排序后头节点不一定，故需要引入 dummy 大法，并以此节点的 `next` 作为最后返回结果的头节点，返回的链表从 `dummy->next` 这里开始构建。首先我们每次都从 `dummy->next` 开始遍历，依次和上一轮处理到的节点的值进行比较，直至找到不小于上一轮节点值的节点为止，随后将上一轮节点插入到当前遍历的节点之前，依此类推。文字描述起来可能比较模糊，大家可以结合以下的代码在纸上分析下。

Python

```python
"""
Definition of ListNode
class ListNode(object):

    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of linked list.
    @return: The head of linked list.
    """
    def insertionSortList(self, head):
        dummy = ListNode(0)
        cur = head
        while cur is not None:
            pre = dummy
            while pre.next is not None and pre.next.val < cur.val:
                pre = pre.next
            temp = cur.next
            cur.next = pre.next
            pre.next = cur
```

```
            cur = temp
        return dummy.next
```

C++

```
/**
 * Definition of ListNode
 * 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 head of linked list.
     */
    ListNode *insertionSortList(ListNode *head) {
        ListNode *dummy = new ListNode(0);
    ListNode *cur = head;
        while (cur != NULL) {
            ListNode *pre = dummy;
            while (pre->next != NULL && pre->next->val < cur->val) {
                pre = pre->next;
            }
            ListNode *temp = cur->next;
            cur->next = pre->next;
            pre->next = cur;
            cur = temp;
        }

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

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode dummy = new ListNode(0);
```

```
        ListNode cur = head;
        while (cur != null) {
            ListNode pre = dummy;
            while (pre.next != null && pre.next.val < cur.val) {
                pre = pre.next;
            }
            ListNode temp = cur.next;
            cur.next = pre.next;
            pre.next = cur;
            cur = temp;
        }

        return dummy.next;
    }
}
```

源码分析

1. 新建 dummy 节点，用以处理最终返回结果中头节点不定的情况。
2. 以 `cur` 表示当前正在处理的节点，在从 dummy 开始遍历前保存 `cur` 的下一个节点作为下一轮的 `cur` .
3. 以 `pre` 作为遍历节点，直到找到不小于 `cur` 值的节点为止。
4. 将 `pre` 的下一个节点 `pre->next` 链接到 `cur->next` 上， `cur` 链接到 `pre->next` , 最后将 `cur` 指向下一个节点。
5. 返回 `dummy->next` 最为最终头节点。

Python 的实现在 lintcode 上会提示 TLE, leetcode 上勉强通过，这里需要注意的是采用 `if A is not None:` 的效率要比 `if A:` 高，不然 leetcode 上也过不了。具体原因可参考 Stack Overflow 上的讨论。

复杂度分析

最好情况：原链表已经有序，每得到一个新节点都需要 i 次比较和一次交换, 时间复杂度为 $1/2O(n^2) + O(n)$, 使用了 dummy 和 pre, 空间复杂度近似为 $O(1)$.

最坏情况：原链表正好逆序，由于是单向链表只能从前往后依次遍历，交换和比较次数均为 $1/2O(n^2)$, 总的时间复杂度近似为 $O(n^2)$, 空间复杂度同上，近似为 $O(1)$.

题解2 - 优化有序链表

从题解1的复杂度分析可以看出其在最好情况下时间复杂度都为 $O(n^2)$ ，这显然是需要优化的。 仔细观察可发现最好情况下的比较次数 是可以优化到 $O(n)$ 的。思路自然就是先判断链表是否有序，仅对降序的部分进行处理。优化之后的代码就没题解1那么容易写对了，建议画个图自行纸上分析下。

Python

```
"""
Definition of ListNode
class ListNode(object):
```

```
    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of linked list.
    @return: The head of linked list.
    """
    def insertionSortList(self, head):
        dummy = ListNode(0)
        dummy.next = head
        cur = head
        while cur is not None:
            if cur.next is not None and cur.next.val < cur.val:
                # find insert position for smaller(cur->next)
                pre = dummy
                while pre.next is not None and pre.next.val < cur.next.val:
                    pre = pre.next
                # insert cur->next after pre
                temp = pre.next
                pre.next = cur.next
                cur.next = cur.next.next
                pre.next.next = temp
            else:
                cur = cur.next
        return dummy.next
```

C++

```
/**
 * Definition of ListNode
 * 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 head of linked list.
     */
    ListNode *insertionSortList(ListNode *head) {
        ListNode *dummy = new ListNode(0);
        dummy->next = head;
    ListNode *cur = head;
        while (cur != NULL) {
            if (cur->next != NULL && cur->next->val < cur->val) {
                ListNode *pre = dummy;
                // find insert position for smaller(cur->next)
                while (pre->next != NULL && pre->next->val <= cur->next->val) {
```

```
                    pre = pre->next;
                }
                // insert cur->next after pre
                ListNode *temp = pre->next;
                pre->next = cur->next;
                cur->next = cur->next->next;
                pre->next->next = temp;
            } else {
                cur = cur->next;
            }
        }

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

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode cur = head;
        while (cur != null) {
            if (cur.next != null && cur.next.val < cur.val) {
                // find insert position for smaller(cur->next)
                ListNode pre = dummy;
                while (pre.next != null && pre.next.val < cur.next.val) {
                    pre = pre.next;
                }
                // insert cur->next after pre
                ListNode temp = pre.next;
                pre.next = cur.next;
                cur.next = cur.next.next;
                pre.next.next = temp;
            } else {
                cur = cur.next;
            }
        }

        return dummy.next;
    }
}
```

源码分析

1. 新建 dummy 节点并将其 `next` 指向 `head`

2. 分情况讨论，仅需要处理逆序部分。
3. 由于已经确认链表逆序，故仅需将较小值(`cur->next` 而不是 `cur`)的节点插入到链表的合适位置。
4. 将 `cur->next` 插入到 `pre` 之后，这里需要四个步骤，需要特别小心！

如上图所示，将 `cur->next` 插入到 `pre` 节点后大致分为3个步骤。

复杂度分析

最好情况下时间复杂度降至 $O(n)$, 其他同题解1.

Reference

- Explained C++ solution (24ms) - Leetcode Discuss
- Insertion Sort List - 九章算法

Check if a singly linked list is palindrome

- tags: [palindrome, linked_list]

Source

- Function to check if a singly linked list is palindrome - GeeksforGeeks

```
Given a singly linked list of characters, write a function that
returns true if the given list is palindrome, else false.
```

题解1 - 使用辅助栈

根据栈的特性(FILO)，可以首先遍历链表并入栈(最后访问栈时则反过来了)，随后再次遍历链表并比较当前节点和栈顶元素，若比较结果完全相同则为回文。 又根据回文的特性，实际上还可以只遍历链表前半部分节点，再用栈中的元素和后半部分元素进行比较，分链表节点个数为奇数或者偶数考虑即可。由于链表长度未知，因此可以考虑使用快慢指针求得。

Java

```java
/**
 * Definition for singly-linked list.
 */
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class Solution {
    public static boolean isPalindrome(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        Stack<Integer> stack = new Stack<Integer>();

        // push node before mid
        while (fast != null && fast.next != null) {
            stack.push(slow.val);
            slow = slow.next;
            fast = fast.next.next;
        }

        // skip mid node for odd size
        if (fast != null) {
            slow = slow.next;
        }

        while (slow != null) {
            int top = stack.pop();
```

```
            // compare top with slow.val
            if (top != slow.val) {
                return false;
            }
            slow = slow.next;
        }

        return true;
    }

    public static void main (String[] args) {
        int len = 9;
        ListNode head = new ListNode(0);
        ListNode node = head;
        for (int i = 1; i < 9; i++) {
            int temp = (i >= len / 2) ? (len - i - 1) : i;
            node.next = new ListNode(temp);
            node = node.next;
        }

        System.out.println(isPalindrome(head));
    }
}
```

源码分析

注意区分好链表中个数为奇数还是偶数就好了，举几个简单例子辅助分析。

复杂度分析

使用了栈作为辅助空间，空间复杂度为 $O(\frac{1}{2}n)$, 分别遍历链表的前半部分和后半部分，时间复杂度为 $O(n)$.

题解2 - 原地翻转

题解 1 的解法使用了辅助空间，在可以改变原来的链表的基础上，可使用原地翻转，思路为翻转前半部分，然后迭代比较。具体可分为以下四个步骤。

1. 找中点。
2. 翻转链表的后半部分。
3. 逐个比较前后部分节点值。
4. 链表复原，翻转后半部分链表。

Java

```
/**
 * Definition for singly-linked list.
 */
class ListNode {
    int val;
    ListNode next;
```

```
    ListNode(int x) { val = x; }
}

public class Solution {
    public static boolean isPalindrome(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        // push node before mid
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        // skip mid node for odd number
        if (fast != null) {
            slow = slow.next;
        }

        ListNode rightHead = reverse(slow);
        ListNode rCurr = rightHead;
        ListNode lCurr = head;
        while (rCurr != null) {
            if (rCurr.val != lCurr.val) {
                return false;
            }
            lCurr = lCurr.next;
            rCurr = rCurr.next;
        }
        // recover list
        rightHead = reverse(rightHead);

        return true;
    }

    public static ListNode reverse (ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode temp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = temp;
        }

        return prev;
    }

    public static void main (String[] args) {
        int len = 9;
        ListNode head = new ListNode(0);
        ListNode node = head;
        for (int i = 1; i < 9; i++) {
            int temp = (i >= len / 2) ? (len - i - 1) : i;
            node.next = new ListNode(temp);
            node = node.next;
        }

        System.out.println(isPalindrome(head));
    }
}
```

源码分析

连续翻转两次右半部分链表即可复原原链表，将一些功能模块如翻转等尽量模块化。

复杂度分析

遍历链表若干次，时间复杂度近似为 $O(n)$, 使用了几个临时遍历，空间复杂度为 $O(1)$.

题解3 - 递归

递归需要两个重要条件，递归步的建立和递归终止条件。对于回文比较，理所当然应该递归比较第 i 个节点和第 n-i 个节点，那么问题来了，如何构建这个递归步？大致可以猜想出来递归的传入参数应该包含两个节点，用以指代第 i 个节点和第 n-i 个节点。返回参数应该包含布尔值(用以提前返回不是回文的情况)和左半部分节点的下一个节点(用以和右半部分的节点进行比较)。由于需要返回两个值，在 Java 中需要使用自定义类进行封装，C/C++ 中则可以使用指针改变在**递归调用后**进行比较时节点的值。

Java

```
/**
 * Definition for singly-linked list.
 */
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class Solution {
    private class Result {
        ListNode node;
        boolean isp;
        Result(ListNode aNode, boolean ret) {
               isp = ret;
               node = aNode;
               }
    }

    public Result helper(ListNode left, ListNode right) {
        Result result = new Result(left, true);

        if (right == null) return result;

        result = helper(left, right.next);
        boolean isp = (right.val == result.node.val);
        if (!isp) {
            result.isp = false;
        }
        result.node = result.node.next;

        return result;
    }
```

```
    public boolean isPalindrome(ListNode head) {
        Result ret = helper(head, head);
        return ret.isp;
    }

    public static void main (String[] args) {
        int len = 9;
        ListNode head = new ListNode(0);
        ListNode node = head;
        for (int i = 1; i < 9; i++) {
            int temp = (i >= len / 2) ? (len - i - 1) : i;
            node.next = new ListNode(temp);
            node = node.next;
        }

        Solution ret = new Solution();
        System.out.println(ret.isPalindrome(head));
    }
}
```

源码分析

核心代码为返回 Result 复合数据类型部分，返回 result 后在返回最终结果之前需要执行 `result.node = result.node.next`，左半部分节点往后递推，用以返回给上层回调用。

复杂度分析

递归调用 n 层，时间复杂度近似为 $O(n)$, 使用了几个临时变量，空间复杂度为 $O(1)$.

Reference

- Function to check if a singly linked list is palindrome - GeeksforGeeks
- 回文判断 | The-Art-Of-Programming-By-July/01.04.md
- ctci/QuestionB.java at master · gaylemcd/ctci

Delete Node in the Middle of Singly Linked List

Source

- lintcode: (372) Delete Node in the Middle of Singly Linked List

```
Implement an algorithm to delete a nodein the middle of a singly linked list,
given only access to that node.

Example
Given 1->2->3->4, and node 3. return 1->2->4
```

题解

根据给定的节点并删除这个节点。弄清楚题意很重要，我首先以为是删除链表的中间节点。:(一般来说删除单向链表中的一个节点需要首先知道节点的前一个节点，改变其指向的下一个节点并删除就可以了。但是从这道题来看无法知道欲删除节点的前一个节点，那么也就是意味着无法改变前一个节点指向的下一个节点，强行删除当前节点将导致非法内存访问。

既然找不到前一个节点，那么也就意味着不能用通常的方法删除给定节点。从实际角度来看，我们关心的往往并不是真的删除了链表中的某个节点，而是访问链表时表现的行为就像是某个节点被删除了一样。这种另类『删除』方法就是——使用下一个节点的值覆盖当前节点的值，删除下一个节点。

Java

```java
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param node: the node in the list should be deleted
     * @return: nothing
     */
    public void deleteNode(ListNode node) {
        if (node == null) return;
        if (node.next == null) node = null;

        node.val = node.next.val;
        node.next = node.next.next;
    }
```

```
}
```

源码分析

注意好边界条件处理即可。

复杂度分析

略。$O(1)$.

Maximum Depth of Binary Tree# Binary Tree - 二叉树

二叉树的基本概念在 Binary Tree | Algorithm 中有简要的介绍，这里就二叉树的一些应用做一些实战演练。

二叉树的遍历大致可分为前序、中序、后序三种方法。

Binary Tree Preorder Traversal

Source

- leetcode: Binary Tree Preorder Traversal | LeetCode OJ
- lintcode: (66) Binary Tree Preorder Traversal

```
Given a binary tree, return the preorder traversal of its nodes' values.

Note
Given binary tree {1,#,2,3},

   1
    \
     2
    /
   3

return [1,2,3].

Example
Challenge
Can you do it without recursion?
```

题解1 - 递归

面试时不推荐递归这种做法。

递归版很好理解，首先判断当前节点(根节点)是否为 `null` ，是则返回空vector，否则先返回当前节点的值，然后对当前节点的左节点递归，最后对当前节点的右节点递归。递归时对返回结果的处理方式不同可进一步细分为遍历和分治两种方法。

Python - Divide and Conquer

```python
"""
Definition of TreeNode:
class TreeNode:
    def __init__(self, val):
        this.val = val
        this.left, this.right = None, None
"""


class Solution:
    """
    @param root: The root of binary tree.
    @return: Preorder in ArrayList which contains node values.
    """
    def preorderTraversal(self, root):
```

```
        if root == None:
            return []
        return [root.val] + self.preorderTraversal(root.left) \
                          + self.preorderTraversal(root.right)
```

C++ - Divide and Conquer

```
/**
 * 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: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root != NULL) {
            // Divide (分)
            vector<int> left = preorderTraversal(root->left);
            vector<int> right = preorderTraversal(root->right);
            // Merge
            result.push_back(root->val);
            result.insert(result.end(), left.begin(), left.end());
            result.insert(result.end(), right.begin(), right.end());
        }

        return result;
    }
};
```

C++ - Traversal

```
/**
 * 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: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        traverse(root, result);

        return result;
    }

private:
    void traverse(TreeNode *root, vector<int> &ret) {
        if (root != NULL) {
            ret.push_back(root->val);
            traverse(root->left, ret);
            traverse(root->right, ret);
        }
    }
};
```

Java - Divide and Conquer

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root != null) {
            // Divide
            List<Integer> left = preorderTraversal(root.left);
            List<Integer> right = preorderTraversal(root.right);
            // Merge
            result.add(root.val);
            result.addAll(left);
            result.addAll(right);
        }

        return result;
    }
}
```

源码分析

使用遍历的方法保存递归返回结果需要使用辅助递归函数 `traverse` ，将结果作为参数传入递归函数中，传值时注意应使用 `vector` 的引用。 分治方法首先分开计算各结果，最后合并到最终结果中。 C++ 中由于是使用vector, 将新的vector插入另一vector不能再使用push_back, 而应该使用insert。 Java 中使用 `addAll` 方法.

复杂度分析

遍历树中节点，时间复杂度 $O(n)$, 未使用额外空间。

题解2 - 迭代

迭代时需要利用栈来保存遍历到的节点，纸上画图分析后发现应首先进行出栈抛出当前节点，保存当前节点的值，随后将右、左节点分别入栈(注意入栈顺序，先右后左)，迭代到其为叶子节点(NULL)为止。

Python

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def preorderTraversal(self, root):
        if root is None:
            return []

        result = []
        s = []
        s.append(root)
        while s:
            root = s.pop()
            result.append(root.val)
            if root.right is not None:
                s.append(root.right)
            if root.left is not None:
                s.append(root.left)

        return result
```

C++

```
/**
 * 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: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root == NULL) return result;

        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();
            result.push_back(node->val);
            if (node->right != NULL) {
                s.push(node->right);
            }
            if (node->left != NULL) {
                s.push(node->left);
            }
        }

        return result;
    }
};
```

Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root == null) return result;

        Stack<TreeNode> s = new Stack<TreeNode>();
        s.push(root);
        while (!s.empty()) {
            TreeNode node = s.pop();
            result.add(node.val);
            if (node.right != null) s.push(node.right);
```

```
            if (node.left != null) s.push(node.left);
        }

        return result;
    }
}
```

源码分析

1. 对root进行异常处理
2. 将root压入栈
3. 循环终止条件为栈s为空，所有元素均已处理完
4. 访问当前栈顶元素(首先取出栈顶元素，随后pop掉栈顶元素)并存入最终结果
5. 将右、左节点分别压入栈内，以便取元素时为先左后右。
6. 返回最终结果

其中步骤4,5,6为迭代的核心，对应前序遍历「根左右」。

所以说到底，**使用迭代，只不过是另外一种形式的递归。**使用递归的思想去理解遍历问题会容易理解许多。

复杂度分析

使用辅助栈，最坏情况下栈空间与节点数相等，空间复杂度近似为 $O(n)$, 对每个节点遍历一次，时间复杂度近似为 $O(n)$.

Binary Tree Inorder Traversal

Source

- leetcode: Binary Tree Inorder Traversal | LeetCode OJ
- lintcode: (67) Binary Tree Inorder Traversal

```
Given a binary tree, return the inorder traversal of its nodes' values.

Example
Given binary tree {1,#,2,3},

   1
    \
     2
    /
   3

return [1,3,2].

Challenge
Can you do it without recursion?
```

题解1 - 递归版

中序遍历的访问顺序为『先左再根后右』，递归版最好理解，递归调用时注意返回值和递归左右子树的顺序即可。

Python

```python
"""
Definition of TreeNode:
class TreeNode:
    def __init__(self, val):
        this.val = val
        this.left, this.right = None, None
"""


class Solution:
    """
    @param root: The root of binary tree.
    @return: Inorder in ArrayList which contains node values.
    """
    def inorderTraversal(self, root):
        if root is None:
            return []
        else:
            return [root.val] + self.inorderTraversal(root.left) \
                              + self.inorderTraversal(root.right)
```

Python - with helper

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def inorderTraversal(self, root):
        result = []
        self.helper(root, result)
        return result

    def helper(self, root, ret):
        if root is not None:
            self.helper(root.left, ret)
            ret.append(root.val)
            self.helper(root.right, ret)
```

C++

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        helper(root, result);
        return result;
    }

private:
    void helper(TreeNode *root, vector<int> &ret) {
        if (root != NULL) {
            helper(root->left, ret);
            ret.push_back(root->val);
            helper(root->right, ret);
        }
    }
};
```

Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        helper(root, result);
        return result;
    }

    private void helper(TreeNode root, List<Integer> ret) {
        if (root != null) {
            helper(root.left, ret);
            ret.add(root.val);
            helper(root.right, ret);
        }
    }
}
```

源码分析

Python 这种动态语言在写递归时返回结果好处理点，无需声明类型。通用的方法为在递归函数入口参数中传入返回结果， 也可使用分治的方法替代辅助函数。

复杂度分析

树中每个节点都需要被访问常数次，时间复杂度近似为 $O(n)$. 未使用额外辅助空间。

题解2 - 迭代版

使用辅助栈改写递归程序，中序遍历没有前序遍历好写，其中之一就在于入栈出栈的顺序和限制规则。我们采用「左根右」的访问顺序可知主要由如下四步构成。

1. 首先需要一直对左子树迭代并将非空节点入栈
2. 节点指针为空后不再入栈
3. 当前节点为空时进行出栈操作，并访问栈顶节点
4. 将当前指针p用其右子节点替代

步骤2,3,4对应「左根右」的遍历结构，只是此时的步骤2取的左值为空。

Python

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def inorderTraversal(self, root):
        result = []
        s = []
        while root is not None or s:
            if root is not None:
                s.append(root)
                root = root.left
            else:
                root = s.pop()
                result.append(root.val)
                root = root.right

        return result
```

C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Inorder in vector which contains node values.
     */
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        stack<TreeNode *> s;

        while (!s.empty() || NULL != root) {
            if (root != NULL) {
                s.push(root);
                root = root->left;
            } else {
                root = s.top();
                s.pop();
                result.push_back(root->val);
                root = root->right;
```

```
            }
        }

        return result;
    }
};
```

Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        Stack<TreeNode> s = new Stack<TreeNode>();
        while (root != null || !s.empty()) {
            if (root != null) {
                s.push(root);
                root = root.left;
            } else {
                root = s.pop();
                result.add(root.val);
                root = root.right;
            }
        }

        return result;
    }
}
```

源码分析

使用栈的思想模拟递归，注意迭代的演进和边界条件即可。

复杂度分析

最坏情况下栈保存所有节点，空间复杂度 $O(n)$, 时间复杂度 $O(n)$.

Reference

Binary Tree Postorder Traversal

Source

- leetcode: Binary Tree Postorder Traversal | LeetCode OJ
- lintcode: (68) Binary Tree Postorder Traversal

```
Given a binary tree, return the postorder traversal of its nodes' values.

Example
Given binary tree {1,#,2,3},

   1
    \
     2
    /
   3

return [3,2,1].

Challenge
Can you do it without recursion?
```

题解1 - 递归

首先使用递归便于理解。

Python - Divide and Conquer

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def postorderTraversal(self, root):
        if root is None:
            return []
        else:
            return self.postorderTraversal(root.left) +\
                   self.postorderTraversal(root.right) + [root.val]
```

C++ - Traversal

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in vector which contains node values.
     */
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;

        traverse(root, result);

        return result;
    }

private:
    void traverse(TreeNode *root, vector<int> &ret) {
        if (root == NULL) {
            return;
        }

        traverse(root->left, ret);
        traverse(root->right, ret);
        ret.push_back(root->val);
    }
};
```

Java - Divide and Conquer

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root != null) {
            List<Integer> left = postorderTraversal(root.left);
            result.addAll(left);
            List<Integer> right = postorderTraversal(root.right);
```

```
            result.addAll(right);
            result.add(root.val);
        }

        return result;
    }
}
```

源码分析

递归版的太简单了，没啥好说的，注意入栈顺序。

复杂度分析

时间复杂度近似为 $O(n)$.

题解2 - 迭代

使用递归写后序遍历那是相当的简单，我们来个不使用递归的迭代版。整体思路仍然为「左右根」，那么怎么才能知道什么时候该访问根节点呢？问题即转化为如何保证左右子节点一定先被访问到？由于入栈之后左右节点已无法区分，因此需要区分左右子节点是否被访问过(加入到最终返回结果中)。除了有左右节点的情况，根节点也可能没有任何子节点，此时也可直接将其值加入到最终返回结果中。

Python

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def postorderTraversal(self, root):
        result = []
        if root is None:
            return result
        s = []
        # previously traversed node
        prev = None
        s.append(root)
        while s:
            curr = s[-1]
            noChild = curr.left is None and curr.right is None
            childVisited = (prev is not None) and \
                           (curr.left == prev or curr.right == prev)
            if noChild or childVisited:
                result.append(curr.val)
                s.pop()
                prev = curr
```

```
            else:
                if curr.right is not None:
                    s.append(curr.right)
                if curr.left is not None:
                    s.append(curr.left)

        return result
```

C++

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> result;
        if (root == NULL) return result;

        TreeNode *prev = NULL;
        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *curr = s.top();
            bool noChild = false;
            if (curr->left == NULL && curr->right == NULL) {
                noChild = true;
            }
            bool childVisited = false;
            if (prev != NULL && (curr->left == prev || curr->right == prev)) {
                childVisited = true;
            }

            // traverse
            if (noChild || childVisited) {
                result.push_back(curr->val);
                s.pop();
                prev = curr;
            } else {
                if (curr->right != NULL) s.push(curr->right);
                if (curr->left != NULL) s.push(curr->left);
            }
        }

        return result;
    }
};
```

Java

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root == null) return result;

        Stack<TreeNode> s = new Stack<TreeNode>();
        s.push(root);
        TreeNode prev = null;
        while (!s.empty()) {
            TreeNode curr = s.peek();
            boolean noChild = false;
            if (curr.left == null && curr.right == null) {
                noChild = true;
            }
            boolean childVisited = false;
            if (prev != null && (curr.left == prev || curr.right == prev)) {
                childVisited = true;
            }

            // traverse
            if (noChild || childVisited) {
                result.add(curr.val);
                s.pop();
                prev = curr;
            } else {
                if (curr.right != null) s.push(curr.right);
                if (curr.left != null) s.push(curr.left);
            }

        }

        return result;
    }
}
```

源码分析

遍历顺序为『左右根』，判断根节点是否应该从栈中剔除有两种条件，一为无子节点，二为子节点已遍历过。判断子节点是否遍历过需要排除 `prev == null` 的情况，因为 prev 初始化为 null.

将递归写成迭代的难点在于如何在迭代中体现递归本质及边界条件的确立，可使用简单示例和纸上画出栈调用图辅助分析。

复杂度分析

最坏情况下栈内存储所有节点，空间复杂度近似为 $O(n)$, 每个节点遍历两次或以上，时间复杂度近似为 $O(n)$.

题解3 - Iterative

要想得到『左右根』的后序遍历结果，我们发现只需将『根右左』的结果转置即可，而先序遍历通常为『根左右』，故改变『左右』的顺序即可，所以如此一来后序遍历的非递归实现起来就非常简单了。

C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in vector which contains node values.
     */
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root == NULL) return result;

        stack<TreeNode*> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();
            result.push_back(node->val);
            // root, right, left => left, right, root
            if (node->left != NULL) s.push(node->left);
            if (node->right != NULL) s.push(node->right);
        }
        // reverse
        std::reverse(result.begin(), result.end());
        return result;
    }
};
```

源码分析

注意入栈的顺序和最后转置即可。

复杂度分析

同先序遍历。

Reference

- [leetcode]Binary Tree Postorder Traversal @ Python - 南郭子綦 - 解释清晰
- 更简单的非递归遍历二叉树的方法 - 比较新颖和简洁的实现

Binary Tree Level Order Traversal

Source

- lintcode: (69) Binary Tree Level Order Traversal

```
Given a binary tree, return the level order traversal of its nodes' values.
(ie, from left to right, level by level).

Example
Given binary tree {3,9,20,#,#,15,7},

    3
   / \
  9  20
    /  \
   15   7

return its level order traversal as:

[
  [3],
  [9,20],
  [15,7]
]
Challenge
Using only 1 queue to implement it.
```

题解 - 使用队列

此题为广搜的基础题，使用一个队列保存每层的节点即可。出队和将子节点入队的实现使用 for 循环，将每一轮的节点输出。

C++

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

class Solution {
    /**
```

```
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int> > result;

        if (NULL == root) {
            return result;
        }

        queue<TreeNode *> q;
        q.push(root);
        while (!q.empty()) {
            vector<int> list;
            int size = q.size(); // keep the queue size first
            for (int i = 0; i != size; ++i) {
                TreeNode * node = q.front();
                q.pop();
                list.push_back(node->val);
                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }
            result.push_back(list);
        }

        return result;
    }
};
```

Java

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


public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null) return result;
```

```
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            int qLen = q.size();
            ArrayList<Integer> aList = new ArrayList<Integer>();
            for (int i = 0; i < qLen; i++) {
                TreeNode node = q.poll();
                aList.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            result.add(aList);
        }

        return result;
    }
}
```

源码分析

1. 异常，还是异常
2. 使用STL的 `queue` 数据结构，将 `root` 添加进队列
3. **遍历当前层所有节点，注意需要先保存队列大小，因为在入队出队时队列大小会变化**
4. `list` 保存每层节点的值，每次使用均要初始化

复杂度分析

使用辅助队列，空间复杂度 $O(n)$, 时间复杂度 $O(n)$.

Binary Tree Level Order Traversal II

Source

- leetcode: Binary Tree Level Order Traversal II | LeetCode OJ
- lintcode: (70) Binary Tree Level Order Traversal II

```
Given a binary tree, return the bottom-up level order traversal of its nodes' values.
(ie, from left to right, level by level from leaf to root).

Example
Given binary tree {3,9,20,#,#,15,7},

    3
   / \
  9  20
    /  \
   15   7


return its bottom-up level order traversal as:
[
  [15,7],
  [9,20],
  [3]
]
```

题解

此题在普通的 BFS 基础上增加了逆序输出，简单的实现可以使用辅助栈或者最后对结果逆序。

Java - Stack

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


public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: buttom-up level order a list of lists of integer
```

```
     */
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null) return result;

        Stack<ArrayList<Integer>> s = new Stack<ArrayList<Integer>>();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            int qLen = q.size();
            ArrayList<Integer> aList = new ArrayList<Integer>();
            for (int i = 0; i < qLen; i++) {
                TreeNode node = q.poll();
                aList.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            s.push(aList);
        }

        while (!s.empty()) {
            result.add(s.pop());
        }
        return result;
    }
}
```

Java - Reverse

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


public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: buttom-up level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null) return result;

        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            int qLen = q.size();
            ArrayList<Integer> aList = new ArrayList<Integer>();
            for (int i = 0; i < qLen; i++) {
```

```
                TreeNode node = q.poll();
                aList.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            result.add(aList);
        }

        Collections.reverse(result);
        return result;
    }
}
```

源码分析

Java 中 Queue 是接口，通常可用 LinkedList 实例化。

复杂度分析

时间复杂度为 $O(n)$, 使用了队列或者辅助栈作为辅助空间，空间复杂度为 $O(n)$.

Maximum Depth of Binary Tree

Source

- lintcode: (97) Maximum Depth of Binary Tree

```
Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down t

Example
Given a binary tree as follow:

        1

     /     \

   2       3

          /    \

        4      5

The maximum depth is 3
```

题解 - 递归

树遍历的题最方便的写法自然是递归，不过递归调用的层数过多可能会导致栈空间溢出，因此需要适当考虑递归调用的层数。我们首先来看看使用递归如何解这道题，要求二叉树的最大深度，直观上来讲使用深度优先搜索判断左右子树的深度孰大孰小即可，从根节点往下一层树的深度即自增1，遇到 `NULL` 时即返回0。

由于对每个节点都会使用一次 `maxDepth` ，故时间复杂度为 $O(n)$, 树的深度最大为 n, 最小为 $\log_2 n$, 故空间复杂度介于 $O(\log n)$ 和 $O(n)$ 之间。

C++ Recursion

```
/**
 * 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: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);

        return max(left_depth, right_depth) + 1;
    }
};
```

Java Recursion

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        // write your code here
        if (root == null) {
            return 0;
        }
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}
```

题解 - 迭代(显式栈)

使用递归可能会导致栈空间溢出，这里使用显式栈空间(使用堆内存)来代替之前的隐式栈空间。从上节递归版的代码(先处理左子树，后处理右子树，最后返回其中的较大值)来看，是可以使用类似后序遍历的迭代思想去实现的。

首先使用后序遍历的模板，在每次迭代循环结束处比较栈当前的大小和当前最大值 `max_depth` 进行比较。

C++ Iterative with stack

```
/**
 * 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: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        TreeNode *curr = NULL, *prev = NULL;
        stack<TreeNode *> s;
        s.push(root);

        int max_depth = 0;

        while(!s.empty()) {
            curr = s.top();
            if (!prev || prev->left == curr || prev->right == curr) {
                if (curr->left) {
                    s.push(curr->left);
                } else if (curr->right){
                    s.push(curr->right);
                }
            } else if (curr->left == prev) {
                if (curr->right) {
                    s.push(curr->right);
                }
            } else {
                s.pop();
            }

            prev = curr;

            if (s.size() > max_depth) {
                max_depth = s.size();
            }
        }

        return max_depth;
```

```
    }
};
```

题解 - 迭代(队列)

在使用了递归/后序遍历求解树最大深度之后，我们还可以直接从问题出发进行分析，树的最大深度即为广度优先搜索中的层数，故可以直接使用广度优先搜索求出最大深度，在原结果返回处使用 `++max_depth` 替代即可。

C++ Iterative with queue

```
/**
 * 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: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        queue<TreeNode *> q;
        q.push(root);

        int max_depth = 0;
        while(!q.empty()) {
            int size = q.size();
            for (int i = 0; i != size; ++i) {
                TreeNode *node = q.front();
                q.pop();

                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }

            ++max_depth;
        }
```

```
        return max_depth;
    }
};
```

Java Iterative with queue

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int level = 0;
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        int curNum = 1; //num of nodes left in current level
        int nextNum = 0; //num of nodes in next level

        while(!queue.isEmpty()) {
            TreeNode n = queue.poll();
            curNum--;
            if (n.left != null) {
                queue.add(n.left);
                nextNum++;
            }
            if (n.right != null) {
                queue.add(n.right);
                nextNum++;
            }
            if (curNum == 0) {
                curNum = nextNum;
                nextNum = 0;
                level++;
            }
        }
        return level;
    }
}
```

Balanced Binary Tree

Source

- lintcode: (93) Balanced Binary Tree

```
Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the

Example

Given binary tree A={3,9,20,#,#,15,7}, B={3,#,20,15,7}

A)  3            B)    3
   / \                  \
  9  20                 20
    /  \               / \
   15   7             15  7

The binary tree A is a height-balanced binary tree, but B is not.
```

题解 - 递归

根据题意，平衡树的定义是两子树的深度差最大不超过1，显然使用递归进行分析较为方便。既然使用递归，那么接下来就需要分析递归调用的终止条件。和之前的 Maximum Depth of Binary Tree | Algorithm 类似， `NULL == root` 必然是其中一个终止条件，返回 `0` ；根据题意还需的另一终止条件应为「左右子树高度差大于1」，但对应此终止条件的返回值是多少？—— `INT_MAX` or `INT_MIN` ？想想都不合适，为何不在传入参数中传入 `bool` 指针或者 `bool` 引用咧？并以此变量作为最终返回值，此法看似可行，先来看看鄙人最开始想到的这种方法。

C++ Recursion with extra bool variable

```
/**
 * 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: The root of binary tree.
     * @return: True if this Binary tree is Balanced, or false.
     */
    bool isBalanced(TreeNode *root) {
        if (NULL == root) {
            return true;
        }

        bool result = true;
        maxDepth(root, result);

        return result;
    }

private:
    int maxDepth(TreeNode *root, bool &isBalanced) {
        if (NULL == root) {
            return 0;
        }

        int leftDepth = maxDepth(root->left, isBalanced);
        int rightDepth = maxDepth(root->right, isBalanced);
        if (abs(leftDepth - rightDepth) > 1) {
            isBalanced = false;
            // speed up the recursion process
            return INT_MAX;
        }

        return max(leftDepth, rightDepth) + 1;
    }
};
```

源码解析

如果在某一次子树高度差大于1时，返回 `INT_MAX` 以减少不必要的计算过程，加速整个递归调用的过程。

初看起来上述代码好像还不错的样子，但是在看了九章的实现后，瞬间觉得自己弱爆了... 首先可以确定 `abs(leftDepth - rightDepth) > 1` 肯定是需要特殊处理的，如果返回 `-1` 呢？咋一看似乎在下一步返回 `max(leftDepth, rightDepth) + 1` 时会出错，再进一步想想，我们能否不让 `max...` 这一句执行呢？如果返回了 `-1` ，其接盘侠必然是 `leftDepth` 或者 `rightDepth` 中的一个，因此我们只需要在判断子树高度差大于1的同时也判断下左右子树深度是否为 `-1` 即可都返回 `-1` ，不得不说这种处理方法要精妙的多，赞！

C++ Recursion without extra bool variable

```
/**
 * forked from http://www.jiuzhang.com/solutions/balanced-binary-tree/
 * 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: The root of binary tree.
     * @return: True if this Binary tree is Balanced, or false.
     */
    bool isBalanced(TreeNode *root) {
        return (-1 != maxDepth(root));
    }

private:
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        if (leftDepth == -1 || rightDepth == -1 || \
            abs(leftDepth - rightDepth) > 1) {
            return -1;
        }

        return max(leftDepth, rightDepth) + 1;
    }
};
```

Binary Tree Maximum Path Sum

Source

- lintcode: (94) Binary Tree Maximum Path Sum

```
Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.
Example
Given the below binary tree,

       1
      / \
     2   3
Return 6.
```

题解1 - 递归中仅返回子树路径长度

如题目右侧的四颗半五角星所示，本题属于中等偏难的那一类题。题目很短，要求返回最大路径和。咋看一下感觉使用递归应该很快就能搞定，实则不然，**因为从题目来看路径和中不一定包含根节点！也就是说可以起止于树中任一连通节点。**

弄懂题意后我们就来剖析剖析，本着由简入难的原则，我们先来分析若路径和包含根节点，如何才能使其路径和达到最大呢？选定根节点后，路径和中必然包含有根节点的值，剩下的部分则为左子树和右子树，要使路径和最大，则必然要使左子树和右子树中的路径长度都取最大。

> 注意区分包含根节点的路径和(题目要的答案)和左子树/右子树部分的路径长度(答案的一个组成部分)。路径和=根节点+左子树路径长度+右子树路径长度

```
       -10
       /  \
      2    -3
     / \   / \
    3   4 5   7
```

如上所示，包含根节点 `-10` 的路径和组成的节点应为 `4 -> 2 -> -10 <- -3 <- 7`，对于左子树而言，其可能的路径组成节点为 `3 -> 2` 或 `4 -> 2`，而不是像根节点的路径和那样为 `3 -> 2 <- 4`. 这种差异也就造成了我们不能很愉快地使用递归来求得最大路径和。

我们使用分治的思想来分析路径和/左子树/右子树，设 $f(root)$ 为 `root` 的子树到 `root` 节点(含)路径长度的最大值，那么我们有 $f(root) = root -> val + \max(f(root -> left),\ f(root -> right))$

递归模型已初步建立起来，接下来就是分析如何将左右子树的路径长度和最终题目要求的「路径和」挂钩。设 $g(root)$ 为当「路径和」中根节点为 `root` 时的值，那么我们有
$g(root) = root -> val + f(root -> left) + f(root -> right)$

顺着这个思路，我们可以遍历树中的每一个节点求得 $g(node)$ 的值，输出最大值即可。如果不采取任何记忆化存储的措施，其时间复杂度必然是指数级别的。嗯，先来看看这个思路的具体实现，后续再对其进行优化。遍历节点我们使用递归版的前序遍历，求单边路径长度采用递归。

C++ Recursion + Iteration(Not Recommended)

Time Limit Exceeded

```
/**
 * 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: The root of binary tree.
     * @return: An integer
     */
    int maxPathSum(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int result = INT_MIN;
        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();

            int temp_path_sum = node->val + singlePathSum(node->left) \
                                          + singlePathSum(node->right);

            if (temp_path_sum > result) {
                result = temp_path_sum;
            }

            if (NULL != node->right) s.push(node->right);
            if (NULL != node->left) s.push(node->left);
        }

        return result;
    }

private:
    int singlePathSum(TreeNode *root) {
        if (NULL == root) {
            return 0;
```

```
        }

        int path_sum = max(singlePathSum(root->left), singlePathSum(root->right));
        return max(0, (root->val + path_sum));
    }
};
```

源码分析

注意 `singlePathSum` 中最后的返回值，如果其路径长度 `path_sum` 比0还小，那么取这一段路径反而会减少最终的路径和，故不应取这一段，我们使用0表示这一隐含意义。

题解2 - 递归中同时返回子树路径长度和路径和

C++ using std::pair

上题求路径和和左右子树路径长度是分开求得的，因此导致了时间复杂度剧增的恶劣情况，从题解的递推关系我们可以看出其实是可以在一次递归调用过程中同时求得路径和和左右子树的路径长度的，只不过此时递归程序需要返回的不再是一个值，而是路径长度和路径和这一组值！C++中我们可以使用 `pair` 或者自定义新的数据类型来相对优雅的解决。

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
private:
    pair<int, int> helper(TreeNode *root) {
        if (NULL == root) {
            return make_pair(0, INT_MIN);
        }

        pair<int, int> leftTree = helper(root->left);
        pair<int, int> rightTree = helper(root->right);

        int single_path_sum = max(leftTree.first, rightTree.first) + root->val;
        single_path_sum = max(0, single_path_sum);

        int max_sub_sum = max(leftTree.second, rightTree.second);
        int max_path_sum = root->val + leftTree.first + rightTree.first;
        max_path_sum = max(max_sub_sum, max_path_sum);

        return make_pair(single_path_sum, max_path_sum);
    }
```

```
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxPathSum(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        return helper(root).second;
    }
};
```

源码分析

除了使用 `pair` 对其进行封装，也可使用嵌套类新建一包含单路径长度和全局路径和两个变量的类，不过我用 C++写的没编译过... 老是提示 `...private` ，遂用 `pair` 改写之。建议使用 `class` 而不是 `pair` 封装 `single_path_sum` 和 `max_path_sum` [pair_is_harmful].

这种方法难以理解的地方在于这种实现方式的正确性，较为关键的语句为 `max_path_sum = max(max_sub_sum, max_path_sum)` , 这行代码是如何体现题目中以下的这句话的呢?

> The path may start and end at any node in the tree.

简单来讲，题解2从两个方面予以保证：

1. 采用「冒泡」法返回不经过根节点的路径和的较大值。
2. 递推子树路径长度(不变值)而不是到该节点的路径和(有可能变化)，从而保证了这种解法的正确性。

如果还不理解的建议就以上面那个根节点为-10的例子画一画。

C++ using self-defined class

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

```
private:
    ResultType helper(TreeNode *root) {
        if (root == NULL) {
            ResultType *nullResult = new ResultType(0, INT_MIN);
            return *nullResult;
        }
        // Divide
        ResultType left = helper(root->left);
        ResultType right = helper(root->right);

        // Conquer
        int singlePath = max(left.singlePath, right.singlePath) + root->val;
        singlePath = max(singlePath, 0);

        int maxPath = max(left.maxPath, right.maxPath);
        maxPath = max(maxPath, left.singlePath + right.singlePath + root->val);

        ResultType *result = new ResultType(singlePath, maxPath);
        return *result;
    }

public:
    int maxPathSum(TreeNode *root) {
        ResultType result = helper(root);
        return result.maxPath;
    }
};
```

源码分析

1. 如果不用 `ResultType *XXX = new ResultType ...` 再 `return *XXX` 的方式，则需要在自定义 class 中重载 `new` operator。
2. 如果遇到 `...private` 的编译错误，则是因为自定义 class 中需要把成员声明为 public，否则需要把访问这个成员的函数也做 class 内部处理。

Reference

- > pair_is_harmful. std::pair considered harmful! « Modern Maintainable Code - 作者指出了 `pair` 不能滥用的原因，如不可维护，信息量小。 ↩

- Binary Tree Maximum Path Sum | 九章算法

Lowest Common Ancestor

Source

- lintcode: (88) Lowest Common Ancestor

```
Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of th

The lowest common ancestor is the node with largest depth which is the ancestor of both n
Example
        4

    /     \

  3         7

          /     \

        5         6
For 3 and 5, the LCA is 4.

For 5 and 6, the LCA is 7.
For 6 and 7, the LCA is 7.
```

题解1 - 自底向上

初次接触这种题可能会没有什么思路，在没有思路的情况下我们就从简单例子开始分析！首先看看 `3` 和 `5` ，这两个节点分居根节点 `4` 的两侧，如果可以从子节点往父节点递推，那么他们将在根节点 `4` 处第一次重合；再来看看 `5` 和 `6` ，这两个都在根节点 `4` 的右侧，沿着父节点往上递推，他们将在节点 `7` 处第一次重合；最后来看看 `6` 和 `7` ，此时由于 `7` 是 `6` 的父节点，故 `7` 即为所求。从这三个基本例子我们可以总结出两种思路——自顶向下(从前往后递推)和自底向上(从后往前递推)。

顺着上述实例的分析，我们首先看看自底向上的思路，自底向上的实现用一句话来总结就是——如果遍历到的当前节点是 A/B 中的任意一个，那么我们就向父节点汇报此节点，否则递归到节点为空时返回空值。具体来说会有如下几种情况：

1. 当前节点不是两个节点中的任意一个，此时应判断左右子树的返回结果。
    - 若左右子树均返回非空节点，那么当前节点一定是所求的根节点，将当前节点逐层向前汇报。// 两个节点分居树的两侧
    - 若左右子树仅有一个子树返回非空节点，则将此非空节点向父节点汇报。// 节点仅存在于树的一侧
    - 若左右子树均返回 `NULL` , 则向父节点返回 `NULL` . // 节点不在这棵树中
2. 当前节点即为两个节点中的一个，此时向父节点返回当前节点。

根据此递归模型容易看出应该使用中序遍历来实现。

C++ Recursion From Bottom to Top

```
/**
 * 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: The root of the binary search tree.
     * @param A and B: two nodes in a Binary.
     * @return: Return the least common ancestor(LCA) of the two nodes.
     */
    TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *A, TreeNode *B) {
        // return either A or B or NULL
        if (NULL == root || root == A || root == B) return root;

        TreeNode *left = lowestCommonAncestor(root->left, A, B);
        TreeNode *right = lowestCommonAncestor(root->right, A, B);

        // A and B are on both sides
        if ((NULL != left) && (NULL != right)) return root;

        // either left or right or NULL
        return (NULL != left) ? left : right;
    }
};
```

源码分析

结合例子和递归的整体思想去理解代码，在 `root == A || root == B` 后即层层上浮(自底向上)，直至找到最终的最小公共祖先节点。

最后一行 `return (NULL != left) ? left : right;` 将非空的左右子树节点和空值都包含在内了，十分精炼！ leetcode

> 细心的你也许会发现，其实题解的分析漏掉了一种情况，即树中可能只含有 A/B 中的一个节点！这种情况应该返回空值，但上述实现均返回非空节点。重复节点就不考虑了，太复杂了...

不会漏掉 A/B 中只有一个节点的情况的方法

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode A, TreeNode B) {
        if (root == null || root == A || root == B) {
            return root;
        }
```

```
        // Divide
        TreeNode left = lowestCommonAncestor(root.left, A, B);
        TreeNode right = lowestCommonAncestor(root.right, A, B);

        // Conquer
        if (left != null && right != null) {
            return root;
        }
        if (left != null) {
            return left;
        }
        if (right != null) {
            return right;
        }
        return null;
    }
}
```

其实这个代码只是把上一个版本的代码最后简洁的判断语句改成复杂的多层判断就可以了。同样是分治法实现。

题解 - 自底向上(计数器)

为了解决上述方法可能导致误判的情况，我们可以对返回结果添加计数器来解决。**由于此计数器的值只能由子树向上递推，故不能再使用中序遍历，而应该改用后序遍历。**

定义 `pair<TreeNode *, int> result(node, counter)` 表示遍历到某节点时的返回结果，返回的 `node` 表示LCA 路径中的可能的最小节点，相应的计数器 `counter` 则表示目前和 `A` 或者 `B` 匹配的节点数，若计数器为2，则表示已匹配过两次，该节点即为所求，若只匹配过一次，还需进一步向上递推。表述地可能比较模糊，还是看看代码吧。

C++ Post-order(counter)

```
/**
 * 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: The root of the binary search tree.
     * @param A and B: two nodes in a Binary.
     * @return: Return the least common ancestor(LCA) of the two nodes.
     */
    TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *A, TreeNode *B) {
```

```
        if ((NULL == A) || (NULL == B)) return NULL;

        pair<TreeNode *, int> result = helper(root, A, B);

        if (A != B) {
            return (2 == result.second) ? result.first : NULL;
        } else {
            return (1 == result.second) ? result.first : NULL;
        }
    }

private:
    pair<TreeNode *, int> helper(TreeNode *root, TreeNode *A, TreeNode *B) {
        TreeNode * node = NULL;
        if (NULL == root) return make_pair(node, 0);

        pair<TreeNode *, int> left = helper(root->left, A, B);
        pair<TreeNode *, int> right = helper(root->right, A, B);

        // return either A or B
        int count = max(left.second, right.second);
        if (A == root || B == root)  return make_pair(root, ++count);

        // A and B are on both sides
        if (NULL != left.first && NULL != right.first) return make_pair(root, 2);

        // return either left or right or NULL
        return (NULL != left.first) ? left : right;
    }
};
```

源码分析

在 `A == B` 时，计数器返回1的节点即为我们需要的节点，否则只取返回2的节点，如此便保证了该方法的正确性。对这种实现还有问题的在下面评论吧。

Reference

- > [leetcode]. Lowest Common Ancestor of a Binary Tree Part I | LeetCode - 清晰易懂的题解和实现。↩

- Lowest Common Ancestor of a Binary Tree Part II | LeetCode - 如果存在指向父节点的指针，我们能否有更好的解决方案?
- Lowest Common Ancestor of a Binary Search Tree (BST) | LeetCode - 二叉搜索树中求最小公共祖先。
- Lowest Common Ancestor | 九章算法 - 第一种和第二种方法可以在知道父节点时使用，但第二种 Divide and Conquer 才是本题需要的思想（第二种解法可以轻易改成不需要 parent 的指针的）。

Invert Binary Tree

Source

- leetcode: Invert Binary Tree | LeetCode OJ
- lintcode: (175) Invert Binary Tree

```
Invert a binary tree.

Example
  1         1
 / \       / \
2   3  => 3   2
   /       \
  4         4
Challenge
Do it in recursion is acceptable, can you do it without recursion?
```

题解1 - Recursive

二叉树的题用递归的思想求解自然是最容易的，此题要求为交换左右子节点，故递归交换之即可。具体实现可分返回值为空或者二叉树节点两种情况，返回值为节点的情况理解起来相对不那么直观一些。

C++ - return void

```cpp
/**
 * 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 TreeNode, the root of the binary tree
     * @return: nothing
     */
    void invertBinaryTree(TreeNode *root) {
        if (root == NULL) return;

        TreeNode *temp = root->left;
        root->left = root->right;
        root->right = temp;
```

```
        invertBinaryTree(root->left);
        invertBinaryTree(root->right);
    }
};
```

C++ - return TreeNode *

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return NULL;

        TreeNode *temp = root->left;
        root->left = invertTree(root->right);
        root->right = invertTree(temp);

        return root;
    }
};
```

源码分析

分三块实现，首先是节点为空的情况，然后使用临时变量交换左右节点，最后递归调用，递归调用的正确性可通过画图理解。

复杂度分析

每个节点遍历一次，时间复杂度为 $O(n)$, 使用了临时变量，空间复杂度为 $O(1)$.

题解2 - Iterative

递归的实现非常简单，那么非递归的如何实现呢？如果将递归改写成栈的实现，那么简单来讲就需要两个栈了，稍显复杂。其实仔细观察此题可发现使用 level-order 的遍历次序也可实现。即从根节点开始入队，交换左右节点，并将非空的左右子节点入队，从队列中取出节点，交换之，直至队列为空。

C++

```
/**
 * 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 TreeNode, the root of the binary tree
     * @return: nothing
     */
    void invertBinaryTree(TreeNode *root) {
        if (root == NULL) return;

        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            // pop out the front node
            TreeNode *node = q.front();
            q.pop();
            // swap between left and right pointer
            swap(node->left, node->right);
            // push non-NULL node
            if (node->left != NULL) q.push(node->left);
            if (node->right != NULL) q.push(node->right);
        }
    }
};
```

源码分析

交换左右指针后需要判断子节点是否非空，仅入队非空子节点。

复杂度分析

遍历每一个节点，时间复杂度为 $O(n)$, 使用了队列，最多存储最下一层子节点数目，最多只有总节点数的一半，故最坏情况下 $O(n)$.

Reference

- 0ms C++ Recursive/Iterative Solutions with Explanations - Leetcode Discuss

Diameter of a Binary Tree

Source

- Diameter of a Binary Tree - GeeksforGeeks

```
The diameter of a tree (sometimes called the width) is the number of nodes
on the longest path between two leaves in the tree.
The diagram below shows two trees each with diameter nine,
the leaves that form the ends of a longest path are shaded
(note that there is more than one path in each tree of length nine,
but no path longer than nine nodes).
```

diameter, 9 nodes, through root

diameter, 9 nodes, NOT through root

题解

和题 Lowest Common Ancestor 分析思路特别接近。

Java

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}

public class Solution {
    public int diameter(TreeNode root) {
        if (root == null) return 0;

        // left, right height
        int leftHight = getHeight(root.left);
```

```
        int rightHight = getHeight(root.right);

        // left, right subtree diameter
        int leftDia = diameter(root.left);
        int rightDia = diameter(root.right);

        int maxSubDia = Math.max(leftDia, rightDia);
        return Math.max(maxSubDia, leftHight + 1 + rightHight);
    }

    private int getHeight(TreeNode root) {
        if (root == null) return 0;

        return 1 + Math.max(getHeight(root.left), getHeight(root.right));
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);
        root.left.right.left = new TreeNode(6);
        root.left.right.left.right = new TreeNode(7);
        root.left.left.left = new TreeNode(8);

        Solution sol = new Solution();
        int maxDistance = sol.diameter(root);
        System.out.println("Max Distance: " + maxDistance);
    }
}
```

Reference

- Diameter of a Binary Tree - GeeksforGeeks
- Diameter of a Binary Tree | Algorithms

Construct Binary Tree from Preorder and Inorder Traversal

Source

- leetcode: Construct Binary Tree from Preorder and Inorder Traversal | LeetCode OJ
- lintcode: (73) Construct Binary Tree from Preorder and Inorder Traversal

```
Given preorder and inorder traversal of a tree, construct the binary tree.

Example
Given in-order [1,2,3] and pre-order [2,1,3], return a tree:
  2
 / \
1   3
Note
You may assume that duplicates do not exist in the tree.
```

题解

二叉树的重建，典型题。核心有两点：

1. preorder 先序遍历的第一个节点即为根节点。
2. 确定 inorder 数组中的根节点后其左子树和右子树也是 preorder 的左子树和右子树。

其中第二点是隐含条件，数组中没有重复元素，故可以根据先序遍历中第一个元素（根节点）得到根节点的值，然后在 inorder 中序遍历的数组中搜索得到根节点的索引值，即为左子树，右边为右子树。根据中序遍历中左子树的索引确定先序遍历数组中左子树的起止索引。递归直至处理完所有数组元素。

Java

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
     *@param preorder : A list of integers that preorder traversal of a tree
     *@param inorder : A list of integers that inorder traversal of a tree
     *@return : Root of a tree
```

```
     */
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || inorder == null) return null;
        if (preorder.length == 0 || inorder.length == 0) return null;
        if (preorder.length != inorder.length) return null;

        TreeNode root = helper(preorder, 0, preorder.length - 1,
                               inorder, 0, inorder.length - 1);
        return root;
    }

    private TreeNode helper(int[] preorder, int prestart, int preend,
                            int[] inorder, int instart, int inend) {
        // corner cases
        if (prestart > preend || instart > inend) return null;
        // build root TreeNode
        int root_val = preorder[prestart];
        TreeNode root = new TreeNode(root_val);
        // find index of root_val in inorder[]
        int index = findIndex(inorder, instart, inend, root_val);
        // build left subtree
        root.left = helper(preorder, prestart + 1, prestart + index - instart,
               inorder, instart, index - 1);
        // build right subtree
        root.right = helper(preorder, prestart + index - instart + 1, preend,
               inorder, index + 1, inend);
        return root;
    }

    private int findIndex(int[] nums, int start, int end, int target) {
        for (int i = start; i <= end; i++) {
            if (nums[i] == target) return i;
        }
        return -1;
    }
}
```

源码分析

由于需要知道左右子树在数组中的索引，故需要引入辅助方法。找根节点这个大家都能很容易地想到，但是最关键的一步——找出左右子树的起止索引，这一点就不那么直接了，老实说想了很久忽略了这个突破点。

复杂度分析

`findIndex` 时间复杂度近似 $O(n)$, `helper` 递归调用，每次调用都需要找中序遍历数组中的根节点，故总的时间复杂度为 $O(n^2)$. 原地生成最终二叉树，空间复杂度为 $O(1)$.

Reference

- Construct Binary Tree from Preorder and Inorder Traversal 参考程序 Java/C++/Python

Construct Binary Tree from Inorder and Postorder Traversal

Source

- lintcode: (72) Construct Binary Tree from Inorder and Postorder Traversal

```
Given inorder and postorder traversal of a tree, construct the binary tree.

Example
Given inorder [1,2,3] and postorder [1,3,2], return a tree:
  2
   / \
   1   3
   Note
   You may assume that duplicates do not exist in the tree.
```

题解

和题 Construct Binary Tree from Preorder and Inorder Traversal 几乎一致，关键在于找到中序遍历中的根节点和左右子树，递归解决。

Java

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
     *@param inorder : A list of integers that inorder traversal of a tree
     *@param postorder : A list of integers that postorder traversal of a tree
     *@return : Root of a tree
     */
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        if (inorder == null || postorder == null) return null;
        if (inorder.length == 0 || postorder.length == 0) return null;
        if (inorder.length != postorder.length) return null;

        TreeNode root = helper(inorder, 0, inorder.length - 1,
               postorder, 0, postorder.length - 1);
```

```
        return root;
    }

    private TreeNode helper(int[] inorder, int instart, int inend,
                            int[] postorder, int poststart, int postend) {
        // corner cases
        if (instart > inend || poststart > postend) return null;

        // build root TreeNode
        int root_val = postorder[postend];
        TreeNode root = new TreeNode(root_val);
        // find index of root_val in inorder[]
        int index = findIndex(inorder, instart, inend, root_val);
        // build left subtree
        root.left = helper(inorder, instart, index - 1,
                           postorder, poststart, poststart + index - instart - 1);
        // build right subtree
        root.right = helper(inorder, index + 1, inend,
                           postorder, poststart + index - instart, postend - 1);
        return root;
    }

    private int findIndex(int[] nums, int start, int end, int target) {
        for (int i = start; i <= end; i++) {
            if (nums[i] == target) return i;
        }
        return -1;
    }
}
```

源码分析

找根节点的方法作为私有方法，辅助函数需要注意索引范围。

复杂度分析

找根节点近似 $O(n)$, 递归遍历整个数组，嵌套找根节点的方法，故总的时间复杂度为 $O(n^2)$.

Subtree

Source

- lintcode: (245) Subtree

```
You have two every large binary trees: T1,
with millions of nodes, and T2, with hundreds of nodes.
Create an algorithm to decide if T2 is a subtree of T1.

Example
T2 is a subtree of T1 in the following case:
       1                3
      / \              /
T1 = 2   3      T2 =  4
        /
       4
T2 isn't a subtree of T1 in the following case:
       1               3
      / \               \
T1 = 2   3       T2 =    4
        /
       4
Note
A tree T2 is a subtree of T1 if there exists a node n in T1 such that
the subtree of n is identical to T2.
That is, if you cut off the tree at node n,
the two trees would be identical.
```

题解

判断 T2是否是 T1的子树，首先应该在 T1中找到 T2的根节点，找到根节点后两棵子树必须完全相同。所以整个思路分为两步走：找根节点，判断两棵树是否全等。咋看起来极其简单，但实际实现中还是比较精妙的，尤其是递归的先后顺序及条件与条件或的处理。

Java

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
```

```
     * @param T1, T2: The roots of binary tree.
     * @return: True if T2 is a subtree of T1, or false.
     */
    public boolean isSubtree(TreeNode T1, TreeNode T2) {
        if (T2 == null) return true;
        if (T1 == null) return false;
        return identical(T1, T2) || isSubtree(T1.left, T2) || isSubtree(T1.right, T2);
    }

    private boolean identical(TreeNode T1, TreeNode T2) {
        if (T1 == null && T2 == null) return true;
        if (T1 == null || T2 == null) return false;
        if (T1.val != T2.val) return false;
        return identical(T1.left, T2.left) && identical(T1.right, T2.right);
    }
}
```

源码分析

这道题的异常处理相对 trick 一点，需要理解 null 对子树的含义。另外需要先调用 `identical` 再递归调用 `isSubtree` 判断左右子树的情况。方法 `identical` 中调用 `.val` 前需要判断是否为 null, 而后递归调用判断左右子树是否 identical。

复杂度分析

identical 的调用，时间复杂度近似 $O(n)$, 查根节点的时间复杂度随机，平均为 $O(m)$, 故总的时间复杂度可近似为 $O(mn)$.

Reference

- LintCode: Subtree

Binary Tree Zigzag Level Order Traversal

Source

- leetcode: Binary Tree Zigzag Level Order Traversal 参考程序 Java/C++/Python
- lintcode: (71) Binary Tree Zigzag Level Order Traversal

```
Given a binary tree, return the zigzag level order traversal of its nodes' values.
(ie, from left to right, then right to left for the next level and alternate between).

Example
Given binary tree {3,9,20,#,#,15,7},

    3
   / \
  9  20
    /  \
   15   7


return its zigzag level order traversal as:

[
  [3],
  [20,9],
  [15,7]
]
```

题解1 - 队列

二叉树的广度优先遍历使用队列非常容易实现，这道题要求的是蛇形遍历，我们可以发现奇数行的遍历仍然可以按照广度优先遍历的方式实现，而对于偶数行，只要翻转一下就好了。

Java

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

public class Solution {
    /**
     * @param root: The root of binary tree.
```

```
     * @return: A list of lists of integer include
     *          the zigzag level order traversal of its nodes' values
     */
    public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null) return result;

        boolean odd = true;
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            // level traversal
            int qLen = q.size();
            ArrayList<Integer> level = new ArrayList<Integer>();
            for (int i = 0; i < qLen; i++) {
                TreeNode node = q.poll();
                level.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            // add level order reverse for even
            if (odd) {
                result.add(level);
            } else {
                Collections.reverse(level);
                result.add(level);
            }
            // flip odd and even
            odd = !odd;
        }

        return result;
    }
}
```

源码分析

区分奇数偶数行使用额外变量。

复杂度分析

需要 reverse 的节点数目近似为 n/2, 故时间复杂度 $O(n)$. 最下层节点数目最多 n/2, 故reverse 操作的空间复杂度可近似为 $O(n/2)$.

总的时间复杂度为 $O(n)$, 空间复杂度也为 $O(n)$.

Reference

- Binary Tree Zigzag Level Order Traversal 参考程序 Java/C++/Python
- Printing a Binary Tree in Zig Zag Level-Order | LeetCode

Binary Tree Serialization

Source

- lintcode: (7) Binary Tree Serialization

```
Design an algorithm and write code to serialize and deserialize a binary tree.
Writing the tree to a file is called 'serialization'
and reading back from the file to reconstruct
the exact same binary tree is 'deserialization'.
There is no limit of how you deserialize or serialize a binary tree,
you only need to make sure you can serialize a binary tree to a string
and deserialize this string to the original structure.
Have you met this question in a real interview? Yes
Example
An example of testdata: Binary tree {3,9,20,#,#,15,7},
denote the following structure:
  3
 / \
9  20
  /  \
 15   7
Our data serialization use bfs traversal.
This is just for when you got wrong answer and want to debug the input.

You can use other method to do serializaiton and deserialization.
```

题解

根据之前由前序，中序，后序遍历恢复二叉树的经验，确定根节点的位置十分重要（但是这里可能有重复元素，故和之前的题目不太一样）。能直接确定根节点的有前序遍历和广度优先搜索，其中较为简洁的为前序遍历。序列化较为简单，但是反序列化的实现不太容易。需要借助字符串解析工具。

Java

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
class Solution {
    /**
     * This method will be invoked first, you should design your own algorithm
     * to serialize a binary tree which denote by a root node to a string which
```

```
     * can be easily deserialized by your own "deserialize" method later.
     */
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        if (root == null) return sb.toString();

        seriaHelper(root, sb);

        return sb.substring(0, sb.length() - 1);
    }

    private void seriaHelper(TreeNode root, StringBuilder sb) {
        if (root == null) {
            sb.append("#,");
        } else {
            sb.append(root.val).append(",");
            seriaHelper(root.left, sb);
            seriaHelper(root.right, sb);
        }
    }

    /**
     * This method will be invoked second, the argument data is what exactly
     * you serialized at method "serialize", that means the data is not given by
     * system, it's given by your own serialize method. So the format of data is
     * designed by yourself, and deserialize it here as you serialize it in
     * "serialize" method.
     */
    public TreeNode deserialize(String data) {
        if (data == null || data.length() == 0) return null;

        StringTokenizer st = new StringTokenizer(data, ",");
        return deseriaHelper(st);
    }

    private TreeNode deseriaHelper(StringTokenizer st) {
        if (!st.hasMoreTokens()) return null;

        String val = st.nextToken();
        if (val.equals("#")) {
            return null;
        }

        TreeNode root = new TreeNode(Integer.parseInt(val));
        root.left = deseriaHelper(st);
        root.right = deseriaHelper(st);

        return root;
    }
}
```

源码分析

由二叉树序列化的过程不难，难就难在根据字符串进行反序列化，这里引入了 Java 中的 StringTokenizer 字符串分割工具，非常方便，使得递归得以顺利实现。其中 `deseriaHelper` 的实现较为巧妙。

复杂度分析

略

Reference

- Serialize and Deserialize a Binary Tree (pre order).
- Serialization/Deserialization of a Binary Tree | LeetCode

Binary Search Tree - 二叉搜索树

二叉搜索树的定义及简介在 Binary Search Trees 中已经有所介绍。简单来说就是当前节点的值大于等于左子结点的值，而小于右子节点的值。

Insert Node in a Binary Search Tree

Source

- lintcode: (85) Insert Node in a Binary Search Tree

```
Given a binary search tree  and a new tree node, insert the node into the tree. You shoul

Example
Given binary search tree as follow:

     2

  /    \

1        4

         /

       3

after Insert node 6, the tree should be:

     2

  /    \

1        4

         /   \

       3        6

Challenge
Do it without recursion
```

题解 - 递归

二叉树的题使用递归自然是最好理解的，代码也简洁易懂，缺点就是递归调用时栈空间容易溢出，故实际实现中一般使用迭代替代递归，性能更佳嘛。不过迭代的缺点就是代码量稍(很)大，逻辑也可能不是那么好懂。

既然确定使用递归，那么接下来就应该考虑具体的实现问题了。在递归的具体实现中，主要考虑如下两点：

1. 基本条件/终止条件 - 返回值需斟酌。
2. 递归步/条件递归 - 能使原始问题收敛。

首先来找找递归步，根据二叉查找树的定义，若插入节点的值若大于当前节点的值，则继续与当前节点的

右子树的值进行比较；反之则继续与当前节点的左子树的值进行比较。题目的要求是返回最终二叉搜索树的根节点，从以上递归步的描述中似乎还难以对应到实际代码，这时不妨分析下终止条件。

有了递归步，终止条件也就水到渠成了，若当前节点为空时，即返回结果。问题是——返回什么结果？当前节点为空时，说明应该将「插入节点」插入到上一个遍历节点的左子节点或右子节点。对应到程序代码中即为 `root->right = node` 或者 `root->left = node` . 也就是说递归步使用 `root->right/left = func(...)` 即可。

C++ Recursion

```
/**
 * forked from http://www.jiuzhang.com/solutions/insert-node-in-binary-search-tree/
 * 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: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    TreeNode* insertNode(TreeNode* root, TreeNode* node) {
        if (NULL == root) {
            return node;
        }

        if (node->val <= root->val) {
            root->left = insertNode(root->left, node);
        } else {
            root->right = insertNode(root->right, node);
        }

        return root;
    }
};
```

Java Recursion

```
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
```

```
    public TreeNode insertNode(TreeNode root, TreeNode node) {
        if (root == null) {
            return node;
        }
        if (root.val > node.val) {
            root.left = insertNode(root.left, node);
        } else {
            root.right = insertNode(root.right, node);
        }
        return root;
    }
}
```

题解 - 迭代

看过了以上递归版的题解，对于这个题来说，将递归转化为迭代的思路也是非常清晰易懂的。迭代比较当前节点的值和插入节点的值，到了二叉树的最后一层时选择是链接至左子结点还是右子节点。

C++

```
/**
 * 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: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    TreeNode* insertNode(TreeNode* root, TreeNode* node) {
        if (NULL == root) {
            return node;
        }

        TreeNode* tempNode = root;
        while (NULL != tempNode) {
            if (node->val <= tempNode->val) {
                if (NULL == tempNode->left) {
                    tempNode->left = node;
                    return root;
                }
                tempNode = tempNode->left;
            } else {
                if (NULL == tempNode->right) {
                    tempNode->right = node;
```

```
                    return root;
                }
                tempNode = tempNode->right;
            }
        }

        return root;
    }
};
```

源码分析

在 `NULL == tempNode->right` 或者 `NULL == tempNode->left` 时需要在链接完 `node` 后立即返回 `root`，避免死循环。

Java Iterative

```
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    public TreeNode insertNode(TreeNode root, TreeNode node) {
        // write your code here
        if (root == null) return node;
        if (node == null) return root;

        TreeNode rootcopy = root;
        while (root != null) {
            if (root.val <= node.val && root.right == null) {
                root.right = node;
                break;
            }
            else if (root.val > node.val && root.left == null) {
                root.left = node;
                break;
            }
            else if(root.val <= node.val) root = root.right;
            else root = root.left;
        }
        return rootcopy;
    }
}
```

Validate Binary Search Tree

Source

- lintcode: (95) Validate Binary Search Tree

```
Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.
Example
An example:

   1
  / \
 2   3
    /
   4
    \
     5
The above binary tree is serialized as "{1,2,3,#,#,4,#,#,5}".
```

题解1 - recursion

按照题中对二叉搜索树所给的定义递归判断，我们从递归的两个步骤出发分析：

1. 基本条件/终止条件 - 返回值需斟酌。
2. 递归步/条件递归 - 能使原始问题收敛。

终止条件好确定——当前节点为空，或者不符合二叉搜索树的定义，返回值分别是什么呢？先别急，分析下递归步试试先。递归步的核心步骤为比较当前节点的 `key` 和左右子节点的 `key` 大小，和定义不符则返回 `false`，否则递归处理。从这里可以看出在节点为空时应返回 `true`，由上层的其他条件判断。但需要注意的是这里不仅要考虑根节点与当前的左右子节点，**还需要考虑左子树中父节点的最小值和右子树中父节点的最大值**。否则程序在 `[10,5,15,#,#,6,20]` 这种 case 误判。

由于不仅需要考虑当前父节点，还需要考虑父节点的父节点... 故递归时需要引入上界和下界值。画图分析可知对于左子树我们需要比较父节点中最小值，对于右子树则是父节点中的最大值。又由于满足二叉搜索树的定义时，左子结点的值一定小于根节点，右子节点的值一定大于根节点，故无需比较所有父节点的值，使用递推即可得上界与下界，这里的实现非常巧妙。

C++ - long long

```
/**
 * 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: The root of binary tree.
     * @return: True if the binary tree is BST, or false
     */
    bool isValidBST(TreeNode *root) {
        if (root == NULL) return true;

        return helper(root, LLONG_MIN, LLONG_MAX);
    }

    bool helper(TreeNode *root, long long lower, long long upper) {
        if (root == NULL) return true;

        if (root->val <= lower || root->val >= upper) return false;
        bool isLeftValidBST = helper(root->left, lower, root->val);
        bool isRightValidBST = helper(root->right, root->val, upper);

        return isLeftValidBST && isRightValidBST;
    }
};
```

C++ - without long long

```
/**
 * 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: The root of binary tree.
     * @return: True if the binary tree is BST, or false
     */
    bool isValidBST(TreeNode *root) {
        if (root == NULL) return true;

        return helper(root, INT_MIN, INT_MAX);
```

```
    }

    bool helper(TreeNode *root, int lower, int upper) {
        if (root == NULL) return true;

        if (root->val <= lower || root->val >= upper) {
            bool right_max = root->val == INT_MAX && root->right == NULL;
            bool left_min = root->val == INT_MIN && root->left == NULL;
            if (!(right_max || left_min)) {
                return false;
            }
        }
        bool isLeftValidBST = helper(root->left, lower, root->val);
        bool isRightValidBST = helper(root->right, root->val, upper);

        return isLeftValidBST && isRightValidBST;
    }
};
```

Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: True if the binary tree is BST, or false
     */
    public boolean isValidBST(TreeNode root) {
        if (root == null) return true;

        return helper(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean helper(TreeNode root, long lower, long upper) {
        if (root == null) return true;
        // System.out.println("root.val = " + root.val + ", lower = " + lower + ", upper
        // left node value < root node value < right node value
        if (root.val >= upper || root.val <= lower) return false;
        boolean isLeftValidBST = helper(root.left, lower, root.val);
        boolean isRightValidBST = helper(root.right, root.val, upper);

        return isLeftValidBST && isRightValidBST;
    }
}
```

源码分析

为避免节点中出现整型的最大最小值，引入 long 型进行比较。有些 BST 的定义允许左子结点的值与根节点相同，此时需要更改比较条件为 `root.val > upper`. C++ 中 long 可能与 int 范围相同，故使用 long long. 如果不使用比 int 型更大的类型，那么就需要在相等时多加一些判断。

复杂度分析

递归遍历所有节点，时间复杂度为 $O(n)$, 使用了部分额外空间，空间复杂度为 $O(1)$.

题解2 - iteration

联想到二叉树的中序遍历。 TBD

Reference

- LeetCode: Validate Binary Search Tree 解题报告 - Yu's Garden - 博客园 - 提供了4种不同的方法，思路可以参考。

Search Range in Binary Search Tree

Source

- lintcode: (11) Search Range in Binary Search Tree

```
Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree.
Find all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x i
Return all the keys in ascending order.

Example
For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.

          20

       /        \

    8           22

  /     \

4       12
```

题解 - 中序遍历

中等偏易难度题，本题涉及到二叉查找树的按序输出，应马上联想到二叉树的中序遍历，对于二叉查找树而言，使用中序遍历即可得到有序元素。对每次访问的元素加以判断即可得最后结果，由于 OJ 上给的模板不适合递归处理，新建一个私有方法即可。

C++ In-order Recursion

```
/**
 * 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: The root of the binary search tree.
     * @param k1 and k2: range k1 to k2.
     * @return: Return all keys that k1<=key<=k2 in ascending order.
```

```
     */
    vector<int> searchRange(TreeNode* root, int k1, int k2) {
        vector<int> result;
        inorder_dfs(result, root, k1, k2);

        return result;
    }

private:
    void inorder_dfs(vector<int> &ret, TreeNode *root, int k1, int k2) {
        if (NULL == root) {
            return;
        }

        inorder_dfs(ret, root->left, k1, k2);
        if ((root->val >= k1) && (root->val <= k2)) {
            ret.push_back(root->val);
        }
        inorder_dfs(ret, root->right, k1, k2);
    }
};
```

源码分析

以上为题解思路的简易实现，可以优化的地方为「剪枝过程」的处理——不递归遍历不可能有解的节点。优化后的 `inorder_dfs` 如下：

```
    void inorder_dfs(vector<int> &ret, TreeNode *root, int k1, int k2) {
        if (NULL == root) {
            return;
        }

        if ((NULL != root->left) && (root->val > k1)) {
            inorder_dfs(ret, root->left, k1, k2);
        } // cut-off for left sub tree

        if ((root->val >= k1) && (root->val <= k2)) {
            ret.push_back(root->val);
        } // add valid value

        if ((NULL != root->right) && (root->val < k2)) {
            inorder_dfs(ret, root->right, k1, k2);
        } // cut-off for right sub tree
    }
```

> 「剪枝」的判断条件容易出错，应将当前节点的值与 `k1` 和 `k2` 进行比较而不是其左子节点或右子节点的值。

Convert Sorted Array to Binary Search Tree

Source

- leetcode: Convert Sorted Array to Binary Search Tree | LeetCode OJ
- lintcode: (177) Convert Sorted Array to Binary Search Tree With Minimal Height

```
Given an array where elements are sorted in ascending order,
convert it to a height balanced BST.

Given a sorted (increasing order) array,
Convert it to create a binary tree with minimal height.

Example
Given [1,2,3,4,5,6,7], return

     4
   /   \
  2     6
 / \    / \
1   3  5   7
Note
There may exist multiple valid solutions, return any of them.
```

题解 - 折半取中

将二叉搜索树按中序遍历即可得升序 key 这个容易实现，但反过来由升序 key 逆推生成二叉搜索树呢？按照二叉搜索树的定义我们可以将较大的 key 链接到前一个树的最右侧节点，这种方法实现极其简单，但是无法达到本题「树高平衡-左右子树的高度差绝对值不超过1」的要求，因此只能另辟蹊径以达到「平衡二叉搜索树」的要求。

要达到「平衡二叉搜索树」这个条件，我们首先应从「平衡二叉搜索树」的特性入手。简单起见，我们先考虑下特殊的满二叉搜索树，满二叉搜索树的一个重要特征就是各根节点的 key 不小于左子树的 key ，而小于右子树的所有 key；另一个则是左右子树数目均相等，那么我们只要能将所给升序序列分成一大一小的左右两半部分即可满足题目要求。又由于此题所给的链表结构中仅有左右子树的链接而无指向根节点的链接，故我们只能从中间的根节点进行分析逐层往下递推直至取完数组中所有 key, 数组中间的索引自然就成为了根节点。由于 OJ 上方法入口参数仅有升序序列，方便起见我们可以另写一私有方法，加入 `start` 和 `end` 两个参数，至此递归模型初步建立。

C++

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
```

```
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode *sortedArrayToBST(vector<int> &num) {
        if (num.empty()) {
            return NULL;
        }

        return middleNode(num, 0, num.size() - 1);
    }

private:
    TreeNode *middleNode(vector<int> &num, const int start, const int end) {
        if (start > end) {
            return NULL;
        }

        TreeNode *root = new TreeNode(num[start + (end - start) / 2]);
        root->left = middleNode(num, start, start + (end - start) / 2 - 1);
        root->right = middleNode(num, start + (end - start) / 2 + 1, end);

        return root;
    }
};
```

Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param A: an integer array
     * @return: a tree node
     */
    public TreeNode sortedArrayToBST(int[] A) {
        if (A == null || A.length == 0) return null;

        return helper(A, 0, A.length - 1);
    }

    private TreeNode helper(int[] nums, int start, int end) {
        if (start > end) return null;

        int mid = start + (end - start) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        root.left = helper(nums, start, mid - 1);
```

```
        root.right = helper(nums, mid + 1, end);

        return root;
    }
}
```

源码分析

从题解的分析中可以看出中间根节点的建立至关重要！由于数组是可以进行随机访问的，故可取数组中间的索引为根节点，左右子树节点可递归求解。虽然这种递归的过程和「二分搜索」的模板非常像，但是切记本题中根据所给升序序列建立平衡二叉搜索树的过程中需要做到**不重不漏**，故边界处理需要异常小心，不能再套用 `start + 1 < end` 的模板了。

复杂度分析

递归调用 `middleNode` 方法时每个 `key` 被访问一次，故时间复杂度可近似认为是 $O(n)$.

Reference

- Convert Sorted Array to Binary Search Tree | 九章算法

Convert Sorted List to Binary Search Tree

Source

- leetcode - Convert Sorted List to Binary Search Tree | LeetCode OJ
- lintcode - (106) Convert Sorted List to Binary Search Tree

```
Given a singly linked list where elements are sorted in ascending order,
convert it to a height balanced BST.
```

题解 - 折半取中

题 Convert Sorted Array to Binary Search Tree | Data Structure and Algorithm 的升级版，不过这里把「有序数组」换成了「有序链表」。我们可以参考上题的题解思路，思考如何才能在链表中找到「中间节点」。对于本题的单向链表来说，要想知道中间位置的节点，则必须需要知道链表的长度，因此我们就自然联想到了可以通过遍历链表来求得其长度。求得长度我们就知道了链表中间位置节点的索引了，进而根据头节点和当前节点则可将链表分为左右两半形成递归模型。到这里还只能算是解决了问题的一半，这道题另一比较麻烦的地方在于边界条件的取舍，很难第一次就 AC, 下面结合代码做进一步的分析。

C++

```cpp
/**
 * 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) {
        if (NULL == head) {
            return NULL;
        }

        // get the size of List
        ListNode *node = head;
        int len = 0;
        while (NULL != node) {
            node = node->next;
            ++len;
        }

        return buildBSTHelper(head, len);
    }

private:
    TreeNode *buildBSTHelper(ListNode *head, int length) {
        if (NULL == head || length <= 0) {
            return NULL;
        }

        // get the middle ListNode as root TreeNode
        ListNode *lnode = head;
        int count = 0;
        while (count < length / 2) {
            lnode = lnode->next;
            ++count;
        }

        TreeNode *root = new TreeNode(lnode->val);
        root->left = buildBSTHelper(head, length / 2);
        root->right = buildBSTHelper(lnode->next, length - 1 - length / 2);

        return root;
    }
};
```

源码分析

1. 异常处理。
2. 获取链表长度。
3. `buildBSTHelper` 输入参数为表头节点地址以及相应的链表长度，递归获取根节点、左节点和右节点。

其中 `buildBSTHelper` 的边界处理很有技巧，首先是递推的终止条件，头节点为 `NULL` 时显然应该返回 `NULL`. 但 `length` 的终止条件又如何确定？拿不定主意时就用几个简单例子来试试，比如 `1`，`1->2`，`1->2->3`.

先来分析下给 `buildBSTHelper` 传入的 `length` 的含义——从表头节点 `head` 开始往后递推长度为 `length` 的链表。故 `length` 为0时表示不访问链表中的任一节点，也就是说应该返回 `NULL`.

再来分析链表的中间位置如何确定，我们引入计数器 `count` 来表示**目前需要遍历 `count` 个链表节点数目**才能得到中间位置的节点。看看四种不同链表长度下的表现。

1. 链表长度为1时，中间位置即为自身，计数器的值为0.
2. 链表长度为2时，中间位置可选第一个节点，也可选第二个节点，相应的计数器值为0或1.
3. 链表长度为3时，中间位置为第二个节点，相应的计数器应为1，表示从表头节点往后递推一个节点。
4. 链表长度为4时，... 计数器的值为1或者2.

从以上四种情况我们可以推断出 `count` 的值可取为 `length / 2` 或者 `length / 2 + 1`，简单起见我们先取 `length / 2` 试试，对应的边界条件即为 `count < length / 2`， `count` 初始值为0. 经过 `count` 次迭代后，目前 `lnode` 即为所需的链表中间节点，取出其值初始化为 `TreeNode` 的根节点。

确定根节点后还需要做的事情就是左子树和右子树中链表头和链表长度的取舍。首先来看看左子树根节点的确定， **`count` 的含义为到达中间节点前遍历过的链表节点数目，那么从另一方面来说它就是前半部分链表的长度！** 故将此长度 `length / 2` 作为得到左子树根节点所需的链表长度参数。除掉链表前半部分节点和中间位置节点这两部分外，剩下的链表长度即为 `length - 1 - length / 2`.

> `length - 1 - length / 2 != length / 2 - 1`

有没有觉得可以进一步化简为 `length / 2 - 1`？我首先也是这么做的，后来发现一直遇到 `TERMSIG=11` 错误信息，这种错误一般是指针乱指或者指针未初始化就去访问。但自己仔细检查后发现并没有这种错误，于是乎在本地做单元测试，发现原来是死循环造成了栈空间溢出(猜的)！也就是说边界条件有问题！可自己的分析明明是没啥问题的啊...

在这种情况下我默默地打开了九章的参考代码，发现他们竟然没有用 `length / 2 - 1`，而是 `length - 1 - length / 2`. 立马意识到这两者可能并不相等。用错误数据试了下，长度为1或者3时两者即不相等。知道对于整型数来说， `1 / 2` 为0，但是却没能活学活用，血泪的教训。:-(一个美好的下午就没了。

在测试出错的时候，还是要相信测试数据的力量，而不是凭自己以前认为对的方式去解决问题。

复杂度分析

首先遍历链表得到链表长度，复杂度为 $O(n)$. 递归遍历链表时，每个链表节点被访问一次，故时间复杂度为 $O(n)$, 两者加起来总的时间复杂度仍为 $O(n)$.

进一步简化代码

```
class Solution {
public:
    TreeNode *sortedListToBST(ListNode *head) {
        int length = 0;
        ListNode *curr = head;
        while (curr != NULL) {
            curr = curr->next;
            ++length;
        }
        return helper(head, length);
    }
private:
    TreeNode *helper(ListNode *&pos, int length) {
        if (length <= 0) {
            return NULL;
        }
```

```
        TreeNode *left = helper(pos, length / 2);
        TreeNode *root = new TreeNode(pos->val); // the sequence cannot be changed!
                                                 // this is important difference of the s
        pos = pos->next;
        root->left = left;
        root->right = helper(pos, length - length / 2 - 1);
        return root;
    }
};
```

源码分析

1. 可以进一步简化 helper 函数代码，注意参数的接口设计。
2. 即是把传入的链表指针向前递进 n 步，并返回经过的链表节点转化成的二分查找树的根节点。
3. 注意注释中的那两句实现，`new root` 和 `new left` 不可调换顺序。这才是精简的要点。但是这种方法不如上面的分治法容易理解。

O(nlogn) 的实现，避免 length 边界

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    public TreeNode sortedListToBST(ListNode head) {
        if (head == null) {
            return null;
        }
        return helper(head);
    }

    private TreeNode helper(ListNode head) {
        if (head == null) {
```

```
            return null;
        }
        if (head.next == null) {
            return new TreeNode(head.val);
        }

        ListNode pre = null;
        ListNode slow = head, fast = head;

        while (fast != null && fast.next != null) {
            pre = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        pre.next = null;

        TreeNode root = new TreeNode(slow.val);
        TreeNode L = helper(head);
        TreeNode R = helper(slow.next);
        root.left = L;
        root.right = R;

        return root;
    }
}
```

源码分析

1. 如果想避免上述 length 边界搞错的问题，可以使用分治法遍历树求中点的方法。
2. 但这种时间复杂度是 $O(nlogn)$，性能上还是比 $O(n)$ 差一点。

Reference

- Convert Sorted List to Binary Search Tree | 九章算法

Binary Search Tree Iterator

Source

- lintcode: (86) Binary Search Tree Iterator

```
Design an iterator over a binary search tree with the following rules:

- Elements are visited in ascending order (i.e. an in-order traversal)
- next() and hasNext() queries run in O(1) time in average.

Example
For the following binary search tree, in-order traversal by using iterator is [1, 6, 10,

           10
         /    \
        1      11
         \       \
             6       12

Challenge
Extra memory usage O(h), h is the height of the tree.

Super Star: Extra memory usage O(1)
```

题解 - 中序遍历

仍然考的是中序遍历，但是是非递归实现。其实这道题等价于写一个二叉树中序遍历的迭代器。需要内置一个栈，一开始先存储到最左叶子节点的路径。在遍历的过程中，只要当前节点存在右子树，则进入右子树，存储从此处开始到当前子树里最左叶子节点的路径。

Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 * Example of iterate a tree:
 * Solution iterator = new Solution(root);
 * while (iterator.hasNext()) {
 *    TreeNode node = iterator.next();
 *    do something for node
 * }
```

```
 */
public class Solution {
    private Stack<TreeNode> stack = new Stack<>();
    private TreeNode curt;

    // @param root: The root of binary tree.
    public Solution(TreeNode root) {
        curt = root;
    }

    //@return: True if there has next node, or false
    public boolean hasNext() {
        return (curt != null || !stack.isEmpty()); //important to judge curt != null
    }

    //@return: return next node
    public TreeNode next() {
        while (curt != null) {
            stack.push(curt);
            curt = curt.left;
        }

        curt = stack.pop();
        TreeNode node = curt;
        curt = curt.right;

        return node;
    }
}
```

源码分析

1. 这里容易出错的是 `hasNext()` 函数中的判断语句，不能漏掉 `curt != null`。
2. 如果是 leetcode 上的原题，由于接口不同，则不需要维护 current 指针。

Exhaustive Search - 穷竭搜索

穷竭搜索又称暴力搜索，指代将所有可能性列出来，然后再在其中寻找满足题目条件的解。常用求解方法和工具有：

1. 递归函数
2. 栈
3. 队列
4. 深度优先搜索(DFS, Depth-First Search)，又常称为回溯法
5. 广度优先搜索(BFS, Breadth-First Search)

1, 2, 3 往往在深搜或者广搜中体现。

DFS

DFS 通常从某个状态开始，根据特定的规则转移状态，直至无法转移(节点为空)，然后回退到之前一步状态，继续按照指定规则转移状态，直至遍历完所有状态。

回溯法包含了多类问题，模板类似。

排列组合模板->搜索问题(是否要排序，哪些情况要跳过)

使用回溯法的一般步骤：

1. 确定所给问题的解空间：首先应明确定义问题的解空间，解空间中至少包含问题的一个解。
2. 确定结点的扩展搜索规则
3. 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

BFS

BFS 从某个状态开始，搜索**所有可以到达的状态**，转移顺序为『初始状态->只需一次转移就可到达的所有状态->只需两次转移就可到达的所有状态->...』，所以对于同一个状态，BFS 只搜索一次，故时间复杂度为 $O(states \times transfer_methods)$. BFS 通常配合队列一起使用，搜索时先将状态加入到队列中，然后从队列顶端不断取出状态，再把从该状态可转移到的状态中尚未访问过的部分加入队列，知道队列为空或已找到解。因此 BFS 适合用于『由近及远』的搜索，比较适合用于求解最短路径、最少操作之类的问题。

Reference

- 《挑战程序设计竞赛》Chaper 2.1 p26 最基础的“穷竭搜索”
- Steven Skiena: Lecture15 - Backtracking
- 全面解析回溯法：算法框架与问题求解 - 五岳 - 博客园
- 五大常用算法之四：回溯法 - 红脸书生 - 博客园
- 演算法筆記 - Backtracking

Subsets - 子集

Source

- leetcode: Subsets | LeetCode OJ
- lintcode: (17) Subsets

```
Given a set of distinct integers, return all possible subsets.

Note
Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

Example
If S = [1,2,3], a solution is:

[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

题解

子集类问题类似Combination，以输入数组 `[1, 2, 3]` 分析，根据题意，最终返回结果中子集类的元素应该按照升序排列，故首先需要对原数组进行排序。题目的第二点要求是子集不能重复，至此原题即转化为数学中的组合问题。我们首先尝试使用 DFS 进行求解，大致步骤如下：

1. `[1] -> [1, 2] -> [1, 2, 3]`
2. `[2] -> [2, 3]`
3. `[3]`

将上述过程转化为代码即为对数组遍历，每一轮都保存之前的结果并将其依次加入到最终返回结果中。

Python

```python
class Solution:
    # @param {integer[]} nums
    # @return {integer[][]}
    def subsets(self, nums):
        if nums is None:
            return []
```

```
        result = []
        nums.sort()
        self.dfs(nums, 0, [], result)
        return result

    def dfs(self, nums, pos, list_temp, ret):
        # append new object with []
        ret.append([] + list_temp)

        for i in xrange(pos, len(nums)):
            list_temp.append(nums[i])
            self.dfs(nums, i + 1, list_temp, ret)
            list_temp.pop()
```

C++

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int> > result;
        if (nums.empty()) return result;

        sort(nums.begin(), nums.end());
        vector<int> list;
        dfs(nums, 0, list, result);

        return result;
    }

private:
    void dfs(vector<int>& nums, int pos, vector<int> &list,
             vector<vector<int> > &ret) {

        ret.push_back(list);

        for (int i = pos; i < nums.size(); ++i) {
            list.push_back(nums[i]);
            dfs(nums, i + 1, list, ret);
            list.pop_back();
        }
    }
};
```

Java

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
        if (nums == null || nums.length == 0) {
            return result;
        }
```

```
        Arrays.sort(nums);
        dfs(nums, 0, list, result);

        return result;
    }

    private void dfs(int[] nums, int pos, List<Integer> list,
                     List<List<Integer>> ret) {

        // add temp result first
        ret.add(new ArrayList<Integer>(list));

        for (int i = pos; i < nums.length; i++) {
            list.add(nums[i]);
            dfs(nums, i + 1, list, ret);
            list.remove(list.size() - 1);
        }
    }
}
```

源码分析

Java 和 Python 的代码中在将临时list 添加到最终结果时新生成了对象，(Python 使用 `[] +`), 否则最终返回结果将随着 `list` 的变化而变化。

Notice: backTrack(num, i + 1, list, ret);中的『i + 1』不可误写为『pos + 1』，因为 `pos` 用于每次大的循环， `i` 用于内循环，第一次写subsets的时候在这坑了很久... :(

回溯法可用图示和函数运行的堆栈图来理解，强烈建议**使用图形和递归的思想**分析，以数组 `[1, 2, 3]` 进行分析。下图所示为 `list` 及 `result` 动态变化的过程，箭头向下表示 `list.add` 及 `result.add` 操作，箭头向上表示 `list.remove` 操作。

复杂度分析

对原有数组排序，时间复杂度近似为 $O(n \log n)$. 状态数为所有可能的组合数 $O(2^n)$, 生成每个状态所需的时间复杂度近似为 $O(1)$, 如 `[1] -> [1, 2]`, 故总的时间复杂度近似为 $O(2^n)$.

使用了临时空间 `list` 保存中间结果， `list` 最大长度为数组长度，故空间复杂度近似为 $O(n)$.

Reference

- [NineChap 1.2] Permutation - Woodstock Blog
- 九章算法 - subsets模板
- LeetCode: Subsets 解题报告 - Yu's Garden - 博客园

Unique Subsets

Source

- lintcode: (18) Unique Subsets

```
Given a list of numbers that may has duplicate numbers, return all possible subsets

Note
Each element in a subset must be in non-descending order.

The ordering between two subsets is free.

The solution set must not contain duplicate subsets.

Example
If S = [1,2,2], a solution is:

[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

题解

此题在上一题的基础上加了有重复元素的情况，因此需要对回溯函数进行一定的剪枝，对于排列组合的模板程序，剪枝通常可以从两个地方出发，一是在返回结果 `result.add` 之前进行剪枝，另一个则是在 `list.add` 处剪枝，具体使用哪一种需要视情况而定，哪种简单就选谁。

由于此题所给数组不一定有序，故首先需要排序。有重复元素对最终结果的影响在于重复元素最多只能出现 `n` 次(重复个数为n时)。具体分析过程如下(此分析过程改编自 九章算法)。

以 $[1, 2_1, 2_2]$ 为例，若不考虑重复，组合有 $[], [1], [1, 2_1], [1, 2_1, 2_2], [1, 2_2], [2_1], [2_1, 2_2], [2_2]$. 其中重复的有 $[1, 2_2], [2_2]$. 从中我们可以看出只能从重复元素的第一个持续往下添加到列表中，而不能取第二个或之后的重复元素。参考上一题Subsets的模板，能代表「重复元素的第一个」即为 for 循环中的 `pos` 变量， `i == pos` 时， `i` 处所代表的变量即为某一层遍历中得「第一个元素」，因此去重时只需判断 `i != pos && s[i] == s[i - 1]` .

C++

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

```
     * @param S: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    vector<vector<int> > subsetsWithDup(const vector<int> &S) {
        vector<vector<int> > result;
        if (S.empty()) {
            return result;
        }

        vector<int> list;
        vector<int> source(S);
        sort(source.begin(), source.end());
        backtrack(result, list, source, 0);

        return result;
    }

private:
    void backtrack(vector<vector<int> > &ret, vector<int> &list,
              vector<int> &s, int pos) {

        ret.push_back(list);

        for (int i = pos; i != s.size(); ++i) {
            if (i != pos && s[i] == s[i - 1]) {
                continue;
            }
            list.push_back(s[i]);
            backtrack(ret, list, s, i + 1);
            list.pop_back();
        }
    }
};
```

Reference

- Subsets II | 九章算法

Permutations

Source

- leetcode: Permutations | LeetCode OJ
- lintcode: (15) Permutations

```
Given a list of numbers, return all possible permutations.

Example
For nums [1,2,3], the permutaions are:

[

    [1,2,3],

    [1,3,2],

    [2,1,3],

    [2,3,1],

    [3,1,2],

    [3,2,1]

]

Challenge
Do it without recursion
```

题解1 - Recursion(using subsets template)

排列常见的有数字全排列，字符串排列等。

使用之前 Subsets 的模板，但是在取结果时只能取 `list.size() == nums.size()` 的解，且在添加list元素的时候需要注意除重以满足全排列的要求。此题假设前提为输入数据中无重复元素。

Python

```python
class Solution:
    """
    @param nums: A list of Integers.
    @return: A list of permutations.
    """
    def permute(self, nums):
        alist = []
        result = [];
        if not nums:
```

```
            return result

        self.helper(nums, alist, result)

        return result

    def helper(self, nums, alist, ret):
        if len(alist) == len(nums):
            # new object
            ret.append([] + alist)
            return

        for i, item in enumerate(nums):
            if item not in alist:
                alist.append(item)
                self.helper(nums, alist, ret)
                alist.pop()
```

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers.
     * @return: A list of permutations.
     */
    vector<vector<int> > permute(vector<int> nums) {
        vector<vector<int> > result;
        if (nums.empty()) {
            return result;
        }

        vector<int> list;
        backTrack(result, list, nums);

        return result;
    }

private:
    void backTrack(vector<vector<int> > &result, vector<int> &list, \
                   vector<int> &nums) {
        if (list.size() == nums.size()) {
            result.push_back(list);
            return;
        }

        for (int i = 0; i != nums.size(); ++i) {
            // remove the element belongs to list
            if (find(list.begin(), list.end(), nums[i]) != list.end()) {
                continue;
            }
            list.push_back(nums[i]);
            backTrack(result, list, nums);
            list.pop_back();
        }
    }
};
```

Java

```
class Solution {
    /**
     * @param nums: A list of integers.
     * @return: A list of permutations.
     */
    public ArrayList<ArrayList<Integer>> permute(ArrayList<Integer> nums) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        ArrayList<Integer> list = new ArrayList<Integer>();

        if (nums == null || nums.isEmpty()) return result;

        helper(nums, list, result);

        return result;
    }

    private void helper(ArrayList<Integer> nums, ArrayList<Integer> list,
                        ArrayList<ArrayList<Integer>> ret
                        ) {

        if (list.size() == nums.size()) {
            ret.add(new ArrayList<Integer>(list));
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (list.contains(nums.get(i))) continue;

            list.add(nums.get(i));
            helper(nums, list, ret);
            list.remove(list.size() - 1);
        }
    }
}
```

源码分析

在除重时使用了标准库 `find` (不可使用时间复杂度更低的 `binary_search` ，因为 `list` 中元素不一定有序)，时间复杂度为 $O(N)$, 也可使用 `hashmap` 记录 `nums` 中每个元素是否被添加到 `list` 中，这样一来空间复杂度为 $O(N)$, 查找的时间复杂度为 $O(1)$.

在 `list.size() == nums.size()` 时，已经找到需要的解，及时 `return` 避免后面不必要的 `for` 循环调用开销。

使用回溯法解题的**关键在于如何确定正确解及排除不符条件的解(剪枝)**。

复杂度分析

以状态数来分析，最终全排列个数应为 $n!$, 每个节点被遍历的次数为 $(n-1)!$, 故节点共被遍历的状态数为

$O(n!)$, 此为时间复杂度的下界，因为这里只算了合法条件下的遍历状态数。若不对 list 中是否包含 nums[i] 进行检查，则总的状态数应为 n^n 种。

由于最终的排列结果中每个列表的长度都为 n, 各列表的相同元素并不共享，故时间复杂度的下界为 $O(n \cdot n!)$, 上界为 $n \cdot n^n$. 实测 `helper` 中 for 循环的遍历次数在 $O(2n \cdot n!)$ 以下，注意这里的时间复杂度并不考虑查找列表里是否包含重复元素。

题解2 - Recursion

与题解1基于 subsets 的模板不同，这里我们直接从全排列的数学定义本身出发，要求给定数组的全排列，可将其模拟为某个袋子里有编号为1到 n 的球，将其放入 n 个不同的盒子怎么放？基本思路就是从袋子里逐个拿球放入盒子，直到袋子里的球拿完为止，拿完时即为一种放法。

Python

```
class Solution:
    # @param {integer[]} nums
    # @return {integer[][]}
    def permute(self, nums):
        if nums is None:
            return [[]]
        elif len(nums) <= 1:
            return [nums]

        result = []
        for i, item in enumerate(nums):
            for p in self.permute(nums[:i] + nums[i + 1:]):
                result.append(p + [item])

        return result
```

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers.
     * @return: A list of permutations.
     */
    vector<vector<int> > permute(vector<int>& nums) {
        vector<vector<int> > result;

        if (nums.size() == 1) {
            result.push_back(nums);
            return result;
        }

        for (int i = 0; i < nums.size(); ++i) {
            vector<int> nums_new = nums;
            nums_new.erase(nums_new.begin() + i);
```

```
            vector<vector<int> > res_tmp = permute(nums_new);
            for (int j = 0; j < res_tmp.size(); ++j) {
                vector<int> temp = res_tmp[j];
                temp.push_back(nums[i]);
                result.push_back(temp);
            }
        }

        return result;
    }
};
```

Java

```
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> numsList = new ArrayList<Integer>();

        if (nums == null) {
            return result;
        } else {
            // convert int[] to List<Integer>
            for (int item : nums) numsList.add(item);
        }

        if (nums.length <= 1) {
            result.add(numsList);
            return result;
        }

        for (int i = 0; i < nums.length; i++) {
            int[] numsNew = new int[nums.length - 1];
            System.arraycopy(nums, 0, numsNew, 0, i);
            System.arraycopy(nums, i + 1, numsNew, i, nums.length - i - 1);

            List<List<Integer>> resTemp = permute(numsNew);
            for (List<Integer> temp : resTemp) {
                temp.add(nums[i]);
                result.add(temp);
            }
        }

        return result;
    }
}
```

源码分析

Python 中使用 `len()` 时需要防止 `None`，递归终止条件为数组中仅剩一个元素或者为空，否则遍历 `nums` 数组，取出第 `i` 个元素并将其加入至最终结果。 `nums[:i] + nums[i + 1:]` 即为去掉第 `i` 个元素后的新列表。

Java 中 ArrayList 和 List 的类型转换需要特别注意。

复杂度分析

由于取的结果都是最终结果，无需去重判断，故时间复杂度为 $O(n!)$, 但是由于 `nums[:i] + nums[i + 1:]` 会产生新的列表，实际运行会比第一种方法慢不少。

题解3 - Iteration

递归版的程序比较简单，咱们来个迭代的实现。非递归版的实现也有好几种，这里基于 C++ STL 中 `next_permutation` 的字典序实现方法。参考 Wikipedia 上的字典序算法，大致步骤如下：

1. 从后往前寻找索引满足 `a[k] < a[k + 1]` , 如果此条件不满足，则说明已遍历到最后一个。
2. 从后往前遍历，找到第一个比 `a[k]` 大的数 `a[l]` , 即 `a[k] < a[l]` .
3. 交换 `a[k]` 与 `a[l]` .
4. 反转 `k + 1 ~ n` 之间的元素。

Python

```
class Solution:
    # @param {integer[]} nums
    # @return {integer[][]}
    def permute(self, nums):
        if nums is None:
            return [[]]
        elif len(nums) <= 1:
            return [nums]

        # sort nums first
        nums.sort()

        result = []
        while True:
            result.append([] + nums)
            # step1: find nums[i] < nums[i + 1], Loop backwards
            i = 0
            for i in xrange(len(nums) - 2, -1, -1):
                if nums[i] < nums[i + 1]:
                    break
                elif i == 0:
                    return result
            # step2: find nums[i] < nums[j], Loop backwards
            j = 0
            for j in xrange(len(nums) - 1, i, -1):
                if nums[i] < nums[j]:
                    break
            # step3: swap betwenn nums[i] and nums[j]
            nums[i], nums[j] = nums[j], nums[i]
            # step4: reverse between [i + 1, n - 1]
            nums[i + 1:len(nums)] = nums[len(nums) - 1:i:-1]

        return result
```

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers.
     * @return: A list of permutations.
     */
    vector<vector<int> > permute(vector<int>& nums) {
        vector<vector<int> > result;
        if (nums.empty() || nums.size() <= 1) {
            result.push_back(nums);
            return result;
        }

        // sort nums first
        sort(nums.begin(), nums.end());
        for (;;) {
            result.push_back(nums);

            // step1: find nums[i] < nums[i + 1]
            int i = 0;
            for (i = nums.size() - 2; i >= 0; --i) {
                if (nums[i] < nums[i + 1]) {
                    break;
                } else if (0 == i) {
                    return result;
                }
            }

            // step2: find nums[i] < nums[j]
            int j = 0;
            for (j = nums.size() - 1; j > i; --j) {
                if (nums[i] < nums[j]) break;
            }

            // step3: swap betwenn nums[i] and nums[j]
            int temp = nums[j];
            nums[j] = nums[i];
            nums[i] = temp;

            // step4: reverse between [i + 1, n - 1]
            reverse(nums, i + 1, nums.size() - 1);
        }
        return result;
    }

private:
    void reverse(vector<int>& nums, int start, int end) {
        for (int i = start, j = end; i < j; ++i, --j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
};
```

Java

```
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> numsList = new ArrayList<Integer>();

        if (nums == null) {
            return result;
        } else {
            // convert int[] to List<Integer>
            for (int item : nums) numsList.add(item);
            if (nums.length <= 1) {
                result.add(numsList);
                return result;
            }
        }

        Collections.sort(numsList);

        boolean flag = true;
        while (flag) {
            result.add(new ArrayList<Integer>(numsList));
            // step1: find nums[i] < nums[i + 1]
            int i = 0;
            for (i = nums.length - 2; i >= 0; i--) {
                if (numsList.get(i) < numsList.get(i + 1)) {
                    break;
                } else if (i == 0) {
                    return result;
                }
            }
            // step2: find nums[i] < nums[j]
            int j = 0;
            for (j = nums.length - 1; j > i; j--) {
                if (numsList.get(i) < numsList.get(j)) break;
            }
            // step3: swap betwenn nums[i] and nums[j]
            Collections.swap(numsList, i, j);
            // step4: reverse between [i + 1, n - 1]
            reverse(numsList, i + 1, nums.length - 1);
        }

        return result;
    }

    private void reverse(List<Integer> nums, int start, int end) {
        for (int i = start, j = end; i < j; i++, j--) {
            Collections.swap(nums, i, j);
        }
    }
}
```

源码分析

Java 中需要使用 `while (flag)`，否则编译可能通不过。其他细节如边界条件和 corner case 需要注意下。

复杂度分析

除了将 $n!$ 个元素添加至最终结果外，首先对元素排序，时间复杂度近似为 $O(n \log n)$, 反转操作近似为 $O(n)$, 故总的时间复杂度为 $O(n!)$. 除了保存结果的 `result` 外，其他空间可忽略不计，所以此题用生成器来实现较为高效，扩展题可见底下的 Python itertools 中的实现，从 n 个元素中选出 m 个进行全排列。

Reference

- Permutation Generation - Robert Sedgewick 的大作，总结了诸多 Permutation 的产生方法。
- Next lexicographical permutation algorithm - 此题非递归方法更为详细的解释。
- Permutation - Wikipedia, the free encyclopedia - 字典序实现。
- Programming Interview Questions 11: All Permutations of String | Arden DertatArden Dertat
- algorithm - complexity of recursive string permutation function - Stack Overflow
- [leetcode]Permutations @ Python - 南郭子綦 - 博客园
- [leetcode] permutations的讨论 - tuantuanls的专栏 - 博客频道 - CSDN.NET
- 非递归排列算法（Permutation Generation）
- 闲谈permutations | HelloYou
- 9.7. itertools — Functions creating iterators for efficient looping — Python 2.7.10 documentation

Unique Permutations

Source

- lintcode: (16) Unique Permutations

```
Given a list of numbers with duplicate number in it. Find all unique permutations.

Example
For numbers [1,2,2] the unique permutations are:

[

    [1,2,2],

    [2,1,2],

    [2,2,1]

]

Challenge
Do it without recursion.
```

题解

在上题的基础上进行剪枝，剪枝的过程和 Unique Subsets 一题极为相似。为了便于分析，我们可以先分析简单的例子，以 $[1, 2_1, 2_2]$ 为例。按照上题 Permutations 的解法，我们可以得到如下全排列。

1. $[1, 2_1, 2_2]$
2. $[1, 2_2, 2_1]$
3. $[2_1, 1, 2_2]$
4. $[2_1, 2_2, 1]$
5. $[2_2, 1, 2_1]$
6. $[2_2, 2_1, 1]$

从以上结果我们注意到 `1` 和 `2` 重复， `5` 和 `3` 重复， `6` 和 `4` 重复，从重复的解我们可以发现其共同特征均是第二个 2_2 在前，而第一个 2_1 在后，因此我们的**剪枝方法为：对于有相同的元素来说，我们只取不重复的一次**。嗯，这样说还是有点模糊，下面以 $[1, 2_1, 2_2]$ 和 $[1, 2_2, 2_1]$ 进行说明。

首先可以确定 $[1, 2_1, 2_2]$ 是我们要的一个解，此时 `list` 为 $[1, 2_1, 2_2]$, 经过两次 `list.pop_back()` 之后， `list` 为 $[1]$, 如果不进行剪枝，那么接下来要加入 `list` 的将为 2_2, 那么我们剪枝要做的就是避免将 2_2 加入到 `list` 中，如何才能做到这一点呢？我们仍然从上述例子出发进行分析，在第一次加入 2_2 时，相

对应的 `visited[1]` 为 `true` (对应 2_1)，而在第二次加入 2_2 时，相对应的 `visited[1]` 为 `false` ，因为在 `list` 为 $[1, 2_1]$ 时，执行 `list.pop_back()` 后即置为 `false` 。

一句话总结即为：在遇到当前元素和前一个元素相等时，如果前一个元素 `visited[i - 1] == false` , 那么我们就跳过当前元素并进入下一次循环，这就是剪枝的关键所在。另一点需要特别注意的是这种剪枝的方法能使用的前提是提供的 `nums` 是有序数组，否则无效。

C++

```
class Solution {
public:
    /**
     * @param nums: A list of integers.
     * @return: A list of unique permutations.
     */
    vector<vector<int> > permuteUnique(vector<int> &nums) {
        vector<vector<int> > ret;
        if (nums.empty()) {
            return ret;
        }

        // important! sort before call `backTrack`
        sort(nums.begin(), nums.end());
        vector<bool> visited(nums.size(), false);
        vector<int> list;
        backTrack(ret, list, visited, nums);

        return ret;
    }

private:
    void backTrack(vector<vector<int> > &result, vector<int> &list, \
                   vector<bool> &visited, vector<int> &nums) {
        if (list.size() == nums.size()) {
            result.push_back(list);
            // avoid unnecessary call for `for loop`, but not essential
            return;
        }

        for (int i = 0; i != nums.size(); ++i) {
            if (visited[i] || (i != 0 && nums[i] == nums[i - 1] \
                && !visited[i - 1])) {
                continue;
            }
            visited[i] = true;
            list.push_back(nums[i]);
            backTrack(result, list, visited, nums);
            list.pop_back();
            visited[i] = false;
        }
    }
};
```

源码分析

Unique Subsets 和 Unique Permutations 的源码模板非常经典！建议仔细研读并体会其中奥义。

后记：在理解 Unique Subsets 和 Unique Permutations 的模板我花了差不多一整天时间才基本理解透彻，建议在想不清楚某些问题时先分析简单的问题，在纸上一步一步分析直至理解完全。

Reference

- Permutation II | 九章算法

Next Permutation

Source

- lintcode: (52) Next Permutation

```
Given a list of integers, which denote a permutation.

Find the next permutation in ascending order.

Example
For [1,3,2,3], the next permutation is [1,3,3,2]

For [4,3,2,1], the next permutation is [1,2,3,4]

Note
The list may contains duplicate integers.
```

题解

找下一个升序排列，C++ STL 源码剖析一书中有提及，Permutations 一小节中也有详细介绍，下面简要介绍一下字典序算法：

1. 从后往前寻找索引满足 `a[k] < a[k + 1]`，如果此条件不满足，则说明已遍历到最后一个。
2. 从后往前遍历，找到第一个比 `a[k]` 大的数 `a[l]`，即 `a[k] < a[l]`.
3. 交换 `a[k]` 与 `a[l]`.
4. 反转 `k + 1 ~ n` 之间的元素。

由于这道题中规定对于 `[4,3,2,1]`，输出为 `[1,2,3,4]`，故在第一步稍加处理即可。

Python

```python
class Solution:
    # @param num :  a list of integer
    # @return : a list of integer
    def nextPermutation(self, num):
        if num is None or len(num) <= 1:
            return num
        # step1: find nums[i] < nums[i + 1], Loop backwards
        i = 0
        for i in xrange(len(num) - 2, -1, -1):
            if num[i] < num[i + 1]:
                break
            elif i == 0:
                # reverse nums if reach maximum
                num = num[::-1]
                return num
        # step2: find nums[i] < nums[j], Loop backwards
```

```
        j = 0
        for j in xrange(len(num) - 1, i, -1):
            if num[i] < num[j]:
                break
        # step3: swap betwenn nums[i] and nums[j]
        num[i], num[j] = num[j], num[i]
        # step4: reverse between [i + 1, n - 1]
        num[i + 1:len(num)] = num[len(num) - 1:i:-1]

        return num
```

C++

```
class Solution {
public:
    /**
     * @param nums: An array of integers
     * @return: An array of integers that's next permuation
     */
    vector<int> nextPermutation(vector<int> &nums) {
        if (nums.empty() || nums.size() <= 1) {
            return nums;
        }
        // step1: find nums[i] < nums[i + 1]
        int i = 0;
        for (i = nums.size() - 2; i >= 0; --i) {
            if (nums[i] < nums[i + 1]) {
                break;
            } else if (0 == i) {
                // reverse nums if reach maximum
                reverse(nums, 0, nums.size() - 1);
                return nums;
            }
        }
        // step2: find nums[i] < nums[j]
        int j = 0;
        for (j = nums.size() - 1; j > i; --j) {
            if (nums[i] < nums[j]) break;
        }
        // step3: swap betwenn nums[i] and nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
        // step4: reverse between [i + 1, n - 1]
        reverse(nums, i + 1, nums.size() - 1);

        return nums;

    }

private:
    void reverse(vector<int>& nums, int start, int end) {
        for (int i = start, j = end; i < j; ++i, --j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
```

```
    }
};
```

Java

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: return nums in-place
     */
    public int[] nextPermutation(int[] nums) {
        if (nums == null || nums.length <= 1) {
            return nums;
        }
        // step1: find nums[i] < nums[i + 1]
        int i = 0;
        for (i = nums.length - 2; i >= 0; i--) {
            if (nums[i] < nums[i + 1]) {
                break;
            } else if (i == 0) {
                // reverse nums if reach maximum
                reverse(nums, 0, nums.length - 1);
                return nums;
            }
        }
        // step2: find nums[i] < nums[j]
        int j = 0;
        for (j = nums.length - 1; j > i; j--) {
            if (nums[i] < nums[j]) {
                break;
            }
        }
        // step3: swap betwenn nums[i] and nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
        // step4: reverse between [i + 1, n - 1]
        reverse(nums, i + 1, nums.length - 1);

        return nums;
    }

    private void reverse(int[] nums, int start, int end) {
        for (int i = start, j = end; i < j; i++, j--) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
}
```

源码分析

和 Permutation 一小节类似，这里只需要注意在step 1中 `i == 0` 时需要反转之以获得最小的序列。对于有

重复元素，只要在 step1和 step2中判断元素大小时不取等号即可。

复杂度分析

最坏情况下，遍历两次原数组，反转一次数组，时间复杂度为 $O(n)$, 使用了 temp 临时变量，空间复杂度可认为是 $O(1)$.

Reference

- Permutations

Previous Permuation

Source

- lintcode: (51) Previous Permuation

```
Given a list of integers, which denote a permutation.

Find the previous permutation in ascending order.

Example
For [1,3,2,3], the previous permutation is [1,2,3,3]

For [1,2,3,4], the previous permutation is [4,3,2,1]

Note
The list may contains duplicate integers.
```

题解

和前一题 Next Permutation 非常类似，这里找上一个排列，仍然使用字典序算法，大致步骤如下：

1. 从后往前寻找索引满足 `a[k] > a[k + 1]`，如果此条件不满足，则说明已遍历到最后一个。
2. 从后往前遍历，找到第一个比 `a[k]` 小的数 `a[l]`，即 `a[k] > a[l]`.
3. 交换 `a[k]` 与 `a[l]`.
4. 反转 `k + 1 ~ n` 之间的元素。

为何不从前往后呢？因为只有从后往前才能保证得到的是相邻的排列，可以举个实际例子自行分析。

Python

```python
class Solution:
    # @param num :  a list of integer
    # @return : a list of integer
    def previousPermuation(self, num):
        if num is None or len(num) <= 1:
            return num
        # step1: find nums[i] > nums[i + 1], Loop backwards
        i = 0
        for i in xrange(len(num) - 2, -1, -1):
            if num[i] > num[i + 1]:
                break
            elif i == 0:
                # reverse nums if reach maximum
                num = num[::-1]
                return num
        # step2: find nums[i] > nums[j], Loop backwards
        j = 0
        for j in xrange(len(num) - 1, i, -1):
```

```
            if num[i] > num[j]:
                break
        # step3: swap betwenn nums[i] and nums[j]
        num[i], num[j] = num[j], num[i]
        # step4: reverse between [i + 1, n - 1]
        num[i + 1:len(num)] = num[len(num) - 1:i:-1]

        return num
```

C++

```
class Solution {
public:
    /**
     * @param nums: An array of integers
     * @return: An array of integers that's previous permuation
     */
    vector<int> previousPermuation(vector<int> &nums) {
        if (nums.empty() || nums.size() <= 1) {
            return nums;
        }
        // step1: find nums[i] > nums[i + 1]
        int i = 0;
        for (i = nums.size() - 2; i >= 0; --i) {
            if (nums[i] > nums[i + 1]) {
                break;
            } else if (0 == i) {
                // reverse nums if reach minimum
                reverse(nums, 0, nums.size() - 1);
                return nums;
            }
        }
        // step2: find nums[i] > nums[j]
        int j = 0;
        for (j = nums.size() - 1; j > i; --j) {
            if (nums[i] > nums[j]) break;
        }
        // step3: swap betwenn nums[i] and nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
        // step4: reverse between [i + 1, n - 1]
        reverse(nums, i + 1, nums.size() - 1);

        return nums;
    }

private:
    void reverse(vector<int>& nums, int start, int end) {
        for (int i = start, j = end; i < j; ++i, --j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
};
```

Java

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers that's previous permuation
     */
    public ArrayList<Integer> previousPermuation(ArrayList<Integer> nums) {
        if (nums == null || nums.size() <= 1) {
            return nums;
        }
        // step1: find nums[i] > nums[i + 1]
        int i = 0;
        for (i = nums.size() - 2; i >= 0; i--) {
            if (nums.get(i) > nums.get(i + 1)) {
                break;
            } else if (i == 0) {
                // reverse nums if reach minimum
                reverse(nums, 0, nums.size() - 1);
                return nums;
            }
        }
        // step2: find nums[i] > nums[j]
        int j = 0;
        for (j = nums.size() - 1; j > i; j--) {
            if (nums.get(i) > nums.get(j)) {
                break;
            }
        }
        // step3: swap betwenn nums[i] and nums[j]
        Collections.swap(nums, i, j);
        // step4: reverse between [i + 1, n - 1]
        reverse(nums, i + 1, nums.size() - 1);

        return nums;
    }

    private void reverse(List<Integer> nums, int start, int end) {
        for (int i = start, j = end; i < j; i++, j--) {
            Collections.swap(nums, i, j);
        }
    }
}
```

源码分析

和 Permutation 一小节类似，这里只需要注意在step 1中 `i == 0` 时需要反转之以获得最大的序列。对于有重复元素，只要在 step1和 step2中判断元素大小时不取等号即可。

复杂度分析

最坏情况下，遍历两次原数组，反转一次数组，时间复杂度为 $O(n)$, 使用了 temp 临时变量，空间复杂度可认为是 $O(1)$.

Reference

- Permutations

Unique Binary Search Trees II

Source

- leetcode: Unique Binary Search Trees II | LeetCode OJ
- lintcode: (164) Unique Binary Search Trees II

```
Given n, generate all structurally unique BST's
(binary search trees) that store values 1...n.

Example
Given n = 3, your program should return all 5 unique BST's shown below.

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

题解

题 Unique Binary Search Trees 的升级版，这道题要求的不是二叉搜索树的数目，而是要构建这样的树。分析方法仍然是可以借鉴的，核心思想为利用『二叉搜索树』的定义，如果以 i 为根节点，那么其左子树由[1, i - 1]构成，右子树由[i + 1, n] 构成。要构建包含1到n的二叉搜索树，只需遍历1到n中的数作为根节点，以 `i` 为界将数列分为左右两部分，小于 `i` 的数作为左子树，大于 `i` 的数作为右子树，使用两重循环将左右子树所有可能的组合链接到以 `i` 为根节点的节点上。

容易看出，以上求解的思路非常适合用递归来处理，接下来便是设计递归的终止步、输入参数和返回结果了。由以上分析可以看出递归严重依赖数的区间和 `i` ，那要不要将 `i` 也作为输入参数的一部分呢？首先可以肯定的是必须使用『数的区间』这两个输入参数，又因为 `i` 是随着『数的区间』这两个参数的，故不应该将其加入到输入参数中。分析方便，不妨设『数的区间』两个输入参数分别为 `start` 和 `end` .

接下来谈谈终止步的确定，由于根据 `i` 拆分左右子树的过程中，递归调用的方法中入口参数会缩小，且存在 `start <= i <= end` , 故终止步为 `start > end` . 那要不要对 `start == end` 返回呢？保险起见可以先写上，后面根据情况再做删改。总结以上思路，简单的伪代码如下：

```
helper(start, end) {
    result;
    if (start > end) {
        result.push_back(NULL);
        return;
    } else if (start == end) {
        result.push_back(TreeNode(i));
        return;
    }

    // dfs
```

```
    for (int i = start; i <= end; ++i) {
        leftTree = helper(start, i - 1);
        rightTree = helper(i + 1, end);
        // link left and right sub tree to the root i
        for (j in leftTree ){
            for (k in rightTree) {
                root = TreeNode(i);
                root->left = leftTree[j];
                root->right = rightTree[k];
                result.push_back(root);
            }
        }
    }

    return result;
}
```

大致的框架如上所示，我们来个简单的数据验证下，以[1, 2, 3]为例，调用堆栈图如下所示：

1. helper(1,3)
    - [leftTree]: helper(1, 0) ==> return NULL
    - ---loop i = 2---
    - [rightTree]: helper(2, 3)
        i. [leftTree]: helper(2,1) ==> return NULL
        ii. [rightTree]: helper(3,3) ==> return node(3)
        iii. [for loop]: ==> return (2->3)
    - ---loop i = 3---
        i. [leftTree]: helper(2,2) ==> return node(2)
        ii. [rightTree]: helper(4,3) ==> return NULL
        iii. [for loop]: ==> return (3->2)
2. ...

简单验证后可以发现这种方法的**核心为递归地构造左右子树并将其链接到相应的根节点中**。对于 `start` 和 `end` 相等的情况的，其实不必单独考虑，因为 `start == end` 时其左右子树均返回空，故在 `for` 循环中返回根节点。当然单独考虑可减少递归栈的层数，但实际测下来后发现运行时间反而变长了不少 :(

Python

```
"""
Definition of TreeNode:
class TreeNode:
    def __init__(self, val):
        this.val = val
        this.left, this.right = None, None
"""
class Solution:
    # @paramn n: An integer
    # @return: A list of root
    def generateTrees(self, n):
        return self.helper(1, n)
```

```
    def helper(self, start, end):
        result = []
        if start > end:
            result.append(None)
            return result

        for i in xrange(start, end + 1):
            # generate left and right sub tree
            leftTree = self.helper(start, i - 1)
            rightTree = self.helper(i + 1, end)
            # link left and right sub tree to root(i)
            for j in xrange(len(leftTree)):
                for k in xrange(len(rightTree)):
                    root = TreeNode(i)
                    root.left = leftTree[j]
                    root.right = rightTree[k]
                    result.append(root)

        return result
```

C++

```
/**
 * 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:
    /**
     * @paramn n: An integer
     * @return: A list of root
     */
    vector<TreeNode *> generateTrees(int n) {
        return helper(1, n);
    }

private:
    vector<TreeNode *> helper(int start, int end) {
        vector<TreeNode *> result;
        if (start > end) {
            result.push_back(NULL);
            return result;
        }

        for (int i = start; i <= end; ++i) {
            // generate left and right sub tree
            vector<TreeNode *> leftTree = helper(start, i - 1);
            vector<TreeNode *> rightTree = helper(i + 1, end);
            // link left and right sub tree to root(i)
```

```
            for (int j = 0; j < leftTree.size(); ++j) {
                for (int k = 0; k < rightTree.size(); ++k) {
                    TreeNode *root = new TreeNode(i);
                    root->left = leftTree[j];
                    root->right = rightTree[k];
                    result.push_back(root);
                }
            }
        }

        return result;
    }
};
```

Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @paramn n: An integer
     * @return: A list of root
     */
    public List<TreeNode> generateTrees(int n) {
        return helper(1, n);
    }

    private List<TreeNode> helper(int start, int end) {
        List<TreeNode> result = new ArrayList<TreeNode>();
        if (start > end) {
            result.add(null);
            return result;
        }

        for (int i = start; i <= end; i++) {
            // generate left and right sub tree
            List<TreeNode> leftTree = helper(start, i - 1);
            List<TreeNode> rightTree = helper(i + 1, end);
            // link left and right sub tree to root(i)
            for (TreeNode lnode: leftTree) {
                for (TreeNode rnode: rightTree) {
                    TreeNode root = new TreeNode(i);
                    root.left = lnode;
                    root.right = rnode;
                    result.add(root);
                }
            }
        }
```

```
        return result;
    }
}
```

源码分析

1. 异常处理，返回None/NULL/null.
2. 遍历start->end, 递归得到左子树和右子树。
3. 两重 `for` 循环将左右子树的所有可能组合添加至最终返回结果。

注意 DFS 辅助方法 `helper` 中左右子树及返回根节点的顺序。

复杂度分析

递归调用，一个合理的数组区间将生成新的左右子树，时间复杂度为指数级别，使用的临时空间最后都被加入到最终结果，空间复杂度(堆)近似为 $O(1)$, 栈上的空间较大。

Reference

- Code Ganker: Unique Binary Search Trees II -- LeetCode
- 水中的鱼: [LeetCode] Unique Binary Search Trees II, Solution
- Accepted Iterative Java solution. - Leetcode Discuss
- Unique Binary Search Trees II 参考程序 Java/C++/Python

Permutation Index

Source

- lintcode: (197) Permutation Index

```
Given a permutation which contains no repeated number,
find its index in all the permutations of these numbers,
which are ordered in lexicographical order. The index begins at 1.

Example
Given [1,2,4], return 1.
```

题解

做过 next permutation 系列题的话自然能想到不断迭代直至最后一个，最后返回计数器的值即可。这种方法理论上自然是可行的，但是最坏情况下时间复杂度为 $O(n!)$, 显然是不能接受的。由于这道题只是列出某给定 permutation 的相对顺序(index), 故我们可从 permutation 的特点出发进行分析。

以序列 `1, 2, 4` 为例，其不同的排列共有 `3!=6` 种，以排列 `[2, 4, 1]` 为例，若将1置于排列的第一位，后面的排列则有 `2!=2` 种。将2置于排列的第一位，由于 `[2, 4, 1]` 的第二位4在1, 2, 4中为第3大数，故第二位可置1或者2，那么相应的排列共有 `2 * 1! = 2` 种，最后一位1为最小的数，故比其小的排列为0。综上，可参考我们常用的十进制和二进制的转换，对于 `[2, 4, 1]` , 可总结出其排列的 `index` 为 `2! * (2 - 1) + 1! * (3 - 1) + 0! * (1 - 1) + 1` .

以上分析看似正确无误，实则有个关键的漏洞，在排定第一个数2后，第二位数只可为1或者4，而无法为2, 故在计算最终的 index 时需要动态计算某个数的相对大小。按照从低位到高位进行计算，我们可通过两重循环得出到某个索引处值的相对大小。

Python

```python
class Solution:
    # @param {int[]} A an integer array
    # @return {long} a long integer
    def permutationIndex(self, A):
        if A is None or len(A) == 0:
            return 0

        index = 1
        factor = 1
        for i in xrange(len(A) - 1, -1, -1):
            rank = 0
            for j in xrange(i + 1, len(A)):
                if A[i] > A[j]:
                    rank += 1

            index += rank * factor
```

```
            factor *= (len(A) - i)

        return index
```

C++

```
class Solution {
public:
    /**
     * @param A an integer array
     * @return a long integer
     */
    long long permutationIndex(vector<int>& A) {
        if (A.empty()) return 0;

        long long index = 1;
        long long factor = 1;
        for (int i = A.size() - 1; i >= 0; --i) {
            int rank = 0;
            for (int j = i + 1; j < A.size(); ++j) {
                if (A[i] > A[j]) ++rank;
            }
            index += rank * factor;
            factor *= (A.size() - i);
        }

        return index;
    }
};
```

Java

```
public class Solution {
    /**
     * @param A an integer array
     * @return a long integer
     */
    public long permutationIndex(int[] A) {
        if (A == null || A.length == 0) return 0;

        long index = 1;
        long factor = 1;
        for (int i = A.length - 1; i >= 0; i--) {
            int rank = 0;
            for (int j = i + 1; j < A.length; j++) {
                if (A[i] > A[j]) rank++;
            }
            index += rank * factor;
            factor *= (A.length - i);
        }

        return index;
    }
}
```

源码分析

注意 index 和 factor 的初始化值，rank 的值每次计算时都需要重新置零，index 先自增，factor 后自乘求阶乘。

复杂度分析

双重 for 循环，时间复杂度为 $O(n^2)$. 使用了部分额外空间，空间复杂度 $O(1)$.

Reference

- Permutation Index

Permutation Index II

Source

- lintcode: (198) Permutation Index II

```
Given a permutation which may contain repeated numbers,
find its index in all the permutations of these numbers,
which are ordered in lexicographical order. The index begins at 1.

Example
Given the permutation [1, 4, 2, 2], return 3.
```

题解

题 Permutation Index 的扩展，这里需要考虑重复元素，有无重复元素最大的区别在于原来的 `1!, 2!, 3!...` 等需要除以重复元素个数的阶乘，颇有点高中排列组合题的味道。记录重复元素个数同样需要动态更新，引入哈希表这个万能的工具较为方便。

Python

```python
class Solution:
    # @param {int[]} A an integer array
    # @return {long} a long integer
    def permutationIndexII(self, A):
        if A is None or len(A) == 0:
            return 0

        index = 1
        factor = 1
        for i in xrange(len(A) - 1, -1, -1):
            hash_map = {A[i]: 1}
            rank = 0
            for j in xrange(i + 1, len(A)):
                if A[j] in hash_map.keys():
                    hash_map[A[j]] += 1
                else:
                    hash_map[A[j]] = 1
                # get rank
                if A[i] > A[j]:
                    rank += 1

            index += rank * factor / self.dupPerm(hash_map)
            factor *= (len(A) - i)

        return index


    def dupPerm(self, hash_map):
```

```
        if hash_map is None or len(hash_map) == 0:
            return 0
        dup = 1
        for val in hash_map.values():
            dup *= self.factorial(val)

        return dup


    def factorial(self, n):
        r = 1
        for i in xrange(1, n + 1):
            r *= i

        return r
```

C++

```
class Solution {
public:
    /**
     * @param A an integer array
     * @return a long integer
     */
    long long permutationIndexII(vector<int>& A) {
        if (A.empty()) return 0;

        long long index = 1;
        long long factor = 1;
        for (int i = A.size() - 1; i >= 0; --i) {
            int rank = 0;
            unordered_map<int, int> hash;
            ++hash[A[i]];
            for (int j = i + 1; j < A.size(); ++j) {
                ++hash[A[j]];

                if (A[i] > A[j]) {
                    ++rank;
                }
            }
            index += rank * factor / dupPerm(hash);
            factor *= (A.size() - i);
        }

        return index;
    }

private:
    long long dupPerm(unordered_map<int, int> hash) {
        if (hash.empty()) return 1;

        long long dup = 1;
        for (auto it = hash.begin(); it != hash.end(); ++it) {
            dup *= fact(it->second);
        }

        return dup;
```

```
    }

    long long fact(int num) {
        long long val = 1;
        for (int i = 1; i <= num; ++i) {
            val *= i;
        }

        return val;
    }
};
```

Java

```
public class Solution {
    /**
     * @param A an integer array
     * @return a long integer
     */
    public long permutationIndexII(int[] A) {
        if (A == null || A.length == 0) return 0;

        long index = 1;
        long factor = 1;
        for (int i = A.length - 1; i >= 0; i--) {
            HashMap<Integer, Integer> hash = new HashMap<Integer, Integer>();
            hash.put(A[i], 1);
            int rank = 0;
            for (int j = i + 1; j < A.length; j++) {
                if (hash.containsKey(A[j])) {
                    hash.put(A[j], hash.get(A[j]) + 1);
                } else {
                    hash.put(A[j], 1);
                }

                if (A[i] > A[j]) {
                    rank++;
                }
            }
            index += rank * factor / dupPerm(hash);
            factor *= (A.length - i);
        }

        return index;
    }

    private long dupPerm(HashMap<Integer, Integer> hash) {
        if (hash == null || hash.isEmpty()) return 1;

        long dup = 1;
        for (int val : hash.values()) {
            dup *= fact(val);
        }

        return dup;
    }
```

```
    private long fact(int num) {
        long val = 1;
        for (int i = 1; i <= num; i++) {
            val *= i;
        }

        return val;
    }
}
```

源码分析

在计算重复元素个数的阶乘时需要注意 `dup *= fact(val);` , 而不是 `dup *= val;` . 对元素 `A[i]` 需要加入哈希表 - `hash.put(A[i], 1);` ，设想一下 `2, 2, 1, 1` 的计算即可知。

复杂度分析

双重 for 循环，时间复杂度为 $O(n^2)$, 使用了哈希表，空间复杂度为 $O(n)$.

Permutation Sequence

Source

- leetcode: Permutation Sequence | LeetCode OJ
- lintcode: (388) Permutation Sequence

```
Given n and k, return the k-th permutation sequence.

Example
For n = 3, all permutations are listed as follows:

"123"
"132"
"213"
"231"
"312"
"321"
If k = 4, the fourth permutation is "231"

Note
n will be between 1 and 9 inclusive.

Challenge
O(n*k) in time complexity is easy, can you do it in O(n^2) or less?
```

题解

和题 Permutation Index 正好相反，这里给定第几个排列的相对排名，输出排列值。和不同进制之间的转化类似，这里的『进制』为 `1!, 2!...`，以n=3, k=4为例，我们从高位到低位转化，直觉应该是用 `k/(n-1)!`，但以 n=3,k=5 和 n=3,k=6 代入计算后发现边界处理起来不太方便，故我们可以尝试将 k 减1进行运算，后面的基准也随之变化。第一个数可以通过 `(k-1)/(n-1)!` 进行计算，那么第二个数呢？联想不同进制数之间的转化，我们可以通过求模运算求得下一个数的 `k-1`，那么下一个数可通过 `(k2 - 1)/(n-2)!` 求得，这里不理解的可以通过进制转换类比进行理解。和减掉相应的阶乘值是等价的。

Python

```python
class Solution:
    """
    @param n: n
    @param k: the k-th permutation
    @return: a string, the k-th permutation
    """
    def getPermutation(self, n, k):
        # generate factorial list
        factorial = [1]
        for i in xrange(1, n + 1):
            factorial.append(factorial[-1] * i)
```

```
        nums = range(1, n + 1)
        perm = []
        for i in xrange(n):
            rank = (k - 1) / factorial[n - i - 1]
            k = (k - 1) % factorial[n - i - 1] + 1
            # append and remove nums[rank]
            perm.append(nums[rank])
            nums.remove(nums[rank])
        # combine digits
        return "".join([str(digit) for digit in perm])
```

C++

```
class Solution {
public:
    /**
      * @param n: n
      * @param k: the kth permutation
      * @return: return the k-th permutation
      */
    string getPermutation(int n, int k) {
        // generate factorial list
        vector<int> factorial = vector<int>(n + 1, 1);
        for (int i = 1; i < n + 1; ++i) {
            factorial[i] = factorial[i - 1] * i;
        }
        // generate digits ranging from 1 to n
        vector<int> nums;
        for (int i = 1; i < n + 1; ++i) {
            nums.push_back(i);
        }

        vector<int> perm;
        for (int i = 0; i < n; ++i) {
            int rank = (k - 1) / factorial[n - i - 1];
            k = (k - 1) % factorial[n - i - 1] + 1;
            // append and remove nums[rank]
            perm.push_back(nums[rank]);
            nums.erase(std::remove(nums.begin(), nums.end(), nums[rank]), nums.end());
        }
        // transform a vector<int> to a string
        std::stringstream result;
        std::copy(perm.begin(), perm.end(), std::ostream_iterator<int>(result, ""));

        return result.str();
    }
};
```

Java

```
class Solution {
    /**
      * @param n: n
```

```
     * @param k: the kth permutation
     * @return: return the k-th permutation
     */
    public String getPermutation(int n, int k) {
        // generate factorial list
        int[] factorial = new int[n + 1];
        factorial[0] = 1;
        for (int i = 1; i < n + 1; i++) {
            factorial[i] = factorial[i - 1] * i;
        }
        // generate digits ranging from 1 to n
        ArrayList<Integer> nums = new ArrayList<Integer>(n);
        for (int i = 0; i < n; i++) {
            nums.add(i + 1);
        }

        int[] perm = new int[n];
        for (int i = 0; i < n; i++) {
            int rank = (k - 1) / factorial[n - i - 1];
            k = (k - 1) % factorial[n - i - 1] + 1;

            perm[i] = nums.get(rank);
            nums.remove(nums.get(rank));
        }

        StringBuilder result = new StringBuilder();
        for (int digit : perm) {
          result.append(digit);
        }
        return result.toString();
    }
}
```

源码分析

源码结构分为三步走，

1. 建阶乘数组
2. 生成排列数字数组
3. 从高位到低位计算排列数值

复杂度分析

几个 for 循环，时间复杂度为 $O(n)$, 用了与 n 等长的一些数组，空间复杂度为 $O(n)$.

Reference

- Permutation Sequence 解题报告
- Permutation Sequence 参考程序 Java/C++/Python
- c++ - How to transform a vector into a string? - Stack Overflow

Palindrome Partitioning

- tags: [palindrome]

Source

- leetcode: Palindrome Partitioning | LeetCode OJ
- lintcode: (136) Palindrome Partitioning

```
Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab",
Return

  [
    ["aa","b"],
    ["a","a","b"]
  ]
```

题解1 - DFS

罗列所有可能，典型的 DFS. 此题要求所有可能的回文子串，即需要找出所有可能的分割，使得分割后的子串都为回文。凭借高中的排列组合知识可知这可以用『隔板法』来解决，具体就是在字符串的每个间隙为一个隔板，对于长度为 n 的字符串，共有 n-1 个隔板可用，每个隔板位置可以选择放或者不放，总共有 $O(2^{n-1})$ 种可能。由于需要满足『回文』条件，故实际上需要穷举的状态数小于 $O(2^{n-1})$.

回溯法看似不难，但是要活学活用起来还是不容易的，核心抓住两点：**深搜的递归建立和剪枝函数的处理。**

根据『隔板法』的思想，我们首先从第一个隔板开始挨个往后取，若取到的子串不是回文则立即取下一个隔板，直到取到最后一个隔板。若取到的子串是回文，则将当前子串加入临时列表中，接着从当前隔板处字符开始递归调用回溯函数，直至取到最后一个隔板，最后将临时列表中的子串加入到最终返回结果中。接下来则将临时列表中的结果一一移除，这个过程和 subsets 模板很像，代码比这个文字描述更为清晰。

Python

```python
class Solution:
    # @param s, a string
    # @return a list of lists of string
    def partition(self, s):
        result = []
        if not s:
            return result
```

```
        palindromes = []
        self.dfs(s, 0, palindromes, result)
        return result

    def dfs(self, s, pos, palindromes, ret):
        if pos == len(s):
            ret.append([] + palindromes)
            return

        for i in xrange(pos + 1, len(s) + 1):
            if not self.isPalindrome(s[pos:i]):
                continue

            palindromes.append(s[pos:i])
            self.dfs(s, i, palindromes, ret)
            palindromes.pop()

    def isPalindrome(self, s):
        if not s:
            return False
        # reverse compare
        return s == s[::-1]
```

C++

```
class Solution {
public:
    /**
     * @param s: A string
     * @return: A list of lists of string
     */
    vector<vector<string>> partition(string s) {
        vector<vector<string> > result;
        if (s.empty()) return result;

        vector<string> palindromes;
        dfs(s, 0, palindromes, result);

        return result;
    }

private:
    void dfs(string s, int pos, vector<string> &palindromes,
             vector<vector<string> > &ret) {

        if (pos == s.size()) {
            ret.push_back(palindromes);
            return;
        }

        for (int i = pos + 1; i <= s.size(); ++i) {
            string substr = s.substr(pos, i - pos);
            if (!isPalindrome(substr)) {
                continue;
            }

            palindromes.push_back(substr);
```

```
            dfs(s, i, palindromes, ret);
            palindromes.pop_back();
        }
    }

    bool isPalindrome(string s) {
        if (s.empty()) return false;

        int n = s.size();
        for (int i = 0; i < n; ++i) {
            if (s[i] != s[n - i - 1]) return false;
        }

        return true;
    }
};
```

Java

```
public class Solution {
    /**
     * @param s: A string
     * @return: A list of lists of string
     */
    public List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<List<String>>();
        if (s == null || s.isEmpty()) return result;

        List<String> palindromes = new ArrayList<String>();
        dfs(s, 0, palindromes, result);

        return result;
    }

    private void dfs(String s, int pos, List<String> palindromes,
                     List<List<String>> ret) {

        if (pos == s.length()) {
            ret.add(new ArrayList<String>(palindromes));
            return;
        }

        for (int i = pos + 1; i <= s.length(); i++) {
            String substr = s.substring(pos, i);
            if (!isPalindrome(substr)) {
                continue;
            }

            palindromes.add(substr);
            dfs(s, i, palindromes, ret);
            palindromes.remove(palindromes.size() - 1);
        }
    }

    private boolean isPalindrome(String s) {
        if (s == null || s.isEmpty()) return false;
```

```
        int n = s.length();
        for (int i = 0; i < n; i++) {
            if (s.charAt(i) != s.charAt(n - i - 1)) return false;
        }

        return true;
    }
}
```

源码分析

回文的判断采用了简化的版本，没有考虑空格等非字母数字字符要求。Java 中 ArrayList 和 List 的实例化需要注意下。Python 中 result 的初始化为[], 不需要初始化为 [[]] 画蛇添足。C++ 中的 `.substr(pos, n)` 含义为从索引为 pos 的位置往后取 n 个(含) 字符，注意与 Java 中区别开来。

复杂度分析

DFS，状态数最多 $O(2^{n-1})$, 故时间复杂度为 $O(2^n)$, 使用了临时列表，空间复杂度为 $O(n)$.

Reference

- Palindrome Partitioning 参考程序 Java/C++/Python
- soulmachine 的 Palindrome Partitioning

Combinations

Source

- leetcode: Combinations | LeetCode OJ
- lintcode: (152) Combinations

```
Given two integers n and k,
return all possible combinations of k numbers out of 1 ... n.

Example
For example,
If n = 4 and k = 2, a solution is:
[[2,4],[3,4],[2,3],[1,2],[1,3],[1,4]]
```

题解

套用 Permutations 模板。

Java

```java
public class Solution {
    /**
     * @param n: Given the range of numbers
     * @param k: Given the numbers of combinations
     * @return: All the combinations of k numbers out of 1..n
     */
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
        if (n <= 0 || k <= 0) return result;

        helper(n, k, 1, list, result);
        return result;
    }

    private void helper(int n, int k, int pos,
                        List<Integer> list, List<List<Integer>> result) {

        if (list.size() == k) {
            result.add(new ArrayList<Integer>(list));
            return;
        }

        for (int i = pos; i <= n; i++) {
            list.add(i);
            helper(n, k, i + 1, list, result);
            list.remove(list.size() - 1);
        }
    }
```

```
}
```

源码分析

注意递归 `helper(n, k, i + 1, list, result);` 中的 `i + 1` ，不是 `pos + 1` 。

复杂度分析

状态数 C_n^2, 每组解有两个元素，故时间复杂度应为 $O(n^2)$. list 只保留最多两个元素，空间复杂度 $O(1)$.

Combination Sum

Source

- leetcode: Combination Sum | LeetCode OJ
- lintcode: (135) Combination Sum

```
Given a set of candidate numbers (C) and a target number (T),
find all unique combinations in C where the candidate numbers sums to T.
The same repeated number may be chosen from C unlimited number of times.

For example, given candidate set 2,3,6,7 and target 7,
A solution set is:
[7]
[2, 2, 3]

Have you met this question in a real interview? Yes
Example
given candidate set 2,3,6,7 and target 7,
A solution set is:
[7]
[2, 2, 3]

Note
- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, … , ak) must be in non-descending order.
(ie, a1 ≤ a2 ≤ … ≤ ak).
- The solution set must not contain duplicate combinations.
```

题解

和 Permutations 十分类似，区别在于剪枝函数不同。这里允许一个元素被多次使用，故递归时传入的索引值不自增，而是由 for 循环改变。

Java

```java
public class Solution {
    /**
     * @param candidates: A list of integers
     * @param target:An integer
     * @return: A list of lists of integers
     */
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
        if (candidates == null) return result;

        Arrays.sort(candidates);
        helper(candidates, 0, target, list, result);
```

```
        return result;
    }

    private void helper(int[] candidates, int pos, int gap,
                        List<Integer> list, List<List<Integer>> result) {

        if (gap == 0) {
            // add new object for result
            result.add(new ArrayList<Integer>(list));
            return;
        }

        for (int i = pos; i < candidates.length; i++) {
            // cut invalid candidate
            if (gap < candidates[i]) {
                return;
            }
            list.add(candidates[i]);
            helper(candidates, i, gap - candidates[i], list, result);
            list.remove(list.size() - 1);
        }
    }
}
```

源码分析

对数组首先进行排序是必须的，递归函数中本应该传入 target 作为入口参数，这里借用了 Soulmachine 的实现，使用 gap 更容易理解。注意在将临时 list 添加至 result 中时需要 new 一个新的对象。

复杂度分析

按状态数进行分析，时间复杂度 $O(n!)$, 使用了list 保存中间结果，空间复杂度 $O(n)$.

Reference

- Soulmachine 的 leetcode 题解

Combination Sum II

Source

- leetcode: Combination Sum II | LeetCode OJ
- lintcode: (153) Combination Sum II

```
Given a collection of candidate numbers (C) and a target number (T),
find all unique combinations in C where the candidate numbers sums to T.
Each number in C may only be used once in the combination.

Have you met this question in a real interview? Yes
Example
For example, given candidate set 10,1,6,7,2,1,5 and target 8,

A solution set is:

[1,7]

[1,2,5]

[2,6]

[1,1,6]

Note
All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, … , ak) must be in non-descending order.
(ie, a1 ≤ a2 ≤ … ≤ ak).
The solution set must not contain duplicate combinations.
```

题解

和 Unique Subsets 非常类似。在 Combination Sum 的基础上改改就好了。

Java

```java
public class Solution {
    /**
     * @param num: Given the candidate numbers
     * @param target: Given the target number
     * @return: All the combinations that sum to target
     */
    public List<List<Integer>> combinationSum2(int[] num, int target) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
        if (num == null) return result;

        Arrays.sort(num);
        helper(num, 0, target, list, result);
```

```
        return result;
    }

    private void helper(int[] nums, int pos, int gap,
                        List<Integer> list, List<List<Integer>> result) {

        if (gap == 0) {
            result.add(new ArrayList<Integer>(list));
            return;
        }

        for (int i = pos; i < nums.length; i++) {
            // ensure only the first same num is chosen, remove duplicate list
            if (i != pos && nums[i] == nums[i - 1]) {
                continue;
            }
            // cut invalid num
            if (gap < nums[i]) {
                return;
            }
            list.add(nums[i]);
            // i + 1 ==> only be used once
            helper(nums, i + 1, gap - nums[i], list, result);
            list.remove(list.size() - 1);
        }
    }
}
```

源码分析

这里去重的方法继承了 Unique Subsets 中的做法，当然也可以新建一变量 `prev`，由于这里每个数最多只能使用一次，故递归时索引变量传 `i + 1`.

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(n)$.

Minimum Depth of Binary Tree

Source

- leetcode: Minimum Depth of Binary Tree | LeetCode OJ
- lintcode: (155) Minimum Depth of Binary Tree

```
Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path
from the root node down to the nearest leaf node.

Example
Given a binary tree as follow:

        1

     /     \

   2       3

          /    \

        4      5
The minimum depth is 2
```

题解

注意审题，题中的最小深度指的是从根节点到**最近的叶子节点（因为题中的最小深度是the number of nodes，故该叶子节点不能是空节点）**，所以需要单独处理叶子节点为空的情况。此题使用 DFS 递归实现比较简单。

Java

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
```

```
     */
    public int minDepth(TreeNode root) {
        if (root == null) return 0;

        int leftDepth = minDepth(root.left);
        int rightDepth = minDepth(root.right);

        // current node is not leaf node
        if (root.left == null) {
            return 1 + rightDepth;
        } else if (root.right == null) {
            return 1 + leftDepth;
        }

        return 1 + Math.min(leftDepth, rightDepth);
    }
}
```

源码分析

建立好递归模型即可，左右子节点为空时需要单独处理下。

复杂度分析

每个节点遍历一次，时间复杂度 $O(n)$. 不计栈空间的话空间复杂度 $O(1)$.

Dynamic Programming - 动态规划

动态规划是一种「分治」的思想，通俗一点来说就是「大事化小，小事化无」的艺术。在将大问题化解为小问题的「分治」过程中，保存对这些小问题已经处理好的结果，并供后面处理更大规模的问题时直接使用这些结果。嗯，感觉讲了和没讲一样，还是不会使用动规的思想解题...

下面看看知乎上的熊大大对动规比较「正经」的描述。

> 动态规划是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。

以上定义言简意赅，可直接用于实战指导，不愧是参加过NOI的。

动规的思想虽然好理解，但是要真正活用起来就需要下点功夫了。建议看看下面知乎上的回答。

动态规划最重要的两个要点：

1. 状态(状态不太好找，可先从转化方程入手分析)
2. 状态间的转化方程(从题目的隐含条件出发寻找递推关系)

其他的要点则是如初始化状态的确定(由状态和转化方程得知)，需要的结果(状态转移的终点)

动态规划问题中一般从以下四个角度考虑：

1. 状态(State)
2. 状态间的转移方程(Function)
3. 状态的初始化(Initialization)
4. 返回结果(Answer)

动规适用的情形：

1. 最大值/最小值
2. 有无可行解
3. 求方案个数(如果需要列出所有方案，则一定不是动规，因为全部方案为指数级别复杂度，所有方案需要列出时往往用递归)
4. 给出的数据不可随便调整位置

单序列(DP_Sequence)

单序列动态规划的状态通常定义为：数组前 i 个位置, 数字, 字母 或者 以第i个为... 返回结果通常为数组的最后一个元素。

按照动态规划的四要素，此类题可从以下四个角度分析。

1. State: f[i] 前i个位置/数字/字母...
2. Function: f[i] = f[i-1]... 找递推关系
3. Initialization: 根据题意进行必要的初始化

4. Answer: f[n-1]

双序列(DP_Two_Sequence)

一般有两个数组或者两个字符串，计算其匹配关系。双序列中常用二维数组表示状态转移关系，但往往可以使用滚动数组的方式对空间复杂度进行优化。举个例子，以题 Distinct Subsequences 为例，状态转移方程如下：

```
f[i+1][j+1] = f[i][j+1] + f[i][j] (if S[i] == T[j])
f[i+1][j+1] = f[i][j+1] (if S[i] != T[j])
```

从以上转移方程可以看出 `f[i+1][*]` 只与其前一个状态 `f[i][*]` 有关，而对于 `f[*][j]` 来说则基于当前索引又与前一个索引有关，故我们以递推的方式省略第一维的空间，并以第一维作为外循环，内循环为 j, 由递推关系可知在使用滚动数组时，若内循环 j 仍然从小到大遍历，那么对于 `f[j+1]` 来说它得到的 `f[j]` 则是当前一轮(`f[i+1][j]`)的值，并不是需要的 `f[i][j]` 的值。所以若想得到上一轮的结果，必须在内循环使用逆推的方式进行。文字表述比较模糊，可以自行画一个二维矩阵的转移矩阵来分析，认识到这一点非常重要。

小结一下，使用滚动数组的核心在于：

1. 状态转移矩阵中只能取 `f[i+1][*]` 和 `f[i][*]` , 这是使用滚动数组的前提。
2. 外循环使用 i, 内循环使用 j 并同时使用逆推，这是滚动数组使用的具体实践。

代码如下：

```java
public class Solution {
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        if (S == null || T == null) return 0;
        if (S.length() < T.length()) return 0;
        if (T.length() == 0) return 1;

        int[] f = new int[T.length() + 1];
        f[0] = 1;
        for (int i = 0; i < S.length(); i++) {
            for (int j = T.length() - 1; j >= 0; j--) {
                if (S.charAt(i) == T.charAt(j)) {
                        f[j + 1] += f[j];
                }
            }
        }

        return f[T.length()];
    }
}
```

纸上得来终觉浅，绝知此事要躬行。光说不练假把戏，下面就来几道DP的题练练手。

Reference

1. 什么是动态规划？动态规划的意义是什么？ - 知乎 - 熊大大和王勐的回答值得细看，适合作为动态规划的科普和入门。维基百科上对动态规划的描述感觉太过学术。
2. 动态规划：从新手到专家 - Topcoder上的一篇译作。

Triangle - Find the minimum path sum from top to bottom

Source

- lintcode: (109) Triangle

```
Given a triangle, find the minimum path sum from top to bottom. Each step you may move to

Note
Bonus point if you are able to do this using only O(n) extra space, where n is the total

Example
For example, given the following triangle

[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).
```

题解

题中要求最短路径和，每次只能访问下行的相邻元素，将triangle视为二维坐标。此题方法较多，下面分小节详述。

Method 1 - Traverse without hashmap

首先考虑最容易想到的方法——递归遍历，逐个累加所有自上而下的路径长度，最后返回这些不同的路径长度的最小值。由于每个点往下都有2条路径，使用此方法的时间复杂度约为 $O(2^n)$, 显然是不可接受的解，不过我们还是先看看其实现思路。

C++ Traverse without hashmap

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int> > &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        int result = INT_MAX;
```

```
        dfs(0, 0, 0, triangle, result);

        return result;
    }

private:
    void dfs(int x, int y, int sum, vector<vector<int> > &triangle, int &result) {
        const int n = triangle.size();
        if (x == n) {
            if (sum < result) {
                result = sum;
            }
            return;
        }

        dfs(x + 1, y, (sum + triangle[x][y]), triangle, result);
        dfs(x + 1, y + 1, (sum + triangle[x][y]), triangle, result);
    }
};
```

源码分析

`dfs()` 的循环终止条件为 `x == n` ，而不是 `x == n - 1` ，主要是方便在递归时sum均可使用 `sum + triangle[x][y]` ，而不必根据不同的y和y+1改变，代码实现相对优雅一些。理解方式则变为从第x行走到第x+1行时的最短路径和，也就是说在此之前并不将第x行的元素值计算在内。

这种遍历的方法时间复杂度如此之高的主要原因是因为在n较大时递归计算了之前已经得到的结果，而这些结果计算一次后即不再变化，可再次利用。因此我们可以使用hashmap记忆已经计算得到的结果从而对其进行优化。

Method 2 - Divide and Conquer without hashmap

既然可以使用递归遍历，当然也可以使用「分治」的方法来解。「分治」与之前的遍历区别在于「分治」需要返回每次「分治」后的计算结果，下面看代码实现。

C++ Divide and Conquer without hashmap

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int> > &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        int result = dfs(0, 0, triangle);

        return result;
    }
```

```
private:
    int dfs(int x, int y, vector<vector<int> > &triangle) {
        const int n = triangle.size();
        if (x == n) {
            return 0;
        }

        return min(dfs(x + 1, y, triangle), dfs(x + 1, y + 1, triangle))  + triangle[x][y
    }
};
```

使用「分治」的方法代码相对简洁一点，接下来我们使用hashmap保存triangle中不同坐标的点计算得到的路径和。

Method 3 - Divide and Conquer with hashmap

新建一份大小和triangle一样大小的hashmap，并对每个元素赋以 `INT_MIN` 以做标记区分。

C++ Divide and Conquer with hashmap

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int> > &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        vector<vector<int> > hashmap(triangle);
        for (int i = 0; i != hashmap.size(); ++i) {
            for (int j = 0; j != hashmap[i].size(); ++j) {
                hashmap[i][j] = INT_MIN;
            }
        }
        int result = dfs(0, 0, triangle, hashmap);

        return result;
    }

private:
    int dfs(int x, int y, vector<vector<int> > &triangle, vector<vector<int> > &hashmap)
        const int n = triangle.size();
        if (x == n) {
            return 0;
        }

        // INT_MIN means no value yet
        if (hashmap[x][y] != INT_MIN) {
            return hashmap[x][y];
        }
        int x1y = dfs(x + 1, y, triangle, hashmap);
```

```
        int x1y1 = dfs(x + 1, y + 1, triangle, hashmap);
        hashmap[x][y] =  min(x1y, x1y1) + triangle[x][y];

        return hashmap[x][y];
    }
};
```

由于已经计算出的最短路径值不再重复计算，计算复杂度由之前的 $O(2^n)$，变为 $O(n^2)$, 每个坐标的元素仅计算一次，故共计算的次数为 $1+2+...+n \approx O(n^2)$.

Method 4 - Dynamic Programming

从主章节中对动态规划的简介我们可以知道使用动态规划的难点和核心在于**状态的定义及转化方程的建立**。那么问题来了，到底如何去找适合这个问题的状态及转化方程呢?

我们仔细分析题中可能的状态和转化关系，发现从 `triangle` 中坐标为 $triangle[x][y]$ 的元素出发，其路径只可能为 $triangle[x][y] -> triangle[x+1][y]$ 或者 $triangle[x][y] -> triangle[x+1][y+1]$. 以点 (x,y) 作为参考，那么可能的状态 $f(x,y)$ 就可以是:

1. 从 (x,y) 出发走到最后一行的最短路径和
2. 从 $(0,0)$ 走到 (x,y)的最短路径和

如果选择1作为状态，则相应的状态转移方程为:

$$f_1(x,y) = min\{f_1(x+1,y), f_1(x+1,y+1)\} + triangle[x][y]$$

如果选择2作为状态，则相应的状态转移方程为:

$$f_2(x,y) = min\{f_2(x-1,y), f_2(x-1,y-1)\} + triangle[x][y]$$

两个状态所对应的初始状态分别为 $f_1(n-1,y), 0 \le y \le n-1$ 和 $f_2(0,0)$. 在代码中应注意考虑边界条件。下面分别就这种不同的状态进行动态规划。

C++ From Bottom to Top

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int> > &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        vector<vector<int> > hashmap(triangle);

        // get the total row number of triangle
        const int N = triangle.size();
```

```
        for (int i = 0; i != N; ++i) {
            hashmap[N-1][i] = triangle[N-1][i];
        }

        for (int i = N - 2; i >= 0; --i) {
            for (int j = 0; j < i + 1; ++j) {
                hashmap[i][j] = min(hashmap[i + 1][j], hashmap[i + 1][j + 1]) + triangle[
            }
        }

        return hashmap[0][0];
    }
};
```

源码分析

1. 异常处理
2. 使用hashmap保存结果
3. 初始化 `hashmap[N-1][i]` ，由于是自底向上，故初始化时保存最后一行元素
4. 使用自底向上的方式处理循环
5. 最后返回结果hashmap[0][0]

从空间利用角度考虑也可直接使用triangle替代hashmap，但是此举会改变triangle的值，不推荐。

C++ From Top to Bottom

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int> > &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        vector<vector<int> > hashmap(triangle);

        // get the total row number of triangle
        const int N = triangle.size();
        //hashmap[0][0] = triangle[0][0];
        for (int i = 1; i != N; ++i) {
            for (int j = 0; j <= i; ++j) {
                if (j == 0) {
                    hashmap[i][j] = hashmap[i - 1][j];
                }
                if (j == i) {
                    hashmap[i][j] = hashmap[i - 1][j - 1];
                }
                if ((j > 0) && (j < i)) {
                    hashmap[i][j] = min(hashmap[i - 1][j], hashmap[i - 1][j - 1]);
                }
```

```
                hashmap[i][j] += triangle[i][j];
            }
        }

        int result = INT_MAX;
        for (int i = 0; i != N; ++i) {
            result = min(result, hashmap[N - 1][i]);
        }
        return result;
    }
};
```

源码解析

自顶向下的实现略微有点复杂，在寻路时需要考虑最左边和最右边的边界，还需要在最后返回结果时比较最小值。

Backpack

Source

- lintcode: (92) Backpack

```
Given n items with size A[i], an integer m denotes the size of a backpack. How full you c

Note
You can not divide any item into small pieces.

Example
If we have 4 items with size [2, 3, 5, 7], the backpack size is 11, we can select 2, 3 an

You function should return the max size we can fill in the given backpack.
```

题解

本题是典型的01背包问题，每种类型的物品最多只能选择一件。先来看看 九章算法 的### 题解

1. 状态: result[i][S] 表示前i个物品，取出一些物品能否组成体积和为S的背包
2. 状态转移方程: $f[i][S] = f[i-1][S-A[i]]\ or\ f[i-1][S]$ (A[i]为第i个物品的大小)
   - 欲从前i个物品中取出一些组成体积和为S的背包，可从两个状态转换得到。
     i. $f[i-1][S-A[i]]$: **放入第i个物品**，前 $i-1$ 个物品能否取出一些体积和为 $S-A[i]$ 的背包。
     ii. $f[i-1][S]$: **不放入第i个物品**，前 $i-1$ 个物品能否取出一些组成体积和为S的背包。
3. 状态初始化: $f[1\cdots n][0] = true;\ f[0][1\cdots m] = false.$ 前1~n个物品组成体积和为0的背包始终为真，其他情况为假。
4. 返回结果: 寻找使 $f[n][S]$ 值为true的最大S ($1 \le S \le m$)

C++ 2D vector for result

```cpp
class Solution {
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    int backPack(int m, vector<int> A) {
        if (A.empty() || m < 1) {
            return 0;
        }

        const int N = A.size() + 1;
        const int M = m + 1;
        vector<vector<bool> > result;
```

```
        result.resize(N);
        for (vector<int>::size_type i = 0; i != N; ++i) {
            result[i].resize(M);
            std::fill(result[i].begin(), result[i].end(), false);
        }

        result[0][0] = true;
        for (int i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                if (j < A[i - 1]) {
                    result[i][j] = result[i - 1][j];
                } else {
                    result[i][j] = result[i - 1][j] || result[i - 1][j - A[i - 1]];
                }
            }
        }

        // return the largest i if true
        for (int i = M; i > 0; --i) {
            if (result[N - 1][i - 1]) {
                return (i - 1);
            }
        }
        return 0;
    }
};
```

源码分析

1. 异常处理
2. 初始化结果矩阵，注意这里需要使用 `resize` 而不是 `reserve` ，否则可能会出现段错误
3. 实现状态转移逻辑，一定要分 `j < A[i - 1]` 与否来讨论
4. 返回结果，只需要比较 `result[N - 1][i - 1]` 的结果，返回true的最大值

状态转移逻辑中代码可以进一步简化，即：

```
        for (int i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                result[i][j] = result[i - 1][j];
                if (j >= A[i - 1] && result[i - 1][j - A[i - 1]]) {
                    result[i][j] = true;
                }
            }
        }
```

考虑背包问题的核心——状态转移方程，如何优化此转移方程？原始方案中用到了二维矩阵来保存result，注意到result的第i行仅依赖于第i-1行的结果，那么能否用一维数组来代替这种隐含的关系呢？我们**在内循环j处递减即可**。如此即可避免 `result[i][S]` 的值由本轮 `result[i][S-A[i]]` 递推得到。

C++ 1D vector for result

```
class Solution {
```

```
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    int backPack(int m, vector<int> A) {
        if (A.empty() || m < 1) {
            return 0;
        }

        const int N = A.size();
        vector<bool> result;
        result.resize(m + 1);
        std::fill(result.begin(), result.end(), false);

        result[0] = true;
        for (int i = 0; i != N; ++i) {
            for (int j = m; j >= 0; --j) {
                if (j >= A[i] && result[j - A[i]]) {
                    result[j] = true;
                }
            }
        }

        // return the largest i if true
        for (int i = m; i > 0; --i) {
            if (result[i]) {
                return i;
            }
        }
        return 0;
    }
};
```

Backpack II

Source

- lintcode: (125) Backpack II

```
Given n items with size A[i] and value V[i], and a backpack with size m. What's the maxim

Note
You cannot divide item into small pieces and the total size of items you choose should sm
Example
Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and a backpack with size 10.
```

题解

首先定义状态 $K(i,w)$ 为前 i 个物品放入size为 w 的背包中所获得的最大价值，则相应的状态转移方程

为： $K(i,w)=\max\{K(i-1,w),K(i-1,w-w_i)+v_i\}$

详细分析过程见本节。

C++ 2D vector for result

```
class Solution {
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    int backPackII(int m, vector<int> A, vector<int> V) {
        if (A.empty() || V.empty() || m < 1) {
            return 0;
        }
        const int N = A.size() + 1;
        const int M = m + 1;
        vector<vector<int> > result;
        result.resize(N);
        for (vector<int>::size_type i = 0; i != N; ++i) {
            result[i].resize(M);
            std::fill(result[i].begin(), result[i].end(), 0);
        }

        for (vector<int>::size_type i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                if (j < A[i - 1]) {
                    result[i][j] = result[i - 1][j];
                } else {
                    int temp = result[i - 1][j - A[i - 1]] + V[i - 1];
                    result[i][j] = max(temp, result[i - 1][j]);
                }
            }
        }

        return result[N - 1][M - 1];
    }
};
```

源码分析

1. 使用二维矩阵保存结果result
2. 返回result矩阵的右下角元素——背包size限制为m时的最大价值

按照第一题backpack的思路，这里可以使用一维数组进行空间复杂度优化。优化方法为逆序求 `result[j]`，优化后的代码如下：

C++ 1D vector for result

```
class Solution {
```

```
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    int backPackII(int m, vector<int> A, vector<int> V) {
        if (A.empty() || V.empty() || m < 1) {
            return 0;
        }

        const int M = m + 1;
        vector<int> result;
        result.resize(M);
        std::fill(result.begin(), result.end(), 0);

        for (vector<int>::size_type i = 0; i != A.size(); ++i) {
            for (int j = m; j >= 0; --j) {
                if (j < A[i]) {
                    // result[j] = result[j];
                } else {
                    int temp = result[j - A[i]] + V[i];
                    result[j] = max(temp, result[j]);
                }
            }
        }

        return result[M - 1];
    }
};
```

Reference

- Lintcode: Backpack - neverlandly - 博客园
- Lintcode: Backpack II - neverlandly - 博客园
- 九章算法 | 背包问题
- 崔添翼 § 翼若垂天之云 › 《背包问题九讲》2.0 alpha1

Minimum Path Sum

- tags: [DP_Matrix]

Source

- lintcode: (110) Minimum Path Sum

```
Given a m x n grid filled with non-negative numbers, find a path from top left to bottom

Note
You can only move either down or right at any point in time.
```

题解

1. State: f[x][y] 从坐标(0,0)走到(x,y)的最短路径和
2. Function: f[x][y] = (x, y) + min{f[x-1][y], f[x][y-1]}
3. Initialization: f[0][0] = A[0][0], f[i][0] = sum(0,0 -> i,0), f[0][i] = sum(0,0 -> 0,i)
4. Answer: f[m-1][n-1]

注意最后返回为f[m-1][n-1]而不是f[m][n].

首先看看如下正确但不合适的答案，OJ上会出现TLE。 未使用hashmap并且使用了递归的错误版本。

C++ dfs without hashmap: ~~Wrong answer~~

```cpp
class Solution {
public:
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    int minPathSum(vector<vector<int> > &grid) {
        if (grid.empty()) {
            return 0;
        }

        const int m = grid.size() - 1;
        const int n = grid[0].size() - 1;

        return helper(grid, m, n);

    }

private:
    int helper(vector<vector<int> > &grid_in, int x, int y) {
        if (0 == x && 0 == y) {
            return grid_in[0][0];
```

```
        }
        if (0 == x) {
            return helper(grid_in, x, y - 1) + grid_in[x][y];
        }
        if (0 == y) {
            return helper(grid_in, x - 1, y) + grid_in[x][y];
        }

        return grid_in[x][y] + min(helper(grid_in, x - 1, y), helper(grid_in, x, y - 1));
    }
};
```

使用迭代思想进行求解的正确实现：

C++ Iterative

```
class Solution {
public:
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    int minPathSum(vector<vector<int> > &grid) {
        if (grid.empty() || grid[0].empty()) {
            return 0;
        }

        const int M = grid.size();
        const int N = grid[0].size();
        vector<vector<int> > ret(M, vector<int> (N, 0));

        ret[0][0] = grid[0][0];
        for (int i = 1; i != M; ++i) {
            ret[i][0] = grid[i][0] + ret[i - 1][0];
        }
        for (int i = 1; i != N; ++i) {
            ret[0][i] = grid[0][i] + ret[0][i - 1];
        }

        for (int i = 1; i != M; ++i) {
            for (int j = 1; j != N; ++j) {
                ret[i][j] = grid[i][j] + min(ret[i - 1][j], ret[i][j - 1]);
            }
        }

        return ret[M - 1][N - 1];
    }
};
```

源码分析

1. 异常处理，不仅要对grid还要对grid[0]分析
2. 对返回结果矩阵进行初始化，注意ret[0][0]须单独初始化以便使用ret[i-1]

3. 递推时i和j均从1开始
4. 返回结果ret[M-1][N-1]，注意下标是从0开始的

此题还可进行空间复杂度优化，和背包问题类似，使用一维数组代替二维矩阵也行，具体代码可参考 水中的鱼: [LeetCode] Minimum Path Sum 解题报告

优化空间复杂度，要么对行遍历进行优化，要么对列遍历进行优化，通常我们习惯先按行遍历再按列遍历，有状态转移方程 f[x][y] = (x, y) + min{f[x-1][y], f[x][y-1]} 知，想要优化行遍历，那么f[y]保存的值应为第x行第y列的和。由于无行下标信息，故初始化时仅能对第一个元素初始化，分析时建议画图理解。

C++ 1D vector

```
class Solution {
public:
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    int minPathSum(vector<vector<int> > &grid) {
        if (grid.empty() || grid[0].empty()) {
            return 0;
        }

        const int M = grid.size();
        const int N = grid[0].size();
        vector<int> ret(N, INT_MAX);

        ret[0] = 0;

        for (int i = 0; i != M; ++i) {
            ret[0] =  ret[0] + grid[i][0];
            for (int j = 1; j != N; ++j) {
                ret[j] = grid[i][j] + min(ret[j], ret[j - 1]);
            }
        }

        return ret[N - 1];
    }
};
```

初始化时需要设置为 `INT_MAX`，便于 `i = 0` 时取 `ret[j]`.

Unique Paths

- tags: [DP_Matrix]

Source

- lintcode: (114) Unique Paths

```
A robot is located at the top-left corner of a m x n grid
(marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time.
The robot is trying to reach the bottom-right corner of the grid
(marked 'Finish' in the diagram below).

How many possible unique paths are there?

Note
m and n will be at most 100.
```

题解

题目要求：给定*m x n*矩阵，求左上角到右下角的路径总数，每次只能向左或者向右前进。按照动态规划中矩阵类问题的通用方法：

1. State: f[m][n] 从起点到坐标(m,n)的路径数目
2. Function: f[m][n] = f[m-1][n] + f[m][n-1] 分析终点与左边及右边节点的路径数，发现从左边或者右边到达终点的路径一定不会重合，相加即为唯一的路径总数
3. Initialization: f[i][j] = 1, 到矩阵中任一节点均至少有一条路径，其实关键之处在于给第0行和第0列初始化，免去了单独遍历第0行和第0列进行初始化
4. Answer: f[m - 1][n - 1]

C++

```
class Solution {
public:
    /**
     * @param n, m: positive integer (1 <= n ,m <= 100)
     * @return an integer
     */
    int uniquePaths(int m, int n) {
        if (m < 1 || n < 1) {
            return 0;
        }

        vector<vector<int> > ret(m, vector<int>(n, 1));

        for (int i = 1; i != m; ++i) {
```

```
            for (int j = 1; j != n; ++j) {
                ret[i][j] = ret[i - 1][j] + ret[i][j - 1];
            }
        }

        return ret[m - 1][n - 1];
    }
};
```

源码分析

1. 异常处理，虽然题目有保证为正整数，但还是判断一下以防万一
2. 初始化二维矩阵，值均为1
3. 按照转移矩阵函数进行累加
4. 任何 `ret[m - 1][n - 1]`

Unique Paths II

- tags: [DP_Matrix]

Source

- lintcode: (115) Unique Paths II

```
Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids.
How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.
Note
m and n will be at most 100.

Example
For example,
There is one obstacle in the middle of a 3x3 grid as illustrated below.

[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
The total number of unique paths is 2.
```

题解

在上题的基础上加了obstacal这么一个限制条件，那么也就意味着凡是遇到障碍点，其路径数马上变为0，需要注意的是初始化环节和上题有较大不同。首先来看看错误的初始化实现。

C++ initialization error

```cpp
class Solution {
public:
    /**
     * @param obstacleGrid: A list of lists of integers
     * @return: An integer
     */
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        if(obstacleGrid.empty() || obstacleGrid[0].empty()) {
            return 0;
        }

        const int M = obstacleGrid.size();
        const int N = obstacleGrid[0].size();

        vector<vector<int> > ret(M, vector<int>(N, 0));
```

```
        for (int i = 0; i != M; ++i) {
            if (0 == obstacleGrid[i][0]) {
                ret[i][0] = 1;
            }
        }
        for (int i = 0; i != N; ++i) {
            if (0 == obstacleGrid[0][i]) {
                ret[0][i] = 1;
            }
        }

        for (int i = 1; i != M; ++i) {
            for (int j = 1; j != N; ++j) {
                if (obstacleGrid[i][j]) {
                    ret[i][j] = 0;
                } else {
                    ret[i][j] = ret[i -1][j] + ret[i][j - 1];
                }
            }
        }

        return ret[M - 1][N - 1];
    }
};
```

源码分析

错误之处在于初始化第0行和第0列时，未考虑到若第0行/列有一个坐标出现障碍物，则当前行/列后的元素路径数均为0！

C++

```
class Solution {
public:
    /**
     * @param obstacleGrid: A list of lists of integers
     * @return: An integer
     */
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        if(obstacleGrid.empty() || obstacleGrid[0].empty()) {
            return 0;
        }

        const int M = obstacleGrid.size();
        const int N = obstacleGrid[0].size();

        vector<vector<int> > ret(M, vector<int>(N, 0));

        for (int i = 0; i != M; ++i) {
            if (obstacleGrid[i][0]) {
                break;
            } else {
                ret[i][0] = 1;
            }
```

```
        }
        for (int i = 0; i != N; ++i) {
            if (obstacleGrid[0][i]) {
                break;
            } else {
                ret[0][i] = 1;
            }
        }

        for (int i = 1; i != M; ++i) {
            for (int j = 1; j != N; ++j) {
                if (obstacleGrid[i][j]) {
                    ret[i][j] = 0;
                } else {
                    ret[i][j] = ret[i -1][j] + ret[i][j - 1];
                }
            }
        }

        return ret[M - 1][N - 1];
    }
};
```

源码分析

1. 异常处理
2. 初始化二维矩阵(全0阵)，尤其注意遇到障碍物时应 `break` 跳出当前循环
3. 递推路径数
4. 返回 `ret[M - 1][N - 1]`

Climbing Stairs

Source

- lintcode: (111) Climbing Stairs

```
You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps.
In how many distinct ways can you climb to the top?

Example
Given an example n=3 , 1+1+1=2+1=1+2=3

return 3
```

题解

题目问的是到达顶端的方法数，我们采用序列类问题的通用分析方法，可以得到如下四要素：

1. State: f[i] 爬到第i级的方法数
2. Function: f[i]=f[i-1]+f[i-2]
3. Initialization: f[0]=1,f[1]=1
4. Answer: f[n]

尤其注意状态转移方程的写法，f[i]只可能由两个中间状态转化而来，一个是f[i-1]，由f[i-1]到f[i]其方法总数并未增加；另一个是f[i-2]，由f[i-2]到f[i]隔了两个台阶，因此有1+1和2两个方法，因此容易写成 f[i]=f[i-1]+f[i-2]+1，但仔细分析后能发现，由f[i-2]到f[i]的中间状态f[i-1]已经被利用过一次，故f[i]=f[i-1]+f[i-2]. 使用动规思想解题时需要分清『重叠子状态』, 如果有重复的需要去重。

C++

```cpp
class Solution {
public:
    /**
     * @param n: An integer
     * @return: An integer
     */
    int climbStairs(int n) {
        if (n < 1) {
            return 0;
        }

        vector<int> ret(n + 1, 1);

        for (int i = 2; i != n + 1; ++i) {
            ret[i] = ret[i - 1] + ret[i - 2];
        }
```

```
        return ret[n];
    }
};
```

1. 异常处理
2. 初始化n+1个元素，初始值均为1。之所以用n+1个元素是下标分析起来更方便
3. 状态转移方程
4. 返回ret[n]

初始化ret[0]也为1，可以认为到第0级也是一种方法。

以上答案的空间复杂度为 $O(n)$，仔细观察后可以发现在状态转移方程中，我们可以使用三个变量来替代长度为n+1的数组。具体代码可参考 climbing-stairs | 九章算法

C++

```
class Solution {
public:
    /**
     * @param n: An integer
     * @return: An integer
     */
    int climbStairs(int n) {
        if (n < 1) {
            return 0;
        }

        int ret0 = 1, ret1 = 1, ret2 = 1;

        for (int i = 2; i != n + 1; ++i) {
            ret0 = ret1 + ret2;
            ret2 = ret1;
            ret1 = ret0;
        }

        return ret0;
    }
};
```

Jump Game

Source

- lintcode:

(116) Jump Game

```
Given an array of non-negative integers, you are initially positioned at the first index

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Example
A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.
```

题解(自顶向下-动态规划)

1. State: f[i] 从起点出发能否达到i
2. Function: `f[i] = OR (f[j], j < i ~\&\&~ j + A[j] \geq i)`, 状态 j 转移到 i, 所有小于i的下标j的元素中是否存在能从j跳转到i得
3. Initialization: f[0] = true;
4. Answer: 递推到第 N - 1 个元素时，f[N-1]

这种自顶向下的方法需要使用额外的 $O(n)$ 空间，保存小于N-1时的状态。且时间复杂度在恶劣情况下有可能变为 $1+2+\cdots+n=O(n^2)$, 出现 TLE 无法AC的情况，不过工作面试能给出这种动规的实现就挺好的了。

C++ from top to bottom

```
class Solution {
public:
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    bool canJump(vector<int> A) {
        if (A.empty()) {
            return true;
        }

        vector<bool> jumpto(A.size(), false);
        jumpto[0] = true;
```

```
        for (int i = 1; i != A.size(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (jumpto[j] && (j + A[j] >= i)) {
                    jumpto[i] = true;
                    break;
                }
            }
        }

        return jumpto[A.size() - 1];
    }
};
```

题解(自底向上-贪心法)

题意为问是否能从起始位置到达最终位置，我们首先分析到达最终位置的条件，从坐标i出发所能到达最远的位置为 $f[i] = i + A[i]$，如果要到达最终位置，即存在某个 i 使得$f[i] \geq N - 1$, 而想到达 i, 则又需存在某个 j 使得 $f[j] \geq i - 1$. 依此类推直到下标为0.

以下分析形式虽为动态规划，实则贪心法！

1. State: f[i] 从 i 出发能否到达最终位置
2. Function: $f[j] = j + A[j] \geq i$, 状态 j 转移到 i, 置为 `true`
3. Initialization: 第一个为 `true` 的元素为 `A.size() - 1`
4. Answer: 递推到第 0 个元素时，若其值为 `true` 返回 `true`

C++ greedy, from bottom to top

```
class Solution {
public:
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    bool canJump(vector<int> A) {
        if (A.empty()) {
            return true;
        }

        int index_true = A.size() - 1;
        for (int i = A.size() - 1; i >= 0; --i) {
            if (i + A[i] >= index_true) {
                index_true = i;
            }
        }

        return 0 == index_true ? true : false;
    }
};
```

题解(自顶向下-贪心法)

针对上述自顶向下可能出现时间复杂度过高的情况，下面使用贪心思想对i进行递推，每次遍历A中的一个元素时更新最远可能到达的元素，最后判断最远可能到达的元素是否大于 `A.size() - 1`

C++ greedy, from top to bottom

```
class Solution {
public:
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    bool canJump(vector<int> A) {
        if (A.empty()) {
            return true;
        }

        int farthest = A[0];

        for (int i = 1; i != A.size(); ++i) {
            if ((i <= farthest) && (i + A[i] > farthest)) {
                farthest = i + A[i];
            }
        }

        return farthest >= A.size() - 1;
    }
};
```

Word Break

- tags: [DP_Sequence]

Source

- leetcode: Word Break | LeetCode OJ
- lintcode: (107) Word Break

```
Given a string s and a dictionary of words dict, determine if s can be
segmented into a space-separated sequence of one or more dictionary words.

For example, given
s = "leetcode",
dict = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".
```

题解

单序列(DP_Sequence) DP 题，由单序列动态规划的四要素可大致写出：

1. State: `f[i]` 表示前 `i` 个字符能否根据词典中的词被成功分词。
2. Function: `f[i] = or{f[j], j < i, letter in [j+1, i] can be found in dict}`，含义为小于 `i` 的索引 `j` 中只要有一个 `f[j]` 为真且 `j+1` 到 `i` 中组成的字符能在词典中找到时，`f[i]` 即为真，否则为假。具体实现可分为自顶向下或者自底向上。
3. Initialization: `f[0] = true`，数组长度为字符串长度 + 1，便于处理。
4. Answer: `f[s.length]`

考虑到单词长度通常不会太长，故在 `s` 较长时使用自底向上效率更高。

Python

```python
class Solution:
    # @param s, a string
    # @param wordDict, a set<string>
    # @return a boolean
    def wordBreak(self, s, wordDict):
        if not s:
            return True
        if not wordDict:
            return False

        max_word_len = max([len(w) for w in wordDict])
        can_break = [True]
        for i in xrange(len(s)):
            can_break.append(False)
```

```
            for j in xrange(i, -1, -1):
                # optimize for too long interval
                if i - j + 1 > max_word_len:
                    break
                if can_break[j] and s[j:i + 1] in wordDict:
                    can_break[i + 1] = True
                    break
        return can_break[-1]
```

C++

```
class Solution {
public:
    bool wordBreak(string s, unordered_set<string>& wordDict) {
        if (s.empty()) return true;
        if (wordDict.empty()) return false;

        // get the max word length of wordDict
        int max_word_len = 0;
        for (unordered_set<string>::iterator it = wordDict.begin();
         it != wordDict.end(); ++it) {

            max_word_len = max(max_word_len, (*it).size());
        }

        vector<bool> can_break(s.size() + 1, false);
        can_break[0] = true;
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                // optimize for too long interval
                if (i - j > max_word_len) break;

                if (can_break[j] &&
            wordDict.find(s.substr(j, i - j)) != wordDict.end()) {

                    can_break[i] = true;
                    break;
                }
            }
        }

        return can_break[s.size()];
    }
};
```

Java

```
public class Solution {
    public boolean wordBreak(String s, Set<String> wordDict) {
        if (s == null || s.length() == 0) return true;
        if (wordDict == null || wordDict.isEmpty()) return false;

        // get the max word length of wordDict
        int max_word_len = 0;
```

```
        for (String word : wordDict) {
            max_word_len = Math.max(max_word_len, word.length());
        }

        boolean[] can_break = new boolean[s.length() + 1];
        can_break[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = i - 1; j >= 0; j--) {
                // optimize for too long interval
                if (i - j > max_word_len) break;

                String word = s.substring(j, i);
                if (can_break[j] && wordDict.contains(word)) {
                    can_break[i] = true;
                    break;
                }
            }
        }

        return can_break[s.length()];
    }
}
```

源码分析

Python 之类的动态语言无需初始化指定大小的数组，使用时下标 `i` 比 C++和 Java 版的程序少1。使用自底向上的方法求解状态转移，首先遍历一次词典求得单词最大长度以便后续优化。

复杂度分析

1. 求解词典中最大单词长度，时间复杂度为词典长度乘上最大单词长度 $O(L_D \cdot L_w)$
2. 词典中找单词的时间复杂度为 $O(1)$(哈希表结构)
3. 两重 for 循环，内循环在超出最大单词长度时退出，故最坏情况下两重 for 循环的时间复杂度为 $O(nL_w)$.
4. 故总的时间复杂度近似为 $O(nL_w)$.
5. 使用了与字符串长度几乎等长的布尔数组和临时单词 `word` ，空间复杂度近似为 $O(n)$.

Longest Increasing Subsequence

- tags: [DP_Sequence]

Source

- lintcode: (76) Longest Increasing Subsequence
- Dynamic Programming | Set 3 (Longest Increasing Subsequence) - GeeksforGeeks

```
Given a sequence of integers, find the longest increasing subsequence (LIS).
You code should return the length of the LIS.
Example
For [5, 4, 1, 2, 3], the LIS  is [1, 2, 3], return 3

For [4, 2, 4, 5, 3, 7], the LIS is [4, 4, 5, 7], return 4

Challenge
Time complexity O(n^2) or O(nlogn)

Clarification
What's the definition of longest increasing subsequence?

    * The longest increasing subsequence problem is to find a subsequence of
    a given sequence in which the subsequence's elements are in sorted order,
    lowest to highest, and in which the subsequence is as long as possible.
    This subsequence is not necessarily contiguous, or unique.

    * https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
```

题解

由题意知这种题应该是单序列动态规划题，结合四要素，可定义 `f[i]` 为前 `i` 个数字中的 LIC 数目，那么问题来了，接下来的状态转移方程如何写？似乎写不出来... 再仔细看看 LIS 的定义，状态转移的关键一环应该为数字本身而不是最后返回的结果(数目)，那么理所当然的，我们应定义 `f[i]` 为前 `i` 个数字中以第 `i` 个数字结尾的 LIS 长度，相应的状态转移方程为 `f[i] = {1 + max{f[j]} where j < i, nums[j] < nums[i]}`，该转移方程的含义为在所有满足以上条件的 j 中将最大的 `f[j]` 赋予 `f[i]`，如果上式不满足，则 `f[i] = 1`. 具体实现时不能直接使用 `f[i] = 1 + max(f[j])`，应为若 `if f[i] < 1 + f[j], f[i] = 1 + f[j]`. 最后返回 `max(f[])`.

Python

```python
class Solution:
    """
    @param nums: The integer array
    @return: The length of LIS (longest increasing subsequence)
    """
    def longestIncreasingSubsequence(self, nums):
```

```
        if not nums:
            return 0

        lis = [1] * len(nums)
        for i in xrange(1, len(nums)):
            for j in xrange(i):
                if nums[j] <= nums[i] and lis[i] < 1 + lis[j]:
                    lis[i] = 1 + lis[j]
        return max(lis)
```

C++

```
class Solution {
public:
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequence)
     */
    int longestIncreasingSubsequence(vector<int> nums) {
        if (nums.empty()) return 0;

        int len = nums.size();
        vector<int> lis(len, 1);

        for (int i = 1; i < len; ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[j] <= nums[i] && (lis[i] < lis[j] + 1)) {
                    lis[i] = 1 + lis[j];
                }
            }
        }

        return *max_element(lis.begin(), lis.end());
    }
};
```

Java

```
public class Solution {
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequence)
     */
    public int longestIncreasingSubsequence(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int[] lis = new int[nums.length];
        Arrays.fill(lis, 1);

        for (int i = 1; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[j] <= nums[i] && (lis[i] < lis[j] + 1)) {
                    lis[i] = lis[j] + 1;
                }
```

```
            }
        }

        // get the max lis
        int max_lis = 0;
        for (int i = 0; i < lis.length; i++) {
            if (lis[i] > max_lis) {
                max_lis = lis[i];
            }
        }

        return max_lis;
    }
}
```

源码分析

1. 初始化数组，初始值为1
2. 根据状态转移方程递推求得 `lis[i]`
3. 遍历 `lis` 数组求得最大值

复杂度分析

使用了与 nums 等长的空间，空间复杂度 $O(n)$. 两重 for 循环，最坏情况下 $O(n^2)$, 遍历求得最大值，时间复杂度为 $O(n)$, 故总的时间复杂度为 $O(n^2)$.

Palindrome Partitioning II

- tags: [DP_Sequence]

Source

- leetcode: Palindrome Partitioning II | LeetCode OJ
- lintcode: (108) Palindrome Partitioning II

```
Given a string s, cut s into some substrings such that
every substring is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Example
For example, given s = "aab",

Return 1 since the palindrome partitioning ["aa","b"] could be produced
using 1 cut.
```

题解1 - 仅对最小切割数使用动态规划

此题为难题，费了我九牛二虎之力才bug-free :(求最小切分数，非常明显的动规暗示。由问题出发可建立状态 `f[i]` 表示到索引 `i` 处时需要的最少切割数(即切割前 i 个字符组成的字符串)，状态转移方程为 `f[i] = min{1 + f[j]}, where j < i and substring [j, i] is palindrome` , 判断区间[j, i] 是否为回文简单的方法可反转后比较。

Python

```python
class Solution:
    # @param s, a string
    # @return an integer
    def minCut(self, s):
        if not s:
            print 0

        cut = [i - 1 for i in xrange(1 + len(s))]

        for i in xrange(1 + len(s)):
            for j in xrange(i):
                # s[j:i] is palindrome
                if s[j:i] == s[j:i][::-1]:
                    cut[i] = min(cut[i], 1 + cut[j])
        return cut[-1]
```

源码分析

1. 当 s 为 None 或者列表为空时返回0
2. 初始化切割数数组
3. 子字符串的索引位置可为 `[0, len(s) - 1]`，内循环 j 比外循环 i 小，故可将 i 的最大值设为 `1 + len(s)` 较为便利。
4. 回文的判断使用了 `[::-1]` 对字符串进行反转
5. 最后返回数组最后一个元素

复杂度分析

两重循环，遍历的总次数为 $1/2 \cdots n^2)$, 每次回文的判断时间复杂度为 $O(len(s))$, 故总的时间复杂度近似为 $O(n^3)$. 在 s 长度较长时会 TLE. 使用了与 s 等长的辅助切割数数组，空间复杂度近似为 $O(n)$.

题解2 - 使用动态规划计算子字符串回文状态

切割数部分使用的是动态规划，优化的空间不大，仔细瞅瞅可以发现在判断字符串是否为回文的部分存在大量重叠计算，故可引入动态规划进行优化，时间复杂度可优化至到平方级别。

定义状态 PaMat[i][j] 为区间 `[i,j]` 是否为回文的标志, 对应此状态的子问题可从回文的定义出发，如果字符串首尾字符相同且在去掉字符串首尾字符后字符串仍为回文，则原字符串为回文，相应的状态转移方程 `PaMat[i][j] = s[i] == s[j] && PaMat[i+1][j-1]`，由于状态转移方程中依赖比 `i` 大的结果，故实现中需要从索引大的往索引小的递推，另外还需要考虑一些边界条件和初始化方式，做到 bug-free 需要点时间。

Python

```
class Solution:
    # @param s, a string
    # @return an integer
    def minCut(self, s):
        if not s:
            print 0

        cut = [i - 1 for i in xrange(1 + len(s))]
        PaMatrix = self.getMat(s)

        for i in xrange(1 + len(s)):
            for j in xrange(i):
                if PaMatrix[j][i - 1]:
                    cut[i] = min(cut[i], cut[j] + 1)
        return cut[-1]

    def getMat(self, s):
        PaMat = [[True for i in xrange(len(s))] for j in xrange(len(s))]
        for i in xrange(len(s), -1, -1):
            for j in xrange(i, len(s)):
                if j == i:
                    PaMat[i][j] = True
        # not necessary if init with True
                # elif j == i + 1:
                #     PaMat[i][j] = s[i] == s[j]
```

```
                else:
                    PaMat[i][j] = s[i] == s[j] and PaMat[i + 1][j - 1]
        return PaMat
```

C++

```
class Solution {
public:
    int minCut(string s) {
        if (s.empty()) return 0;

        int len = s.size();
        vector<int> cut;
        for (int i = 0; i < 1 + len; ++i) {
            cut.push_back(i - 1);
        }
        vector<vector<bool> > mat = getMat(s);

        for (int i = 1; i < 1 + len; ++i) {
            for (int j = 0; j < i; ++j) {
                if (mat[j][i - 1]) {
                    cut[i] = min(cut[i], 1 + cut[j]);
                }
            }
        }

        return cut[len];
    }

    vector<vector<bool> > getMat(string s) {
        int len = s.size();
        vector<vector<bool> > mat = vector<vector<bool> >(len, vector<bool>(len, true));
        for (int i = len; i >= 0; --i) {
            for (int j = i; j < len; ++j) {
                if (j == i) {
                    mat[i][j] = true;
                } else if (j == i + 1) {
                    mat[i][j] = (s[i] == s[j]);
                } else {
                    mat[i][j] = ((s[i] == s[j]) && mat[i + 1][j - 1]);
                }
            }
        }

        return mat;
    }
};
```

Java

```
public class Solution {
    public int minCut(String s) {
        if (s == null || s.length() == 0) return 0;
```

```
        int len = s.length();
        int[] cut = new int[1 + len];
        for (int i = 0; i < 1 + len; ++i) {
            cut[i] = i - 1;
        }
        boolean[][] mat = paMat(s);

        for (int i = 1; i < 1 + len; i++) {
            for (int j = 0; j < i; j++) {
                if (mat[j][i - 1]) {
                    cut[i] = Math.min(cut[i], 1 + cut[j]);
                }
            }
        }

        return cut[len];
    }

    private boolean[][] paMat(String s) {
        int len = s.length();
        boolean[][] mat = new boolean[len][len];

        for (int i = len - 1; i >= 0; i--) {
            for (int j = i; j < len; j++) {
                if (j == i) {
                    mat[i][j] = true;
                } else if (j == i + 1) {
                    mat[i][j] = (s.charAt(i) == s.charAt(j));
                } else {
                    mat[i][j] = (s.charAt(i) == s.charAt(j)) && mat[i + 1][j - 1];
                }
            }
        }

        return mat;
    }
}
```

源码分析

初始化 cut 长度为 `1 + len(s)`，`cut[0] = -1` 便于状态转移方程实现。在执行 `mat[i][j] == ... mat[i + 1][j - 1]` 时前提是 `j - 1 > i + 1`，所以才需要分情况赋值。使用 `getMat` 得到字符串区间的回文矩阵，由于cut 的长度为1+len(s), 两重 for 循环时需要注意索引的取值，这个地方非常容易错。

复杂度分析

最坏情况下每次 for 循环都遍历 n 次，时间复杂度近似为 $O(n^2)$, 使用了二维回文矩阵保存记忆化搜索结果，空间复杂度为 $O(n^2)$.

Reference

- Palindrome Partitioning II 参考程序 Java/C++/Python

- soulmachine 的 leetcode 题解

Longest Common Subsequence

- tags: [DP_Two_Sequence]

Source

- lintcode: (77) Longest Common Subsequence

```
Given two strings, find the longest common subsequence (LCS).

Your code should return the length of LCS.

Have you met this question in a real interview? Yes
Example
For "ABCD" and "EDCA", the LCS is "A" (or "D", "C"), return 1.

For "ABCD" and "EACB", the LCS is "AC", return 2.

Clarification
What's the definition of Longest Common Subsequence?

https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
http://baike.baidu.com/view/2020307.htm
```

题解

求最长公共子序列的数目，注意这里的子序列可以不是连续序列，务必问清楚题意。求『最长』类的题目往往与动态规划有点关系，这里是两个字符串，故应为双序列动态规划。

这道题的状态很容易找，不妨先试试以 `f[i][j]` 表示字符串 A 的前 `i` 位和字符串 B 的前 `j` 位的最长公共子序列数目，那么接下来试试寻找其状态转移方程。从实际例子 `ABCD` 和 `EDCA` 出发，首先初始化 `f` 的长度为字符串长度加1，那么有 `f[0][0] = 0`，`f[0][*] = 0`，`f[*][0] = 0`，最后应该返回 `f[lenA][lenB]`. 即 f 中索引与字符串索引对应(字符串索引从1开始算起)，那么在A 的第一个字符与 B 的第一个字符相等时，`f[1][1] = 1 + f[0][0]`，否则 `f[1][1] = max(f[0][1], f[1][0])`。

推而广之，也就意味着若 `A[i] == B[j]`，则分别去掉这两个字符后，原 LCS 数目减一，那为什么一定是1而不是0或者2呢？因为不管公共子序列是以哪个字符结尾，在 `A[i] == B[j]` 时 LCS 最多只能增加1. 而在 `A[i] != B[j]` 时，由于 `A[i]` 或者 `B[j]` 不可能同时出现在最终的 LCS 中，故这个问题可进一步缩小，`f[i][j] = max(f[i - 1][j], f[i][j - 1])`. 需要注意的是这种状态转移方程只依赖最终的 LCS 数目，而不依赖于公共子序列到底是以第几个索引结束。

Python

```python
class Solution:
    """
    @param A, B: Two strings.
```

```
    @return: The length of longest common subsequence of A and B.
    """
    def longestCommonSubsequence(self, A, B):
        if not A or not B:
            return 0

        lenA, lenB = len(A), len(B)
        lcs = [[0 for i in xrange(1 + lenA)] for j in xrange(1 + lenB)]

        for i in xrange(1, 1 + lenA):
            for j in xrange(1, 1 + lenB):
                if A[i - 1] == B[j - 1]:
                    lcs[i][j] = 1 + lcs[i - 1][j - 1]
                else:
                    lcs[i][j] = max(lcs[i - 1][j], lcs[i][j - 1])
        return lcs[lenA][lenB]
```

C++

```
class Solution {
public:
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A and B.
     */
    int longestCommonSubsequence(string A, string B) {
        if (A.empty()) return 0;
        if (B.empty()) return 0;

        int lenA = A.size();
        int lenB = B.size();
        vector<vector<int> > lcs = \
            vector<vector<int> >(1 + lenA, vector<int>(1 + lenB));

        for (int i = 1; i < 1 + lenA; i++) {
            for (int j = 1; j < 1 + lenB; j++) {
                if (A[i - 1] == B[j - 1]) {
                    lcs[i][j] = 1 + lcs[i - 1][j - 1];
                } else {
                    lcs[i][j] = max(lcs[i - 1][j], lcs[i][j - 1]);
                }
            }
        }

        return lcs[lenA][lenB];
    }
};
```

Java

```
 public class Solution {
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A and B.
```

```
     */
    public int longestCommonSubsequence(String A, String B) {
        if (A == null || A.length() == 0) return 0;
        if (B == null || B.length() == 0) return 0;

        int lenA = A.length();
        int lenB = B.length();
        int[][] lcs = new int[1 + lenA][1 + lenB];

        for (int i = 1; i < 1 + lenA; i++) {
            for (int j = 1; j < 1 + lenB; j++) {
                if (A.charAt(i - 1) == B.charAt(j - 1)) {
                    lcs[i][j] = 1 + lcs[i - 1][j - 1];
                } else {
                    lcs[i][j] = Math.max(lcs[i - 1][j], lcs[i][j - 1]);
                }
            }
        }

        return lcs[lenA][lenB];
    }
}
```

源码分析

注意 Python 中的多维数组初始化方式，不可简单使用 `[[0] * len(A)] * len(B)]` , 具体原因是因为 Python 中的对象引用方式 [Stackoverflow]。

复杂度分析

两重for 循环，时间复杂度为 $O(lenA \times lenB)$, 使用了二维数组，空间复杂度也为 $O(lenA \times lenB)$.

Reference

- [Stackoverflow]. Python multi-dimensional array initialization without a loop - Stack Overflow ↩

Edit Distance

- tags: [DP_Two_Sequence]

Source

- leetcode: Edit Distance | LeetCode OJ
- lintcode: (119) Edit Distance

```
Given two words word1 and word2, find the minimum number of steps required
to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

Insert a character
Delete a character
Replace a character
Example
Given word1 = "mart" and word2 = "karma", return 3.
```

题解1 - 双序列动态规划

两个字符串比较，求最值，直接看似乎并不能直接找出解决方案，这时往往需要使用动态规划的思想寻找递推关系。使用双序列动态规划的通用做法，不妨定义 `f[i][j]` 为字符串1的前 `i` 个字符和字符串2的前 `j` 个字符的编辑距离，那么接下来寻找其递推关系。增删操作互为逆操作，即增或者删产生的步数都是一样的。故初始化时容易知道 `f[0][j] = j, f[i][0] = i`，接下来探讨 `f[i][j]` 和 `f[i - 1][j - 1]` 的关系，和 LCS 问题类似，我们分两种情况讨论，即 `word1[i] == word2[j]` 与否，第一种相等的情况有：

1. `i == j`，且有 `word1[i] == word2[j]`，则由 `f[i - 1][j - 1] -> f[i][j]` 不增加任何操作，有 `f[i][j] = f[i - 1][j - 1]`.
2. `i != j`，由于字符数不等，肯定需要增/删一个字符，但是增删 word1 还是 word2 是不知道的，故可取其中编辑距离的较小值，即 `f[i][j] = 1 + min{f[i - 1][j], f[i][j - 1]}`.

第二种不等的情况有：

1. `i == j`，有 `f[i][j] = 1 + f[i - 1][j - 1]`.
2. `i != j`，由于字符数不等，肯定需要增/删一个字符，但是增删 word1 还是 word2 是不知道的，故可取其中编辑距离的较小值，即 `f[i][j] = 1 + min{f[i - 1][j], f[i][j - 1]}`.

最后返回 `f[len(word1)][len(word2)]`

Python

```python
class Solution:
    # @param word1 & word2: Two string.
```

```
    # @return: The minimum number of steps.
    def minDistance(self, word1, word2):
        len1, len2 = 0, 0
        if word1:
            len1 = len(word1)
        if word2:
            len2 = len(word2)
        if not word1 or not word2:
            return max(len1, len2)

        f = [[i + j for i in xrange(1 + len2)] for j in xrange(1 + len1)]

        for i in xrange(1, 1 + len1):
            for j in xrange(1, 1 + len2):
                if word1[i - 1] == word2[j - 1]:
                    f[i][j] = min(f[i - 1][j - 1], 1 + f[i - 1][j], 1 + f[i][j - 1])
                else:
                    f[i][j] = 1 + min(f[i - 1][j - 1], f[i - 1][j], f[i][j - 1])
        return f[len1][len2]
```

C++

```
class Solution {
public:
    /**
     * @param word1 & word2: Two string.
     * @return: The minimum number of steps.
     */
    int fistance(string word1, string word2) {
        if (word1.empty() || word2.empty()) {
            return max(word1.size(), word2.size());
        }

        int len1 = word1.size();
        int len2 = word2.size();
        vector<vector<int> > f = \
            vector<vector<int> >(1 + len1, vector<int>(1 + len2, 0));
        for (int i = 0; i <= len1; ++i) {
            f[i][0] = i;
        }
        for (int i = 0; i <= len2; ++i) {
            f[0][i] = i;
        }

        for (int i = 1; i <= len1; ++i) {
            for (int j = 1; j <= len2; ++j) {
                if (word1[i - 1] == word2[j - 1]) {
                    f[i][j] = min(f[i - 1][j - 1], 1 + f[i - 1][j]);
                    f[i][j] = min(f[i][j], 1 + f[i][j - 1]);
                } else {
                    f[i][j] = min(f[i - 1][j - 1], f[i - 1][j]);
                    f[i][j] = 1 + min(f[i][j], f[i][j - 1]);
                }
            }
        }

        return f[len1][len2];
```

```
    }
};
```

Java

```
public class Solution {
    public int minDistance(String word1, String word2) {
        int len1 = 0, len2 = 0;
        if (word1 != null && word2 != null) {
            len1 = word1.length();
            len2 = word2.length();
        }
        if (word1 == null || word2 == null) {
            return Math.max(len1, len2);
        }

        int[][] f = new int[1 + len1][1 + len2];
        for (int i = 0; i <= len1; i++) {
            f[i][0] = i;
        }
        for (int i = 0; i <= len2; i++) {
            f[0][i] = i;
        }

        for (int i = 1; i <= len1; i++) {
            for (int j = 1; j <= len2; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                    f[i][j] = Math.min(f[i - 1][j - 1], 1 + f[i - 1][j]);
                    f[i][j] = Math.min(f[i][j], 1 + f[i][j - 1]);
                } else {
                    f[i][j] = Math.min(f[i - 1][j - 1], f[i - 1][j]);
                    f[i][j] = 1 + Math.min(f[i][j], f[i][j - 1]);
                }
            }
        }

        return f[len1][len2];
    }
}
```

源码解析

1. 边界处理
2. 初始化二维矩阵(Python 中初始化时 list 中 len2 在前，len1 在后)
3. i, j 从1开始计数，比较 word1 和 word2 时注意下标
4. 返回 `f[len1][len2]`

复杂度分析

两重 for 循环，时间复杂度为 $O(len1 \cdot len2)$. 使用二维矩阵，空间复杂度为 $O(len1 \cdot len2)$.

Jump Game II

Source

- lintcode: (117) Jump Game II

```
Given an array of non-negative integers,
you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

Example
Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2.
(Jump 1 step from index 0 to 1, then 3 steps to the last index.)
```

题解(自顶向下-动态规划)

首先来看看使用动态规划的解法，由于复杂度较高在A元素较多时会出现TLE，因为时间复杂度接近 $O(n^3)$. 工作面试中给出动规的实现就挺好了。

1. State: f[i] 从起点跳到这个位置最少需要多少步
2. Function: f[i] = MIN(f[j]+1, j < i && j + A[j] >= i) 取出所有能从j到i中的最小值
3. Initialization: f[0] = 0，即一个元素时不需移位即可到达
4. Answer: f[n-1]

C++ Dynamic Programming

```cpp
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return -1;
        }

        const int N = A.size() - 1;
        vector<int> steps(N, INT_MAX);
        steps[0] = 0;

        for (int i = 1; i != N + 1; ++i) {
            for (int j = 0; j != i; ++j) {
```

```
                if ((steps[j] != INT_MAX) && (j + A[j] >= i)) {
                    steps[i] = steps[j] + 1;
                    break;
                }
            }
        }

        return steps[N];
    }
};
```

源码分析

状态转移方程为

```
if ((steps[j] != INT_MAX) && (j + A[j] >= i)) {
    steps[i] = steps[j] + 1;
    break;
}
```

其中break即体现了MIN操作，最开始满足条件的j即为最小步数。

题解(贪心法-自底向上)

使用动态规划解Jump Game的题复杂度均较高，这里可以使用贪心法达到线性级别的复杂度。

贪心法可以使用自底向上或者自顶向下，首先看看我最初使用自底向上做的。对A数组遍历，找到最小的下标 `min_index` ，并在下一轮中用此 `min_index` 替代上一次的 `end` , 直至 `min_index` 为0，返回最小跳数 `jumps` 。以下的实现有个 bug，细心的你能发现吗?

C++ greedy from bottom to top, bug version

```
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return -1;
        }

        const int N = A.size() - 1;
        int jumps = 0;
        int last_index = N;
        int min_index = N;

        for (int i = N - 1; i >= 0; --i) {
            if (i + A[i] >= last_index) {
                min_index = i;
```

```
            }

            if (0 == min_index) {
                return ++jumps;
            }

            if ((0 == i) && (min_index < last_index)) {
                ++jumps;
                last_index = min_index;
                i = last_index - 1;
            }
        }

        return jumps;
    }
};
```

源码分析

使用jumps记录最小跳数，last_index记录离终点最远的坐标，min_index记录此次遍历过程中找到的最小下标。

以上的bug在于当min_index为1时，i = 0, for循环中仍有--i，因此退出循环，无法进入 `if (0 == min_index)` 语句，因此返回的结果会小1个。

C++ greedy, from bottom to top

```
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return 0;
        }

        const int N = A.size() - 1;
        int jumps = 0, end = N, min_index = N;

        while (end > 0) {
            for (int i = end - 1; i >= 0; --i) {
                if (i + A[i] >= end) {
                    min_index = i;
                }
            }

            if (min_index < end) {
                ++jumps;
                end = min_index;
            } else {
                // cannot jump to the end
                return -1;
            }
```

```
        }

        return jumps;
    }
};
```

源码分析

之前的 bug version 代码实在是太丑陋了，改写了个相对优雅的实现，加入了是否能到达终点的判断。在更新 `min_index` 的内循环中也可改为如下效率更高的方式：

```
            for (int i = 0; i != end; ++i) {
                if (i + A[i] >= end) {
                    min_index = i;
                    break;
                }
            }
```

题解(贪心法-自顶向下)

看过了自底向上的贪心法，我们再来瞅瞅自顶向下的实现。自顶向下使用 `farthest` 记录当前坐标出发能到达的最远坐标，遍历当前 `start` 与 `end` 之间的坐标，若 `i+A[i] > farthest` 时更新 `farthest` (寻找最小跳数)，当前循环遍历结束时递推 `end = farthest` 。 `end >= A.size() - 1` 时退出循环，返回最小跳数。

C++

```
/**
 * http://www.jiuzhang.com/solutions/jump-game-ii/
 */
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return 0;
        }

        const int N = A.size() - 1;
        int start = 0, end = 0, jumps = 0;

        while (end < N) {
            int farthest = end;
            for (int i = start; i <= end; ++i) {
                if (i + A[i] >= farthest) {
                    farthest = i + A[i];
                }
            }
```

```
            if (end < farthest) {
                ++jumps;
                start = end + 1;
                end = farthest;
            } else {
                // cannot jump to the end
                return -1;
            }
        }

        return jumps;
    }
};
```

Best Time to Buy and Sell Stock

Source

- leetcode: Best Time to Buy and Sell Stock | LeetCode OJ
- lintcode: (149) Best Time to Buy and Sell Stock

```
Say you have an array for
which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction
(ie, buy one and sell one share of the stock),
design an algorithm to find the maximum profit.

Example
Given an example [3,2,3,1,2], return 1
```

题解

最多只允许进行一次交易，显然我们只需要把波谷和波峰分别找出来就好了。但是这样的话问题又来了，有多个波峰和波谷时怎么办？——找出差值最大的一对波谷和波峰。故需要引入一个索引用于记录当前的波谷，结果即为当前索引值减去波谷的值。

Python

```python
class Solution:
    """
    @param prices: Given an integer array
    @return: Maximum profit
    """
    def maxProfit(self, prices):
        if prices is None or len(prices) <= 1:
            return 0

        profit = 0
        cur_price_min = 2**31 - 1
        for price in prices:
            profit = max(profit, price - cur_price_min)
            cur_price_min = min(cur_price_min, price)

        return profit
```

C++

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

```
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    int maxProfit(vector<int> &prices) {
        if (prices.size() <= 1) return 0;

        int profit = 0;
        int cur_price_min = INT_MAX;
        for (int i = 0; i < prices.size(); ++i) {
            profit = max(profit, prices[i] - cur_price_min);
            cur_price_min = min(cur_price_min, prices[i]);
        }

        return profit;
    }
};
```

Java

```
public class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length <= 1) return 0;

        int profit = 0;
        int curPriceMin = Integer.MAX_VALUE;
        for (int price : prices) {
            profit = Math.max(profit, price - curPriceMin);
            curPriceMin = Math.min(curPriceMin, price);
        }

        return profit;
    }
}
```

源码分析

善用 `max` 和 `min` 函数，减少 `if` 的使用。

复杂度分析

遍历一次 prices 数组，时间复杂度为 $O(n)$, 使用了几个额外变量，空间复杂度为 $O(1)$.

Reference

- soulmachine 的卖股票系列

Best Time to Buy and Sell Stock II

Source

- leetcode: Best Time to Buy and Sell Stock II | LeetCode OJ
- lintcode: (150) Best Time to Buy and Sell Stock II

```
Say you have an array for
which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit.
You may complete as many transactions as you like
(ie, buy one and sell one share of the stock multiple times).
However, you may not engage in multiple transactions at the same time
(ie, you must sell the stock before you buy again).

Example
Given an example [2,1,2,0,1], return 2
```

题解

卖股票系列之二，允许进行多次交易，但是不允许同时进行多笔交易。直觉上我们可以找到连续的多对波谷波峰，在波谷买入，波峰卖出，稳赚不赔~ 那么这样是否比只在一个差值最大的波谷波峰处交易赚的多呢？即比上题的方案赚的多。简单的证明可先假设存在一单调上升区间，若人为改变单调区间使得区间内存在不少于一对波谷波峰，那么可以得到进行两次交易的差值之和比单次交易大，证毕。

好了，思路知道了——计算所有连续波谷波峰的差值之和。需要遍历求得所有波谷波峰的值吗？我最开始还真是这么想的，看了 soulmachine 的题解才发现原来可以把数组看成时间序列，只需要计算相邻序列的差值即可，只累加大于0的差值。

Python

```python
class Solution:
    """
    @param prices: Given an integer array
    @return: Maximum profit
    """
    def maxProfit(self, prices):
        if prices is None or len(prices) <= 1:
            return 0

        profit = 0
        for i in xrange(1, len(prices)):
            diff = prices[i] - prices[i - 1]
            if diff > 0:
                profit += diff

        return profit
```

C++

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

        int profit = 0;
        for (int i = 1; i < prices.size(); ++i) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) profit += diff;
        }

        return profit;
    }
};
```

Java

```
class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length <= 1) return 0;

        int profit = 0;
        for (int i = 1; i < prices.length; i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) profit += diff;
        }

        return profit;
    }
};
```

源码分析

核心在于将多个波谷波峰的差值之和的计算转化为相邻序列的差值，故 i 从1开始算起。

复杂度分析

遍历一次原数组，时间复杂度为 $O(n)$, 用了几个额外变量，空间复杂度为 $O(1)$.

Reference

- soulmachine 的卖股票系列

Best Time to Buy and Sell Stock III

Source

- leetcode: Best Time to Buy and Sell Stock III | LeetCode OJ
- lintcode: (151) Best Time to Buy and Sell Stock III

```
Say you have an array for
which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit.
You may complete at most two transactions.

Example
Given an example [4,4,6,1,1,4,2,5], return 6.

Note
You may not engage in multiple transactions at the same time
(ie, you must sell the stock before you buy again).
```

题解

与前两道允许一次或者多次交易不同，这里只允许最多两次交易，且这两次交易不能交叉。咋一看似乎无从下手，我最开始想到的是找到排在前2个的波谷波峰，计算这两个差值之和。原理上来讲应该是可行的，但是需要记录 $O(n^2)$ 个波谷波峰并对其排序，实现起来也比较繁琐。

除了以上这种直接分析问题的方法外，是否还可以借助分治的思想解决呢？最多允许两次不相交的交易，也就意味着这两次交易间存在某一分界线，考虑到可只交易一次，也可交易零次，故分界线的变化范围为第一天至最后一天，只需考虑分界线两边各自的最大利润，最后选出利润和最大的即可。

这种方法抽象之后则为首先将 [1,n] 拆分为 [1,i] 和 [i+1,n], 参考卖股票系列的第一题计算各自区间内的最大利润即可。[1,i] 区间的最大利润很好算，但是如何计算 [i+1,n] 区间的最大利润值呢？难道需要重复 n 次才能得到？注意到区间的右侧 n 是个不变值，我们从 [1, i] 计算最大利润是更新波谷的值，那么我们可否逆序计算最大利润呢？这时候就需要更新记录波峰的值了。逆向思维大法好！Talk is cheap, show me the code!

Python

```python
class Solution:
    """
    @param prices: Given an integer array
    @return: Maximum profit
    """
    def maxProfit(self, prices):
        if prices is None or len(prices) <= 1:
            return 0

        n = len(prices)
```

```
        # get profit in the front of prices
        profit_front = [0] * n
        valley = prices[0]
        for i in xrange(1, n):
            profit_front[i] = max(profit_front[i - 1], prices[i] - valley)
            valley = min(valley, prices[i])
        # get profit in the back of prices, (i, n)
        profit_back = [0] * n
        peak = prices[-1]
        for i in xrange(n - 2, -1, -1):
            profit_back[i] = max(profit_back[i + 1], peak - prices[i])
            peak = max(peak, prices[i])
        # add the profit front and back
        profit = 0
        for i in xrange(n):
            profit = max(profit, profit_front[i] + profit_back[i])

        return profit
```

C++

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

        int n = prices.size();
        // get profit in the front of prices
        vector<int> profit_front = vector<int>(n, 0);
        for (int i = 1, valley = prices[0]; i < n; ++i) {
            profit_front[i] = max(profit_front[i - 1], prices[i] - valley);
            valley = min(valley, prices[i]);
        }
        // get profit in the back of prices, (i, n)
        vector<int> profit_back = vector<int>(n, 0);
        for (int i = n - 2, peak = prices[n - 1]; i >= 0; --i) {
            profit_back[i] = max(profit_back[i + 1], peak - prices[i]);
            peak = max(peak, prices[i]);
        }
        // add the profit front and back
        int profit = 0;
        for (int i = 0; i < n; ++i) {
            profit = max(profit, profit_front[i] + profit_back[i]);
        }

        return profit;
    }
};
```

Java

```java
class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length <= 1) return 0;

        // get profit in the front of prices
        int[] profitFront = new int[prices.length];
        profitFront[0] = 0;
        for (int i = 1, valley = prices[0]; i < prices.length; i++) {
            profitFront[i] = Math.max(profitFront[i - 1], prices[i] - valley);
            valley = Math.min(valley, prices[i]);
        }
        // get profit in the back of prices, (i, n)
        int[] profitBack = new int[prices.length];
        profitBack[prices.length - 1] = 0;
        for (int i = prices.length - 2, peak = prices[prices.length - 1]; i >= 0; i--) {
            profitBack[i] = Math.max(profitBack[i + 1], peak - prices[i]);
            peak = Math.max(peak, prices[i]);
        }
        // add the profit front and back
        int profit = 0;
        for (int i = 0; i < prices.length; i++) {
            profit = Math.max(profit, profitFront[i] + profitBack[i]);
        }

        return profit;
    }
};
```

源码分析

整体分为三大部分，计算前半部分的最大利润值，然后计算后半部分的最大利润值，最后遍历得到最终的最大利润值。

复杂度分析

三次遍历原数组，时间复杂度为 $O(n)$, 利用了若干和数组等长的数组，空间复杂度也为 $O(n)$.

Reference

- soulmachine 的卖股票系列

Best Time to Buy and Sell Stock IV

Source

- leetcode: Best Time to Buy and Sell Stock IV | LeetCode OJ
- lintcode: (393) Best Time to Buy and Sell Stock IV

```
Say you have an array for
which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit.
You may complete at most k transactions.

Example
Given prices = [4,4,6,1,1,4,2,5], and k = 2, return 6.

Note
You may not engage in multiple transactions at the same time
(i.e., you must sell the stock before you buy again).

Challenge
O(nk) time.
```

题解1

卖股票系列中最难的一道，较易实现的方法为使用动态规划，动规的实现又分为大约3大类方法，这里先介绍一种最为朴素的方法，过不了大量数据，会 TLE.

最多允许 k 次交易，由于一次增加收益的交易至少需要两天，故当 k >= n/2时，此题退化为卖股票的第二道题，即允许任意多次交易。当 k < n/2 时，使用动规来求解，动规的几个要素如下：

f[i][j] 代表第 i 天为止交易 k 次获得的最大收益，那么将问题分解为前 x 天交易 k-1 次，第 x+1 天至第 i 天交易一次两个子问题，于是动态方程如下：

```
f[i][j] = max(f[x][j - 1] + profit(x + 1, i))
```

简便起见，初始化二维矩阵为0，下标尽可能从1开始，便于理解。

Python

```python
class Solution:
    """
    @param k: an integer
    @param prices: a list of integer
    @return: an integer which is maximum profit
    """
```

```
    def maxProfit(self, k, prices):
        if prices is None or len(prices) <= 1 or k <= 0:
            return 0

        n = len(prices)
        # k >= prices.length / 2 ==> multiple transactions Stock II
        if k >= n / 2:
            profit_max = 0
            for i in xrange(1, n):
                diff = prices[i] - prices[i - 1]
                if diff > 0:
                    profit_max += diff
            return profit_max

        f = [[0 for i in xrange(k + 1)] for j in xrange(n + 1)]
        for j in xrange(1, k + 1):
            for i in xrange(1, n + 1):
                for x in xrange(0, i + 1):
                    f[i][j] = max(f[i][j], f[x][j - 1] + self.profit(prices, x + 1, i))

        return f[n][k]

    # calculate the profit of prices(l, u)
    def profit(self, prices, l, u):
        if l >= u:
            return 0
        valley = 2**31 - 1
        profit_max = 0
        for price in prices[l - 1:u]:
            profit_max = max(profit_max, price - valley)
            valley = min(valley, price)
        return profit_max
```

C++

```
class Solution {
public:
    /**
     * @param k: An integer
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    int maxProfit(int k, vector<int> &prices) {
        if (prices.size() <= 1 || k <= 0) return 0;

        int n = prices.size();
        // k >= prices.length / 2 ==> multiple transactions Stock II
        if (k >= n / 2) {
            int profit_max = 0;
            for (int i = 1; i < n; ++i) {
                int diff = prices[i] - prices[i - 1];
                if (diff > 0) {
                    profit_max += diff;
                }
            }
            return profit_max;
```

```
        }

        vector<vector<int> > f = vector<vector<int> >(n + 1, vector<int>(k + 1, 0));
        for (int j = 1; j <= k; ++j) {
            for (int i = 1; i <= n; ++i) {
                for (int x = 0; x <= i; ++x) {
                    f[i][j] = max(f[i][j], f[x][j - 1] + profit(prices, x + 1, i));
                }
            }
        }

        return f[n][k];
    }

private:
    int profit(vector<int> &prices, int l, int u) {
        if (l >= u) return 0;

        int valley = INT_MAX;
        int profit_max = 0;
        for (int i = l - 1; i < u; ++i) {
            profit_max = max(profit_max, prices[i] - valley);
            valley = min(valley, prices[i]);
        }

        return profit_max;
    }
};
```

Java

```
class Solution {
    /**
     * @param k: An integer
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int k, int[] prices) {
        if (prices == null || prices.length <= 1 || k <= 0) return 0;

        int n = prices.length;
        if (k >= n / 2) {
            int profit_max = 0;
            for (int i = 1; i < n; i++) {
                if (prices[i] - prices[i - 1] > 0) {
                    profit_max += prices[i] - prices[i - 1];
                }
            }
            return profit_max;
        }

        int[][] f = new int[n + 1][k + 1];
        for (int j = 1; j <= k; j++) {
            for (int i = 1; i <= n; i++) {
                for (int x = 0; x <= i; x++) {
                    f[i][j] = Math.max(f[i][j], f[x][j - 1] + profit(prices, x + 1, i));
                }
```

```
            }
        }

        return f[n][k];
    }

    private int profit(int[] prices, int l, int u) {
        if (l >= u) return 0;

        int valley = Integer.MAX_VALUE;
        int profit_max = 0;
        for (int i = l - 1; i < u; i++) {
            profit_max = Math.max(profit_max, prices[i] - valley);
            valley = Math.min(valley, prices[i]);
        }
        return profit_max;
    }
};
```

源码分析

注意 Python 中的多维数组初始化方式，不可简单使用 `[[0] * k] * n]`，具体原因是因为 Python 中的对象引用方式。可以优化的地方是 profit 方法及最内存循环。

复杂度分析

三重循环，时间复杂度近似为 $O(n^2 \cdot k)$, 使用了 f 二维数组，空间复杂度为 $O(n \cdot k)$.

Reference

- [LeetCode] Best Time to Buy and Sell Stock I II III IV | 梁佳宾的网络日志
- Best Time to Buy and Sell Stock IV 参考程序 Java/C++/Python
- leetcode-Best Time to Buy and Sell Stock 系列 // 陈辉的技术博客
- [LeetCode]Best Time to Buy and Sell Stock IV | 书影博客

Distinct Subsequences

Source

- leetcode: Distinct Subsequences | LeetCode OJ
- lintcode: (118) Distinct Subsequences

```
Given a string S and a string T, count the number of distinct subsequences of T in S.
A subsequence of a string is a new string
which is formed from the original string by deleting some (can be none) of the characters
without disturbing the relative positions of the remaining characters.
(ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Example
Given S = "rabbbit", T = "rabbit", return 3.
Challenge
Do it in O(n2) time and O(n) memory.

O(n2) memory is also acceptable if you do not know how to optimize memory.
```

题解1

首先分清 subsequence 和 substring 两者的区别，subsequence 可以是不连续的子串。题意要求 S 中子序列 T 的个数。如果不考虑程序实现，我们能想到的办法是逐个比较 S 和 T 的首字符，相等的字符删掉，不等时则删除 S 中的首字符，继续比较后续字符直至 T 中字符串被删完。这种简单的思路有这么几个问题，题目问的是子序列的个数，而不是是否存在，故在字符不等时不能轻易删除掉 S 中的字符。那么如何才能得知子序列的个数呢？

要想得知不同子序列的个数，那么我们就不能在 S 和 T 中首字符不等时简单移除 S 中的首字符了，取而代之的方法应该是先将 S 复制一份，再用移除 S 中首字符后的新字符串和 T 进行比较，这点和深搜中的剪枝函数的处理有点类似。

Python

```python
class Solution:
    # @param S, T: Two string.
    # @return: Count the number of distinct subsequences
    def numDistinct(self, S, T):
        if S is None or T is None:
            return 0
        if len(S) < len(T):
            return 0
        if len(T) == 0:
            return 1

        num = 0
        for i, Si in enumerate(S):
```

```
            if Si == T[0]:
                num += self.numDistinct(S[i + 1:], T[1:])

        return num
```

C++

```
class Solution {
public:
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    int numDistinct(string &S, string &T) {
        if (S.size() < T.size()) return 0;
        if (T.empty()) return 1;

        int num = 0;
        for (int i = 0; i < S.size(); ++i) {
            if (S[i] == T[0]) {
                string Si = S.substr(i + 1);
                string t = T.substr(1);
                num += numDistinct(Si, t);
            }
        }

        return num;
    }
};
```

Java

```
public class Solution {
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        if (S == null || T == null) return 0;
        if (S.length() < T.length()) return 0;
        if (T.length() == 0) return 1;

        int num = 0;
        for (int i = 0; i < S.length(); i++) {
            if (S.charAt(i) == T.charAt(0)) {
                // T.length() >= 1, T.substring(1) will not throw index error
                num += numDistinct(S.substring(i + 1), T.substring(1));
            }
        }

        return num;
    }
}
```

源码分析

1. 对 null 异常处理(C++ 中对 string 赋NULL 是错的，函数内部无法 handle 这种情况)
2. S 字符串长度若小于 T 字符串长度，T 必然不是 S 的子序列，返回0
3. T 字符串长度为0，证明 T 是 S 的子序列，返回1

由于进入 for 循环的前提是 `T.length() >= 1`, 故当 T 的长度为1时，Java 中对 T 取子串 `T.substring(1)` 时产生的是空串 `""` 而并不抛出索引越界的异常。

复杂度分析

最好情况下，S 中没有和 T 相同的字符，时间复杂度为 $O(n)$; 最坏情况下，S 中的字符和 T 中字符完全相同，此时可以画出递归调用栈，发现和深搜非常类似，数学关系为 $f(n) = \sum_{i=1}^{n-1} f(i)$, 这比 Fibonacci 的复杂度还要高很多。

题解2 - Dynamic Programming

从题解1 的复杂度分析中我们能发现由于存在较多的重叠子状态(相同子串被比较多次), 因此可以想到使用动态规划优化。但是动规的三大要素如何建立？由于本题为两个字符串之间的关系，故可以尝试使用双序列(DP_Two_Sequence)动规的思路求解。

定义 `f[i][j]` 为 S[0:i] 中子序列为 T[0:j] 的个数，接下来寻找状态转移关系，状态转移应从 f[i-1][j], f[i-1][j-1], f[i][j-1] 中寻找，接着寻找突破口——S[i] 和 T[j] 的关系。

1. `S[i] == T[j]` : 两个字符串的最后一个字符相等，我们可以选择 S[i] 和 T[j] 配对，那么此时有 f[i][j] = f[i-1][j-1]; 若不使 S[i] 和 T[j] 配对，而是选择 S[0:i-1] 中的某个字符和 T[j] 配对，那么 f[i][j] = f[i-1][j]. 综合以上两种选择，可得知在 `S[i] == T[j]` 时有 f[i][j] = f[i-1][j-1] + f[i-1][j]
2. `S[i] != T[j]` : 最后一个字符不等时，S[i] 不可能和 T[j] 配对，故 f[i][j] = f[i-1][j]

为便于处理第一个字符相等的状态(便于累加)，初始化f[i][0]为1, 其余为0. 这里对于 S 或 T 为空串时返回0，返回1 也能说得过去。

Python

```
class Solution:
    # @param S, T: Two string.
    # @return: Count the number of distinct subsequences
    def numDistinct(self, S, T):
        if S is None or T is None:
            return 0
        if len(S) < len(T):
            return 0
        if len(T) == 0:
            return 1

        f = [[0 for i in xrange(len(T) + 1)] for j in xrange(len(S) + 1)]
        for i, Si in enumerate(S):
            f[i][0] = 1
            for j, Tj in enumerate(T):
```

```
                if Si == Tj:
                    f[i + 1][j + 1] = f[i][j + 1] + f[i][j]
                else:
                    f[i + 1][j + 1] = f[i][j + 1]

        return f[len(S)][len(T)]
```

C++

```
class Solution {
public:
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    int numDistinct(string &S, string &T) {
        if (S.size() < T.size()) return 0;
        if (T.empty()) return 1;

        vector<vector<int> > f(S.size() + 1, vector<int>(T.size() + 1, 0));
        for (int i = 0; i < S.size(); ++i) {
            f[i][0] = 1;
            for (int j = 0; j < T.size(); ++j) {
                if (S[i] == T[j]) {
                    f[i + 1][j + 1] = f[i][j + 1] + f[i][j];
                } else {
                    f[i + 1][j + 1] = f[i][j + 1];
                }
            }
        }

        return f[S.size()][T.size()];
    }
};
```

Java

```
public class Solution {
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        if (S == null || T == null) return 0;
        if (S.length() < T.length()) return 0;
        if (T.length() == 0) return 1;

        int[][] f = new int[S.length() + 1][T.length() + 1];
        for (int i = 0; i < S.length(); i++) {
            f[i][0] = 1;
            for (int j = 0; j < T.length(); j++) {
                if (S.charAt(i) == T.charAt(j)) {
                    f[i + 1][j + 1] = f[i][j + 1] + f[i][j];
                } else {
```

```
                    f[i + 1][j + 1] = f[i][j + 1];
                }
            }
        }

        return f[S.length()][T.length()];
    }
}
```

源码分析

异常处理部分和题解1 相同，初始化时维度均多一个元素便于处理。

复杂度分析

由于免去了重叠子状态的计算，双重 for 循环，时间复杂度为 $O(n^2)$, 使用了二维矩阵保存状态，空间复杂度为 $O(n^2)$. 空间复杂度可以通过滚动数组的方式优化，详见 Dynamic Programming - 动态规划.

空间复杂度优化之后的代码如下：

Java

```java
public class Solution {
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        if (S == null || T == null) return 0;
        if (S.length() < T.length()) return 0;
        if (T.length() == 0) return 1;

        int[] f = new int[T.length() + 1];
        f[0] = 1;
        for (int i = 0; i < S.length(); i++) {
            for (int j = T.length() - 1; j >= 0; j--) {
                if (S.charAt(i) == T.charAt(j)) {
                        f[j + 1] += f[j];
                }
            }
        }

        return f[T.length()];
    }
}
```

Reference

- LeetCode: Distinct Subsequences（不同子序列的个数） - 亦忘却_亦纪念
- soulmachine leetcode-cpp 中 Distinct Subsequences 部分
- Distinct Subsequences | Training dragons the hard way

Interleaving String

Source

- leetcode: Interleaving String | LeetCode OJ
- lintcode: (29) Interleaving String

```
Given three strings: s1, s2, s3,
determine whether s3 is formed by the interleaving of s1 and s2.

Example
For s1 = "aabcc", s2 = "dbbca"

When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.
Challenge
O(n2) time or better
```

题解1 - bug

题目意思是 s3 是否由 s1 和 s2 交叉构成，不允许跳着从 s1 和 s2 挑选字符。那么直觉上可以对三个字符串设置三个索引，首先从 s3 中依次取字符，然后进入内循环，依次从 s1 和 s2 中取首字符，若能匹配上则进入下一次循环，否则立即返回 false. 我们先看代码，再分析 bug 之处。

Java

```java
public class Solution {
    /**
     * Determine whether s3 is formed by interleaving of s1 and s2.
     * @param s1, s2, s3: As description.
     * @return: true or false.
     */
    public boolean isInterleave(String s1, String s2, String s3) {
        int len1 = (s1 == null) ? 0 : s1.length();
        int len2 = (s2 == null) ? 0 : s2.length();
        int len3 = (s3 == null) ? 0 : s3.length();

        if (len3 != len1 + len2) return false;

        int i1 = 0, i2 = 0;
        for (int i3 = 0; i3 < len3; i3++) {
            boolean result = false;
            if (i1 < len1 && s1.charAt(i1) == s3.charAt(i3)) {
                i1++;
                result = true;
                continue;
            }
            if (i2 < len2 && s2.charAt(i2) == s3.charAt(i3)) {
                i2++;
```

```
                result = true;
                continue;
            }

            // return instantly if both s1 and s2 can not pair with s3
            if (!result) return false;
        }

        return true;
    }
}
```

源码分析

异常处理部分：首先求得 s1, s2, s3 的字符串长度，随后用索引 i1, i2, i3 巧妙地避开了繁琐的 null 检测。这段代码能过前面的一部分数据，但在 lintcode 的第15个 test 跪了。不想马上看以下分析的可以自己先 debug 下。

我们可以注意到以上代码还有一种情况并未考虑到，那就是当 s1[i1] 和 s2[i2] 均和 s3[i3] 相等时，我们可以拿 s1 或者 s2 去匹配，那么问题来了，由于不允许跳着取，那么可能出现在取了 s1 中的字符后，接下来的 s1 和 s2 首字符都无法和 s3 匹配到，因此原本应该返回 true 而现在返回 false. 建议将以上代码贴到 OJ 上看看测试用例。

以上 bug 可以通过加入对 `(s1[i1] == s3[i3]) && (s2[i2] == s3[i3])` 这一特殊情形考虑，即分两种情况递归调用 isInterleave, 只不过 s1, s2, s3 为新生成的字符串。

复杂度分析

遍历一次 s3, 时间复杂度为 $O(n)$, 空间复杂度 $O(1)$.

题解2

在 `(s1[i1] == s3[i3]) && (s2[i2] == s3[i3])` 时分两种情况考虑，即让 s1[i1] 和 s3[i3] 配对或者 s2[i2] 和 s3[i3] 配对，那么嵌套调用时新生成的字符串则分别为 `s1[1+i1:], s2[i2], s3[1+i3:]` 和 `s1[i1:], s2[1+i2], s3[1+i3:]`. 嵌套调用结束后立即返回最终结果，因为递归调用时整个结果已经知晓，不立即返回则有可能会产生错误结果，递归调用并未影响到调用处的 i1 和 i2.

Python

```
class Solution:
    """
    @params s1, s2, s3: Three strings as description.
    @return: return True if s3 is formed by the interleaving of
             s1 and s2 or False if not.
    @hint: you can use [[True] * m for i in range (n)] to allocate a n*m matrix.
    """
    def isInterleave(self, s1, s2, s3):
        len1 = 0 if s1 is None else len(s1)
        len2 = 0 if s2 is None else len(s2)
        len3 = 0 if s3 is None else len(s3)
```

```
        if len3 != len1 + len2:
            return False

        i1, i2 = 0, 0
        for i3 in xrange(len(s3)):
            result = False
            if (i1 < len1 and s1[i1] == s3[i3]) and \
               (i1 < len1 and s1[i1] == s3[i3]):
                # s1[1+i1:], s2[i2:], s3[1+i3:]
                case1 = self.isInterleave(s1[1 + i1:], s2[i2:], s3[1 + i3:])
                # s1[i1:], s2[1+i2:], s3[1+i3:]
                case2 = self.isInterleave(s1[i1:], s2[1 + i2:], s3[1 + i3:])
                return case1 or case2

            if i1 < len1 and s1[i1] == s3[i3]:
                i1 += 1
                result = True
                continue

            if i2 < len2 and s2[i2] == s3[i3]:
                i2 += 1
                result = True
                continue

            # return instantly if both s1 and s2 can not pair with s3
            if not result:
                return False

        return True
```

C++

```
class Solution {
public:
    /**
     * Determine whether s3 is formed by interleaving of s1 and s2.
     * @param s1, s2, s3: As description.
     * @return: true of false.
     */
    bool isInterleave(string s1, string s2, string s3) {
        int len1 = s1.size();
        int len2 = s2.size();
        int len3 = s3.size();

        if (len3 != len1 + len2) return false;

        int i1 = 0, i2 = 0;
        for (int i3 = 0; i3 < len3; ++i3) {
            bool result = false;
            if (i1 < len1 && s1[i1] == s3[i3] &&
                i2 < len2 && s2[i2] == s3[i3]) {
                // s1[1+i1:], s2[i2:], s3[1+i3:]
                bool case1 = isInterleave(s1.substr(1 + i1), s2.substr(i2), s3.substr(1 +
                // s1[i1:], s2[1+i2:], s3[1+i3:]
                bool case2 = isInterleave(s1.substr(i1), s2.substr(1 + i2), s3.substr(1 +
                // return instantly
```

```
                return case1 || case2;
            }

            if (i1 < len1 && s1[i1] == s3[i3]) {
                i1++;
                result = true;
                continue;
            }

            if (i2 < len2 && s2[i2] == s3[i3]) {
                i2++;
                result = true;
                continue;
            }

            // return instantly if both s1 and s2 can not pair with s3
            if (!result) return false;
        }

        return true;
    }
};
```

Java

```
public class Solution {
    /**
     * Determine whether s3 is formed by interleaving of s1 and s2.
     * @param s1, s2, s3: As description.
     * @return: true or false.
     */
    public boolean isInterleave(String s1, String s2, String s3) {
        int len1 = (s1 == null) ? 0 : s1.length();
        int len2 = (s2 == null) ? 0 : s2.length();
        int len3 = (s3 == null) ? 0 : s3.length();

        if (len3 != len1 + len2) return false;

        int i1 = 0, i2 = 0;
        for (int i3 = 0; i3 < len3; i3++) {
            boolean result = false;
            if (i1 < len1 && s1.charAt(i1) == s3.charAt(i3) &&
                i2 < len2 && s2.charAt(i2) == s3.charAt(i3)) {
                // s1[1+i1:], s2[i2:], s3[1+i3:]
                boolean case1 = isInterleave(s1.substring(1 + i1), s2.substring(i2), s3.s
                // s1[i1:], s2[1+i2:], s3[1+i3:]
                boolean case2 = isInterleave(s1.substring(i1), s2.substring(1 + i2), s3.s
                // return instantly
                return case1 || case2;
            }

            if (i1 < len1 && s1.charAt(i1) == s3.charAt(i3)) {
                i1++;
                result = true;
                continue;
            }
```

```
            if (i2 < len2 && s2.charAt(i2) == s3.charAt(i3)) {
                i2++;
                result = true;
                continue;
            }

            // return instantly if both s1 and s2 can not pair with s3
            if (!result) return false;
        }

        return true;
    }
}
```

题解3 - 动态规划

看过题解1 和 题解2 的思路后动规的状态和状态方程应该就不难推出了。按照经典的序列规划，不妨假设状态 f[i1][i2][i3] 为 s1的前i1个字符和 s2的前 i2个字符是否能交叉构成 s3的前 i3个字符，那么根据 s1[i1], s2[i2], s3[i3]的匹配情况可以分为8种情况讨论。咋一看这似乎十分麻烦，但实际上我们注意到其实还有一个隐含条件：`len3 == len1 + len2`, 故状态转移方程得到大幅简化。

新的状态可定义为 f[i1][i2], 含义为s1的前 `i1` 个字符和 s2的前 `i2` 个字符是否能交叉构成 s3的前 `i1 + i2` 个字符。根据 `s1[i1] == s3[i3]` 和 `s2[i2] == s3[i3]` 的匹配情况可建立状态转移方程为：

```
f[i1][i2] = (s1[i1 - 1] == s3[i1 + i2 - 1] && f[i1 - 1][i2]) ||
            (s2[i2 - 1] == s3[i1 + i2 - 1] && f[i1][i2 - 1])
```

这道题的初始化有点 trick, 考虑到空串的可能，需要单独初始化 `f[*][0]` 和 `f[0][*]`.

Python

```
class Solution:
    """
    @params s1, s2, s3: Three strings as description.
    @return: return True if s3 is formed by the interleaving of
             s1 and s2 or False if not.
    @hint: you can use [[True] * m for i in range (n)] to allocate a n*m matrix.
    """
    def isInterleave(self, s1, s2, s3):
        len1 = 0 if s1 is None else len(s1)
        len2 = 0 if s2 is None else len(s2)
        len3 = 0 if s3 is None else len(s3)

        if len3 != len1 + len2:
            return False

        f = [[True] * (1 + len2) for i in xrange (1 + len1)]
        # s1[i1 - 1] == s3[i1 + i2 - 1] && f[i1 - 1][i2]
        for i in xrange(1, 1 + len1):
```

```
            f[i][0] = s1[i - 1] == s3[i - 1] and f[i - 1][0]
        # s2[i2 - 1] == s3[i1 + i2 - 1] && f[i1][i2 - 1]
        for i in xrange(1, 1 + len2):
            f[0][i] = s2[i - 1] == s3[i - 1] and f[0][i - 1]
        # i1 >= 1, i2 >= 1
        for i1 in xrange(1, 1 + len1):
            for i2 in xrange(1, 1 + len2):
                case1 = s1[i1 - 1] == s3[i1 + i2 - 1] and f[i1 - 1][i2]
                case2 = s2[i2 - 1] == s3[i1 + i2 - 1] and f[i1][i2 - 1]
                f[i1][i2] = case1 or case2

        return f[len1][len2]
```

C++

```
class Solution {
public:
    /**
     * Determine whether s3 is formed by interleaving of s1 and s2.
     * @param s1, s2, s3: As description.
     * @return: true of false.
     */
    bool isInterleave(string s1, string s2, string s3) {
        int len1 = s1.size();
        int len2 = s2.size();
        int len3 = s3.size();

        if (len3 != len1 + len2) return false;

        vector<vector<bool> > f(1 + len1, vector<bool>(1 + len2, true));
        // s1[i1 - 1] == s3[i1 + i2 - 1] && f[i1 - 1][i2]
        for (int i = 1; i <= len1; ++i) {
            f[i][0] = s1[i - 1] == s3[i - 1] && f[i - 1][0];
        }
        // s2[i2 - 1] == s3[i1 + i2 - 1] && f[i1][i2 - 1]
        for (int i = 1; i <= len2; ++i) {
            f[0][i] = s2[i - 1] == s3[i - 1] && f[0][i - 1];
        }
        // i1 >= 1, i2 >= 1
        for (int i1 = 1; i1 <= len1; ++i1) {
            for (int i2 = 1; i2 <= len2; ++i2) {
                bool case1 = s1[i1 - 1] == s3[i1 + i2 - 1] && f[i1 - 1][i2];
                bool case2 = s2[i2 - 1] == s3[i1 + i2 - 1] && f[i1][i2 - 1];
                f[i1][i2] = case1 || case2;
            }
        }

        return f[len1][len2];
    }
};
```

Java

```
public class Solution {
```

```java
    /**
     * Determine whether s3 is formed by interleaving of s1 and s2.
     * @param s1, s2, s3: As description.
     * @return: true or false.
     */
    public boolean isInterleave(String s1, String s2, String s3) {
        int len1 = (s1 == null) ? 0 : s1.length();
        int len2 = (s2 == null) ? 0 : s2.length();
        int len3 = (s3 == null) ? 0 : s3.length();

        if (len3 != len1 + len2) return false;

        boolean [][] f = new boolean[1 + len1][1 + len2];
        f[0][0] = true;
        // s1[i1 - 1] == s3[i1 + i2 - 1] && f[i1 - 1][i2]
        for (int i = 1; i <= len1; i++) {
            f[i][0] = s1.charAt(i - 1) == s3.charAt(i - 1) && f[i - 1][0];
        }
        // s2[i2 - 1] == s3[i1 + i2 - 1] && f[i1][i2 - 1]
        for (int i = 1; i <= len2; i++) {
            f[0][i] = s2.charAt(i - 1) == s3.charAt(i - 1) && f[0][i - 1];
        }
        // i1 >= 1, i2 >= 1
        for (int i1 = 1; i1 <= len1; i1++) {
            for (int i2 = 1; i2 <= len2; i2++) {
                boolean case1 = s1.charAt(i1 - 1) == s3.charAt(i1 + i2 - 1) && f[i1 - 1][
                boolean case2 = s2.charAt(i2 - 1) == s3.charAt(i1 + i2 - 1) && f[i1][i2 -
                f[i1][i2] = case1 || case2;
            }
        }

        return f[len1][len2];
    }
}
```

源码分析

为后面递推方便，初始化时数组长度多加1，for 循环时需要注意边界(取到等号)。

复杂度分析

双重 for 循环，时间复杂度为 $O(n^2)$, 使用了二维矩阵，空间复杂度 $O(n^2)$. 其中空间复杂度可以优化。

Reference

- soulmachine 的 Interleaving String 部分
- Interleaving String 参考程序 Java/C++/Python

Maximum Subarray

Source

- leetcode: Maximum Subarray | LeetCode OJ
- lintcode: (41) Maximum Subarray

```
Given an array of integers,
find a contiguous subarray which has the largest sum.

Example
Given the array [-2,2,-3,4,-1,2,1,-5,3],
the contiguous subarray [4,-1,2,1] has the largest sum = 6.

Note
The subarray should contain at least one number.

Challenge
Can you do it in time complexity O(n)?
```

题解1 - 贪心

求最大子数组和，即求区间和的最大值，不同子区间共有约 n^2 中可能，遍历虽然可解，但是时间复杂度颇高。

这里首先介绍一种巧妙的贪心算法，用 `sum` 表示当前子数组和， `maxSum` 表示求得的最大子数组和。当 `sum <= 0` 时，累加数组中的元素只会使得到的和更小，故此时应将此部分和丢弃，使用此时遍历到的数组元素替代。需要注意的是由于有 `maxSum` 更新 `sum` , 故直接丢弃小于0的 `sum` 并不会对最终结果有影响。即不会漏掉前面的和比后面的元素大的情况。

Java

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(ArrayList<Integer> nums) {
        // -1 is not proper for illegal input
        if (nums == null || nums.isEmpty()) return -1;

        int sum = 0, maxSub = Integer.MIN_VALUE;
        for (int num : nums) {
            // drop negtive sum
            sum = Math.max(sum, 0);
            sum += num;
            // update maxSub
            maxSub = Math.max(maxSub, sum);
```

```
        }

        return maxSub;
    }
}
```

源码分析

贪心的实现较为巧妙，需要 `sum` 和 `maxSub` 配合运作才能正常工作。

复杂度分析

遍历一次数组，时间复杂度 $O(n)$, 使用了几个额外变量，空间复杂度 $O(1)$.

题解2 - 动态规划1(区间和)

求最大/最小这种字眼往往都可以使用动态规划求解，此题为单序列动态规划。我们可以先求出到索引 i 的子数组和，然后用子数组和的最大值减去最小值，最后返回最大值即可。用这种动态规划需要注意初始化条件和求和顺序。

Java

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(ArrayList<Integer> nums) {
        // -1 is not proper for illegal input
        if (nums == null || nums.isEmpty()) return -1;

        int sum = 0, minSum = 0, maxSub = Integer.MIN_VALUE;
        for (int num : nums) {
            minSum = Math.min(minSum, sum);
            sum += num;
            maxSub = Math.max(maxSub, sum - minSum);
        }

        return maxSub;
    }
}
```

源码分析

首先求得当前的最小子数组和，初始化为0，随后比较子数组和减掉最小子数组和的差值和最大区间和，并更新最大区间和。

复杂度分析

时间复杂度 $O(n)$, 使用了类似滚动数组的处理方式，空间复杂度 $O(1)$.

题解3 - 动态规划2(局部与全局)

这种动规的实现和题解1 的思想几乎一模一样，只不过这里用局部最大值和全局最大值两个数组来表示。

Java

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(ArrayList<Integer> nums) {
        // -1 is not proper for illegal input
        if (nums == null || nums.isEmpty()) return -1;

        int size = nums.size();
        int[] local = new int[size];
        int[] global = new int[size];
        local[0] = nums.get(0);
        global[0] = nums.get(0);
        for (int i = 1; i < size; i++) {
            // drop local[i - 1] < 0
            local[i] = Math.max(nums.get(i), local[i - 1] + nums.get(i));
            // update global with local
            global[i] = Math.max(global[i - 1], local[i]);
        }

        return global[size - 1];
    }
}
```

源码分析

由于局部最大值需要根据之前的局部值是否大于0进行更新，故方便起见初始化 local 和 global 数组的第一个元素为数组第一个元素。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度也为 $O(n)$.

Reference

- 《剑指 Offer》第五章
- Maximum Subarray 参考程序 Java/C++/Python

Maximum Subarray II

Source

- lintcode: (42) Maximum Subarray II

```
Given an array of integers,
find two non-overlapping subarrays which have the largest sum.

The number in each subarray should be contiguous.

Return the largest sum.

Example
For given [1, 3, -1, 2, -1, 2],
the two subarrays are [1, 3] and [2, -1, 2] or [1, 3, -1, 2] and [2],
they both have the largest sum 7.

Note
The subarray should contain at least one number

Challenge
Can you do it in time complexity O(n) ?
```

题解

严格来讲这道题这道题也可以不用动规来做，这里还是采用经典的动规解法。Maximum Subarray 中要求的是数组中最大子数组和，这里是求不相重叠的两个子数组和的和最大值，做过买卖股票系列的题的话这道题就非常容易了，既然我们已经求出了单一子数组的最大和，那么我们使用隔板法将数组一分为二，分别求这两段的最大子数组和，求相加后的最大值即为最终结果。隔板前半部分的最大子数组和很容易求得，但是后半部分难道需要将索引从0开始依次计算吗？NO!!! 我们可以采用从后往前的方式进行遍历，这样时间复杂度就大大降低了。

Java

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: An integer denotes the sum of max two non-overlapping subarrays
     */
    public int maxTwoSubArrays(ArrayList<Integer> nums) {
        // -1 is not proper for illegal input
        if (nums == null || nums.isEmpty()) return -1;

        int size = nums.size();
        // get max sub array forward
        int[] maxSubArrayF = new int[size];
        forwardTraversal(nums, maxSubArrayF);
        // get max sub array backward
```

```
        int[] maxSubArrayB = new int[size];
        backwardTraversal(nums, maxSubArrayB);
        // get maximum subarray by iteration
        int maxTwoSub = Integer.MIN_VALUE;
        for (int i = 0; i < size - 1; i++) {
            // non-overlapping
            maxTwoSub = Math.max(maxTwoSub, maxSubArrayF[i] + maxSubArrayB[i + 1]);
        }

        return maxTwoSub;
    }

    private void forwardTraversal(List<Integer> nums, int[] maxSubArray) {
        int sum = 0, minSum = 0, maxSub = Integer.MIN_VALUE;
        int size = nums.size();
        for (int i = 0; i < size; i++) {
            minSum = Math.min(minSum, sum);
            sum += nums.get(i);
            maxSub = Math.max(maxSub, sum - minSum);
            maxSubArray[i] = maxSub;
        }
    }

    private void backwardTraversal(List<Integer> nums, int[] maxSubArray) {
        int sum = 0, minSum = 0, maxSub = Integer.MIN_VALUE;
        int size = nums.size();
        for (int i = size - 1; i >= 0; i--) {
            minSum = Math.min(minSum, sum);
            sum += nums.get(i);
            maxSub = Math.max(maxSub, sum - minSum);
            maxSubArray[i] = maxSub;
        }
    }
}
```

源码分析

前向搜索和逆向搜索我们使用私有方法实现，可读性更高。注意是求非重叠子数组和，故求 `maxTwoSub` 时i的范围为 `0, size - 2`, 前向数组索引为 i, 后向索引为 i + 1.

复杂度分析

前向和后向搜索求得最大子数组和，时间复杂度 $O(2n) = O(n)$, 空间复杂度 $O(n)$. 遍历子数组和的数组求最终两个子数组和的最大值，时间复杂度 $O(n)$. 故总的时间复杂度为 $O(n)$, 空间复杂度 $O(n)$.

Longest Increasing Continuous subsequence

Source

- lintcode: (397) Longest Increasing Continuous subsequence

Problem

Give you an integer array (index from 0 to n-1, where n is the size of this array)，find the longest increasing continuous subsequence in this array. (The definition of the longest increasing continuous subsequence here can be from right to left or from left to right)

Example

For `[5, 4, 2, 1, 3]`, the LICS is `[5, 4, 2, 1]`, return 4.

For `[5, 1, 2, 3, 4]`, the LICS is `[1, 2, 3, 4]`, return 4.

Note

O(n) time and O(1) extra space.

题解1

题目只要返回最大长度，注意此题中的连续递增指的是双向的，即可递增也可递减。简单点考虑可分两种情况，一种递增，另一种递减，跟踪最大递增长度，最后返回即可。也可以在一个 for 循环中搞定，只不过需要增加一布尔变量判断之前是递增还是递减。

Java - two for loop

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if (A == null || A.length == 0) return 0;

        int lics = 1, licsMax = 1, prev = A[0];
        // ascending order
        for (int a : A) {
            lics = (prev < a) ? lics + 1 : 1;
            licsMax = Math.max(licsMax, lics);
            prev = a;
        }
        // reset
        lics = 1;
        prev = A[0];
```

```
        // descending order
        for (int a : A) {
            lics = (prev > a) ? lics + 1 : 1;
            licsMax = Math.max(licsMax, lics);
            prev = a;
        }

        return licsMax;
    }
}
```

Java - one for loop

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if (A == null || A.length == 0) return 0;

        int start = 0, licsMax = 1;
        boolean ascending = false;
        for (int i = 1; i < A.length; i++) {
            // ascending order
            if (A[i - 1] < A[i]) {
                if (!ascending) {
                    ascending = true;
                    start = i - 1;
                }
            } else if (A[i - 1] > A[i]) {
            // descending order
                if (ascending) {
                    ascending = false;
                    start = i - 1;
                }
            } else {
                start = i - 1;
            }
            licsMax = Math.max(licsMax, i - start + 1);
        }

        return licsMax;
    }
}
```

源码分析

使用两个 for 循环时容易在第二次循环忘记重置。使用一个 for 循环时使用下标来计数较为方便。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

题解2 - 动态规划

除了题解1 中分两种情况讨论外，我们还可以使用动态规划求解。状态转移方程容易得到——要么向右增长，要么向左增长。相应的状态 `dp[i]` 即为从索引 i 出发所能得到的最长连续递增子序列。这样就避免了分两个循环处理了，这种思想对此题的 follow up 有特别大的帮助。

Java

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if (A == null || A.length == 0) return 0;

        int lics = 0;
        int[] dp = new int[A.length];
        for (int i = 0; i < A.length; i++) {
            if (dp[i] == 0) {
                lics = Math.max(lics, dfs(A, i, dp));
            }
        }

        return lics;
    }

    private int dfs(int[] A, int i, int[] dp) {
        if (dp[i] != 0) return dp[i];

        // increasing from xxx to left, right
        int left = 0, right = 0;
        // increasing from right to left
        if (i > 0 && A[i - 1] > A[i]) left = dfs(A, i - 1, dp);
        // increasing from left to right
        if (i + 1 < A.length && A[i + 1] > A[i]) right = dfs(A, i + 1, dp);

        dp[i] = 1 + Math.max(left, right);
        return dp[i];
    }
}
```

源码分析

dfs 中使用记忆化存储避免重复递归，分左右两个方向递增，最后取较大值。这种方法对于数组长度较长时栈会溢出。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 (n).

Reference

- Lintcode: Longest Increasing Continuous subsequence | codesolutiony

Longest Increasing Continuous subsequence II

Source

- lintcode: (398) Longest Increasing Continuous subsequence II

Problem

Give you an integer matrix (with row size n, column size m)，find the longest increasing continuous subsequence in this matrix. (The definition of the longest increasing continuous subsequence here can start at any row or column and go up/down/right/left any direction).

Example

Given a matrix:

```
[
  [1 ,2 ,3 ,4 ,5],
  [16,17,24,23,6],
  [15,18,25,22,7],
  [14,19,20,21,8],
  [13,12,11,10,9]
]
```

return 25

Challenge

O(nm) time and memory.

题解

题 Longest Increasing Continuous subsequence 的 follow up, 变成一道比较难的题了。从之前的一维 DP 变为现在的二维 DP，自增方向可从上下左右四个方向进行。需要结合 DFS 和动态规划两大重量级武器。

根据二维 DP 的通用方法，我们首先需要关注状态及状态转移方程，状态转移方程相对明显一点，即上下左右四个方向的元素值递增关系，根据此转移方程，**不难得到我们需要的状态为 `dp[i][j]` ——表示从坐标 `(i, j)` 出发所得到的最长连续递增子序列。**根据状态及转移方程我们不难得到初始化应该为1或者0，这要视具体情况而定。

这里我们可能会纠结的地方在于自增的方向，平时见到的二维 DP 自增方向都是从小到大，而这里的增长方向却不一定。**这里需要突破思维定势的地方在于我们可以不理会从哪个方向自增，只需要处理自增和边界条件即可。**根据转移方程可以知道使用递归来解决是比较好的方式，这里关键的地方就在于递归的终止条件。比较容易想到的一个递归终止条件自然是当前元素是整个矩阵中的最大元素，索引朝四个方向出发都无法自增，因此返回1. 另外可以预想到的是如果不进行记忆化存储，递归过程中自然会产生大量重复计算，根据记忆化存储的通用方法，这里可以以结果是否为0(初始化为0时)来进行区分。

Java

```
public class Solution {
    /**
     * @param A an integer matrix
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequenceII(int[][] A) {
        if (A == null || A.length == 0 || A[0].length == 0) return 0;

        int lics = 0;
        int[][] dp = new int[A.length][A[0].length];
        for (int row = 0; row < A.length; row++) {
            for (int col = 0; col < A[0].length; col++) {
                if (dp[row][col] == 0) {
                    lics = Math.max(lics, dfs(A, row, col, dp));
                }
            }
        }

        return lics;
    }

    private int dfs(int[][] A, int row, int col, int[][] dp) {
        if (dp[row][col] != 0) {
            return dp[row][col];
        }

        // increasing from xxx to up, down, left, right
        int up = 0, down = 0, left = 0, right = 0;
        // increasing from down to up
        if (row > 0 && A[row - 1][col] > A[row][col]) {
            up = dfs(A, row - 1, col, dp);
        }
        // increasing from up to down
        if (row + 1 < A.length && A[row + 1][col] > A[row][col]) {
            down = dfs(A, row + 1, col, dp);
        }
        // increasing from right to left
        if (col > 0 && A[row][col - 1] > A[row][col]) {
            left = dfs(A, row, col - 1, dp);
        }
        // increasing from left to right
        if (col + 1 < A[0].length && A[row][col + 1] > A[row][col]) {
            right = dfs(A, row, col + 1, dp);
        }
        // return maximum of up, down, left, right
        dp[row][col] = 1 + Math.max(Math.max(up, down), Math.max(left, right));

        return dp[row][col];
    }
}
```

源码分析

dfs 递归最深一层即矩阵中最大的元素处，然后逐层返回。这道题对状态 `dp[i][j]` 的理解很重要，否则会

陷入对上下左右四个方向的迷雾中。

复杂度分析

由于引入了记忆化存储，时间复杂度逼近 $O(mn)$, 空间复杂度 $O(mn)$.

Reference

- Lintcode: Longest Increasing Continuous subsequence II | codesolutiony

Graph

本章主要总结图与搜索相关题目。

Topological Sorting

Source

- lintcode: (127) Topological Sorting
- Topological Sorting - GeeksforGeeks

```
Given an directed graph, a topological order of the graph nodes is defined as follow:

For each directed edge A -> B in graph, A must before B in the order list.
The first node in the order can be any node in the graph with no nodes direct to it.
Find any topological order for the given graph.
```

Example For graph as follow:

```
The topological order can be:

[0, 1, 2, 3, 4, 5]
[0, 2, 3, 1, 5, 4]
...
Note
You can assume that there is at least one topological order in the graph.

Challenge
Can you do it in both BFS and DFS?
```

题解1 - DFS and BFS

图搜索相关的问题较为常见的解法是用 DFS，这里结合 BFS 进行求解，分为三步走：

1. 统计各定点的入度——只需统计节点在邻接列表中出现的次数即可知。
2. 遍历图中各节点，找到入度为0的节点。
3. 对入度为0的节点进行递归 DFS，将节点加入到最终返回结果中。

C++

```
/**
 * Definition for Directed graph.
 * struct DirectedGraphNode {
 *     int label;
 *     vector<DirectedGraphNode *> neighbors;
 *     DirectedGraphNode(int x) : label(x) {};
 * };
 */
class Solution {
public:
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
    vector<DirectedGraphNode*> topSort(vector<DirectedGraphNode*> graph) {
        vector<DirectedGraphNode*> result;
        if (graph.size() == 0) return result;

        map<DirectedGraphNode*, int> indegree;
        // get indegree of all DirectedGraphNode
        indeg(graph, indegree);
        // dfs recursively
        for (int i = 0; i < graph.size(); ++i) {
            if (indegree[graph[i]] == 0) {
                dfs(indegree, graph[i], result);
            }
        }

        return result;
    }

private:
    /** get indegree of all DirectedGraphNode
     *
     */
    void indeg(vector<DirectedGraphNode*> &graph,
                   map<DirectedGraphNode*, int> &indegree) {

        for (int i = 0; i < graph.size(); ++i) {
            for (int j = 0; j < graph[i]->neighbors.size(); j++) {
                if (indegree.find(graph[i]->neighbors[j]) == indegree.end()) {
                    indegree[graph[i]->neighbors[j]] = 1;
                } else {
                    indegree[graph[i]->neighbors[j]] += 1;
                }
            }
        }
    }

    void dfs(map<DirectedGraphNode*, int> &indegree, DirectedGraphNode *i,
             vector<DirectedGraphNode*> &ret) {

        ret.push_back(i);
        indegree[i]--;
        for (int j = 0; j < i->neighbors.size(); ++j) {
            indegree[i->neighbors[j]]--;
            if (indegree[i->neighbors[j]] == 0) {
                dfs(indegree, i->neighbors[j], ret);
```

```
            }
        }
    }
};
```

源码分析

C++中使用 unordered_map 可获得更高的性能，私有方法中使用引用传值。

复杂度分析

以 V 表示顶点数，E 表示有向图中边的条数。

首先获得节点的入度数，时间复杂度为 $O(V+E)$, 使用了哈希表存储，空间复杂度为 $O(V)$. 遍历图求得入度为0的节点，时间复杂度为 $O(V)$. 仅在入度为0时调用 DFS，故时间复杂度为 $O(V+E)$.

需要注意的是这里的 DFS 不是纯 DFS，使用了 BFS 的思想进行了优化，否则一个节点将被遍历多次，时间复杂度可能恶化为指数级别。

综上，时间复杂度近似为 $O(V+E)$, 空间复杂度为 $O(V)$.

题解2 - BFS

拓扑排序除了可用 DFS 求解外，也可使用 BFS, 具体方法为：

1. 获得图中各节点的入度。
2. BFS 首先遍历求得入度数为0的节点，入队，便于下一次 BFS。
3. 队列不为空时，弹出队顶元素并对其邻接节点进行 BFS，将入度为0的节点加入到最终结果和队列中，重复此过程直至队列为空。

C++

```
/**
 * Definition for Directed graph.
 * struct DirectedGraphNode {
 *     int label;
 *     vector<DirectedGraphNode *> neighbors;
 *     DirectedGraphNode(int x) : label(x) {};
 * };
 */
class Solution {
public:
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
    vector<DirectedGraphNode*> topSort(vector<DirectedGraphNode*> graph) {
        vector<DirectedGraphNode*> result;
        if (graph.size() == 0) return result;

        map<DirectedGraphNode*, int> indegree;
```

```
        // get indegree of all DirectedGraphNode
        indeg(graph, indegree);
        queue<DirectedGraphNode*> q;
        // bfs
        bfs(graph, indegree, q, result);

        return result;
    }

private:
    /** get indegree of all DirectedGraphNode
     *
     */
    void indeg(vector<DirectedGraphNode*> &graph,
                  map<DirectedGraphNode*, int> &indegree) {

        for (int i = 0; i < graph.size(); ++i) {
            for (int j = 0; j < graph[i]->neighbors.size(); j++) {
                if (indegree.find(graph[i]->neighbors[j]) == indegree.end()) {
                    indegree[graph[i]->neighbors[j]] = 1;
                } else {
                    indegree[graph[i]->neighbors[j]] += 1;
                }
            }
        }
    }

    void bfs(vector<DirectedGraphNode*> &graph, map<DirectedGraphNode*, int> &indegree,
             queue<DirectedGraphNode *> &q, vector<DirectedGraphNode*> &ret) {

        for (int i = 0; i < graph.size(); ++i) {
            if (indegree[graph[i]] == 0) {
                ret.push_back(graph[i]);
                q.push(graph[i]);
            }
        }

        while (!q.empty()) {
            DirectedGraphNode *cur = q.front();
            q.pop();
            for(int j = 0; j < cur->neighbors.size(); ++j) {
                indegree[cur->neighbors[j]]--;
                if (indegree[cur->neighbors[j]] == 0) {
                    ret.push_back(cur->neighbors[j]);
                    q.push(cur->neighbors[j]);
                }
            }
        }
    }
};
```

源码分析

C++中在判断入度是否为0时将对 map 产生副作用，在求入度数时只有入度数大于等于1才会出现在 map 中，故不在 map 中时直接调用 indegree 方法将产生新的键值对，初始值为0，恰好满足此题需求。

复杂度分析

同题解1 的分析，时间复杂度为 $O(V+E)$, 空间复杂度为 $O(V)$.

Reference

- Topological Sorting 参考程序 Java/C++/Python

Word Ladder

Source

- leetcode: Word Ladder | LeetCode OJ
- lintcode: (120) Word Ladder

Problem

Given two words (*start* and *end*), and a dictionary, find the length of shortest transformation sequence from *start* to *end*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

Example

Given: *start* = `"hit"` *end* = `"cog"` *dict* = `["hot","dot","dog","lot","log"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`, return its length `5`.

Note

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

题解

咋一看还以为是 Edit Distance 的变体，仔细审题后发现和动态规划没啥关系。题中有两大关键点：一次只能改动一个字符；改动的中间结果必须出现在词典中。那么大概总结下来共有四种情形：

1. start 和 end 相等。
2. end 在 dict 中，且 start 可以转换为 dict 中的一个单词。
3. end 不在 dict 中，但可由 start 或者 dict 中的一个单词转化而来。
4. end 无法由 start 转化而来。

由于中间结果也必须出现在词典中，故此题相当于图搜索问题，将 start, end, dict 中的单词看做图中的节点，节点与节点（单词与单词）可通过一步转化得到，可以转换得到的节点相当于边的两个节点，边的权重为1（都是通过1步转化）。到这里问题就比较明确了，相当于搜索从 start 到 end 两点间的最短距离，即 Dijkstra 最短路径算法。**通过 BFS 和哈希表实现。**

首先将 start 入队，随后弹出该节点，比较其和 end 是否相同；再从 dict 中选出所有距离为1的单词入队，并将所有与当前节点距离为1且未访问过的节点（需要使用哈希表）入队，方便下一层遍历时使用，直至队列为空。

Java

```java
public class Solution {
    /**
      * @param start, a string
      * @param end, a string
      * @param dict, a set of string
      * @return an integer
      */
    public int ladderLength(String start, String end, Set<String> dict) {
        if (start == null && end == null) return 0;
        if (start.length() == 0 && end.length() == 0) return 0;
        assert(start.length() == end.length());
        if (dict == null || dict.size() == 0) {
            return 0;
        }

        int ladderLen = 1;
        dict.add(end); // add end to dict, important!
        Queue<String> q = new LinkedList<String>();
        Set<String> hash = new HashSet<String>();
        q.offer(start);
        hash.add(start);
        while (!q.isEmpty()) {
            ladderLen++;
            int qLen = q.size();
            for (int i = 0; i < qLen; i++) {
                String strTemp = q.poll();

                for (String nextWord : getNextWords(strTemp, dict)) {
                    if (nextWord.equals(end)) return ladderLen;
                    // filter visited word in the dict
                    if (hash.contains(nextWord)) continue;
                    q.offer(nextWord);
                    hash.add(nextWord);
                }
            }
        }

        return 0;
    }

    private Set<String> getNextWords(String curr, Set<String> dict) {
        Set<String> nextWords = new HashSet<String>();
        for (int i = 0; i < curr.length(); i++) {
            char[] chars = curr.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                chars[i] = c;
                String temp = new String(chars);
                if (dict.contains(temp)) {
                    nextWords.add(temp);
                }
            }
        }

        return nextWords;
    }
}
```

源码分析

`getNextWords` 的实现

首先分析给定单词 `curr` 并从 dict 中选出所有距离为1 的单词。常规的思路可能是将 `curr` 与 dict 中的单词逐个比较，并遍历每个字符串，返回距离为1的单词组。这种找距离为1的节点的方法复杂度为 $l(length\ of\ word) \times n(size\ of\ dict) \times m(queue\ length) = O(lmn)$. 在 dict 较长时会 TLE. 其实根据 dict 的数据结构特点，比如查找任一元素的时间复杂度可认为是 $O(1)$. 根据哈希表和单个单词长度通常不会太长这一特点，我们就可以根据给定单词构造到其距离为一的单词变体，然后查询其是否在 dict 中，这种实现的时间复杂度为 $O(26(a\ to\ z) \times l \times m) = O(lm)$, 与 dict 长度没有太大关系，大大优化了时间复杂度。

经验教训：根据给定数据结构特征选用合适的实现，遇到哈希表时多用其查找的 $O(1)$ 特性。

BFS 和哈希表的配合使用

BFS 用作搜索，哈希表用于记录已经访问节点。在可以改变输入数据的前提下，需要将 end 加入 dict 中，否则对于不在 dict 中出现的 end 会有问题。

复杂度分析

主要在于 `getNextWords` 方法的时间复杂度，时间复杂度 $O(lmn)$。使用了队列存储中间处理节点，空间复杂度平均条件下应该是常量级别，当然最坏条件下可能恶化为 $O(n)$, 即 dict 中某个点与其他点距离均为 1.

Reference

- Word Ladder 参考程序 Java/C++/Python
- Java Solution using Dijkstra's algorithm, with explanation - Leetcode Discuss

Data Structure

本章主要总结数据结构如 Queue, Stack 等相关的题。

Implement Queue by Two Stacks

Source

- lintcode: (40) Implement Queue by Two Stacks

```
As the title described, you should only use two stacks to implement a queue's actions.

The queue should support push(element),
pop() and top() where pop is pop the first(a.k.a front) element in the queue.

Both pop and top methods should return the value of first element.

Example
For push(1), pop(), push(2), push(3), top(), pop(), you should return 1, 2 and 2

Challenge
implement it by two stacks, do not use any other data structure and push,
pop and top should be O(1) by AVERAGE.
```

题解

两个栈模拟队列，栈是 LIFO, 队列是 FIFO, 故用两个栈模拟队列时可结合栈1和栈2, LIFO + LIFO ==> FIFO, 即先将一个栈元素全部 push 到另一个栈，效果即等价于 Queue.

Java

```java
public class Solution {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public Solution() {
        // source stack
        stack1 = new Stack<Integer>();
        // target stack
        stack2 = new Stack<Integer>();
    }

    public void push(int element) {
        stack1.push(element);
    }

    public int pop() {
        if (stack2.empty()) {
            stack1ToStack2(stack1, stack2);
        }
        return stack2.pop();
    }

    public int top() {
```

```
        if (stack2.empty()) {
            stack1ToStack2(stack1, stack2);
        }
        return stack2.peek();
    }

    private void stack1ToStack2(Stack<Integer> stack1, Stack<Integer> stack2) {
        while (!stack1.empty()) {
            stack2.push(stack1.pop());
        }
    }
}
```

源码分析

将栈1作为原始栈，将栈1元素压入栈2是公共方法，故写成一个私有方法。

复杂度分析

视连续 push 的元素而定，时间复杂度近似为 $O(1)$.

Reference

- Implement Queue by Two Stacks 参考程序 Java/C++/Python

Min Stack

Source

- lintcode: (12) Min Stack

```
Implement a stack with min() function,
which will return the smallest number in the stack.

It should support push, pop and min operation all in O(1) cost.

Example
Operations: push(1), pop(), push(2), push(3), min(), push(1), min() Return: 1, 2, 1

Note
min operation will never be called if there is no number in the stack
```

题解

『最小』栈，要求在栈的基础上实现可以在 $O(1)$ 的时间内找出最小值，一般这种 $O(1)$的实现往往就是哈希表或者哈希表的变体，这里简单起见可以另外克隆一个栈用以跟踪当前栈的最小值。

Java

```java
public class Solution {
    public Solution() {
        stack1 = new Stack<Integer>();
        stack2 = new Stack<Integer>();
    }

    public void push(int number) {
        stack1.push(number);
        if (stack2.empty()) {
            stack2.push(number);
        } else {
            stack2.push(Math.min(number, stack2.peek()));
        }
    }

    public int pop() {
        stack2.pop();
        return stack1.pop();
    }

    public int min() {
        return stack2.peek();
    }

    private Stack<Integer> stack1; // original stack
    private Stack<Integer> stack2; // min stack
```

```
}
```

源码分析

取最小栈的栈顶值时需要先判断是否为空栈(而不仅是 null)。

复杂度分析

均为 $O(1)$.

Sliding Window Maximum

Source

- leetcode: Sliding Window Maximum | LeetCode OJ
- lintcode: (362) Sliding Window Maximum

```
Given an array of n integer with duplicate number, and a moving window(size k),
move the window at each iteration from the start of the array,
find the maximum number inside the window at each moving.

Example
For array [1, 2, 7, 7, 8], moving window size k = 3. return [7, 7, 8]

At first the window is at the start of the array like this

[|1, 2, 7| ,7, 8] , return the maximum 7;

then the window move one step forward.

[1, |2, 7 ,7|, 8], return the maximum 7;

then the window move one step forward again.

[1, 2, |7, 7, 8|], return the maximum 8;

Challenge
o(n) time and O(k) memory
```

题解

$O(nk)$ 的时间复杂度的方法很容易想到，不停地从当前窗口中取最大就好了。但其实可以发现下一个窗口的最大值与当前窗口的最大值其实是有一定关系的，但这个关系不是简单的将前一个窗口的最大值传递给下一个窗口，**因为数组中每一个元素都是有其作用范围的，超过窗口长度后就失效了！**所以现在思路就稍微清晰一些了，将前一个窗口的最大值传递给下一个窗口时需要判断当前遍历的元素下标和前一个窗口的最大元素下标之差是否已经超过一个窗口长度。

问题来了，思路基本定型，现在就是选用合适的数据结构了。根据上面的思路，这种数据结构应该能在 $O(1)$ 的时间内返回最大值，且存储的元素最大可以不超过窗口长度。常规一点的可以采用队列，但是此题中使用普通队列似乎还是很难实现，因为要在 $O(1)$ 的时间内返回最大值。符合这个要求的数据结构必须能支持从两端对队列元素进行维护，其中一种实现方法为队首维护最大值，队尾用于插入新元素。双端队列无疑了，有关双端队列的科普见 双端队列。可以自己试着以一个实际例子来帮助理解。

Java

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

```java
     * @param nums: A list of integers.
     * @return: The maximum number inside the window at each moving.
     */
    public ArrayList<Integer> maxSlidingWindow(int[] nums, int k) {
        ArrayList<Integer> winMax = new ArrayList<Integer>();
        if (nums == null || nums.length == 0 || k <= 0) return winMax;

        int len = nums.length;
        Deque<Integer> deque = new ArrayDeque<Integer>();
        for (int i = 0; i < len; i++) {
            // remove the smaller in the rear of queue
            while ((!deque.isEmpty()) && (nums[i] > deque.peekLast())) {
                deque.pollLast();
            }
            // push element in the rear of queue
            deque.offer(nums[i]);
            // remove invalid max
            if (i + 1 > k && deque.peekFirst() == nums[i - k]) {
                deque.pollFirst();
            }
            // add max in current window
            if (i + 1 >= k) {
                winMax.add(deque.peekFirst());
            }
        }

        return winMax;
    }
}
```

源码分析

1. 移除队尾元素时首先判断是否为空，因为在移除过程中可能会将队列元素清空。
2. 在移除队尾元素时 `nums[i] > deque.peekLast()` 不可取等于号，因为这样会将相等的元素全部移除，这样会在窗口中部分元素相等时错误地移除本该添加到最终结果的元素。
3. 移除失效元素和添加元素到最终结果时需要注意下标 `i` 和 `k` 的关系，建议举例确定。

复杂度分析

时间复杂度 $O(n)$, 空间复杂度 $O(k)$. 空间复杂度可能不是那么直观，可以这么理解，双端队列中的元素最多只能存活 k 次，因为只有最大元素的存活时间最久，而最大元素在超过窗口长度时即被移除，故空间复杂度为 $O(k)$.

Reference

- 《剑指 Offer》
- sliding-window-maximum 参考程序 Java/C++/Python
- Maximum of all subarrays of size k (Added a O(n) method) - GeeksforGeeks

Longest Words

Source

- lintcode: (133) Longest Words

```
Given a dictionary, find all of the longest words in the dictionary.

Example
Given

{
  "dog",
  "google",
  "facebook",
  "internationalization",
  "blabla"
}
the longest words are(is) ["internationalization"].

Given

{
  "like",
  "love",
  "hate",
  "yes"
}
the longest words are ["like", "love", "hate"].

Challenge
It's easy to solve it in two passes, can you do it in one pass?
```

题解

简单题，容易想到的是首先遍历以便，找到最长的字符串，第二次遍历时取最长的放到最终结果中。但是如果只能进行一次遍历呢？一次遍历意味着需要维护当前遍历的最长字符串，这必然有比较与更新删除操作，这种情况下使用双端队列最为合适，这道题稍微特殊一点，不必从尾端插入，只需在遍历时若发现比数组中最长的元素还长时删除整个列表。

Java

```java
class Solution {
    /**
     * @param dictionary: an array of strings
     * @return: an arraylist of strings
     */
    ArrayList<String> longestWords(String[] dictionary) {
        ArrayList<String> result = new ArrayList<String>();
```

```
        if (dictionary == null || dictionary.length == 0) return result;

        for (String str : dictionary) {
            // combine empty and shorter length
            if (result.isEmpty() || str.length() > result.get(0).length()) {
                result.clear();
                result.add(str);
            } else if (str.length() == result.get(0).length()) {
                result.add(str);
            }
        }

        return result;
    }
}
```

源码分析

熟悉变长数组的常用操作。

复杂度分析

时间复杂度 $O(n)$, 最坏情况下需要保存 n - 1个字符串，空间复杂度 $O(n)$.

Reference

- Lintcode: Longest Words | codesolutiony

Heapify

Source

- lintcode: (130) Heapify

```
Given an integer array, heapify it into a min-heap array.

For a heap array A, A[0] is the root of heap, and for each A[i],
A[i * 2 + 1] is the left child of A[i] and A[i * 2 + 2] is the right child of A[i].

Example
Given [3,2,1,4,5], return [1,2,3,4,5] or any legal heap array.

Challenge
O(n) time complexity

Clarification
What is heap?

Heap is a data structure, which usually have three methods: push, pop and top.
where "push" add a new element the heap,
"pop" delete the minimum/maximum element in the heap,
"top" return the minimum/maximum element.

What is heapify?
Convert an unordered integer array into a heap array.
If it is min-heap, for each element A[i],
we will get A[i * 2 + 1] >= A[i] and A[i * 2 + 2] >= A[i].

What if there is a lot of solutions?
Return any of them.
```

题解

参考前文提到的 Heap Sort 可知此题要实现的只是小根堆的堆化过程，并不要求堆排。

C++

```cpp
class Solution {
public:
    /**
     * @param A: Given an integer array
     * @return: void
     */
    void heapify(vector<int> &A) {
        // build min heap
        for (int i = A.size() / 2; i >= 0; --i) {
            min_heap(A, i);
        }
```

```
    }

private:
    void min_heap(vector<int> &nums, int k) {
        int len = nums.size();
        while (k < len) {
            int min_index = k;
            // left leaf node search
            if (k * 2 + 1 < len && nums[k * 2 + 1] < nums[min_index]) {
                min_index = k * 2 + 1;
            }
            // right leaf node search
            if (k * 2 + 2 < len && nums[k * 2 + 2] < nums[min_index]) {
                min_index = k * 2 + 2;
            }
            if (k == min_index) {
                break;
            }
            // swap with the minimal
            int temp = nums[k];
            nums[k] = nums[min_index];
            nums[min_index] = temp;
            // not only current index
            k = min_index;
        }
    }
};
```

源码分析

堆排的简化版，最后一步 `k = min_index` 不能忘，因为增删节点时需要重新建堆，这样才能保证到第一个节点时数组已经是二叉堆。

复杂度分析

由于采用的是自底向上的建堆方式，时间复杂度为 (N), 证明待补充...

Reference

- Heap Sort
- Heapify 参考程序 Java/C++/Python

Problem Misc

本章主要总结暂时不方便归到其他章节的题目。

Nuts and Bolts Problem

Source

- lintcode: (399) Nuts & Bolts Problem

```
Given a set of n nuts of different sizes and n bolts of different sizes.
There is a one-one mapping between nuts and bolts.
Comparison of a nut to another nut or a bolt to another bolt is not allowed.
It means nut can only be compared with bolt and bolt can only
be compared with nut to see which one is bigger/smaller.

We will give you a compare function to compare nut with bolt.

Example
Given nuts = ['ab','bc','dd','gg'], bolts = ['AB','GG', 'DD', 'BC'].

Your code should find the matching bolts and nuts.

one of the possible return:

nuts = ['ab','bc','dd','gg'], bolts = ['AB','BC','DD','GG'].

we will tell you the match compare function.
If we give you another compare function.

the possible return is the following:

nuts = ['ab','bc','dd','gg'], bolts = ['BC','AA','DD','GG'].

So you must use the compare function that we give to do the sorting.

The order of the nuts or bolts does not matter.
You just need to find the matching bolt for each nut.
```

题解

首先结合例子读懂题意，本题为 nuts 和 bolts 的配对问题，但是需要根据题目所提供的比较函数，且 nuts 与 nuts 之间的元素无法直接比较，compare 仅能在 nuts 与 bolts 之间进行。首先我们考虑若没有比较函数的限制，那么我们可以分别对 nuts 和 bolts 进行排序，由于是一一配对，故排完序后即完成配对。那么在只能通过比较对方元素得知相对大小时怎么完成排序呢？

我们容易通过以一组元素作为参考进行遍历获得两两相等的元素，这样一来在最坏情况下时间复杂度为 $O(n^2)$, 相当于冒泡排序。根据排序算法理论可知基于比较的排序算法最好的时间复杂度为 $O(n\log n)$, 也就是说这道题应该是可以进一步优化。回忆一些基于比较的排序算法，能达到 $O(n\log n)$ 时间复杂度的有堆排、归并排序和快速排序，由于这里只能通过比较得到相对大小的关系，故可以联想到快速排序。

快速排序的核心即为定基准，划分区间。由于这里只能以对方的元素作为基准，故一趟划分区间后仅能得到某一方基准元素排序后的位置，那通过引入 $O(n)$ 的额外空间来对已处理的基准元素进行标记如何呢？

这种方法实现起来较为困难，因为只能对一方的元素划分区间，而对方的元素无法划分区间进而导致递归无法正常进行。

山穷水尽疑无路，柳暗花明又一村。由于只能通过对方进行比较，故需要相互配合进行 partition 操作(这个点确实难以想到)。核心在于：**首先使用 nuts 中的某一个元素作为基准对 bolts 进行 partition 操作，随后将 bolts 中得到的基准元素作为基准对 nuts 进行 partition 操作。**

Python

```
# class Comparator:
#     def cmp(self, a, b)
# You can use Compare.cmp(a, b) to compare nuts "a" and bolts "b",
# if "a" is bigger than "b", it will return 1, else if they are equal,
# it will return 0, else if "a" is smaller than "b", it will return -1.
# When "a" is not a nut or "b" is not a bolt, it will return 2, which is not valid.
class Solution:
    # @param nuts: a list of integers
    # @param bolts: a list of integers
    # @param compare: a instance of Comparator
    # @return: nothing
    def sortNutsAndBolts(self, nuts, bolts, compare):
        if nuts is None or bolts is None:
            return
        if len(nuts) != len(bolts):
            return
        self.qsort(nuts, bolts, 0, len(nuts) - 1, compare)

    def qsort(self, nuts, bolts, l, u, compare):
        if l >= u:
            return
        # find the partition index for nuts with bolts[l]
        part_inx = self.partition(nuts, bolts[l], l, u, compare)
        # partition bolts with nuts[part_inx]
        self.partition(bolts, nuts[part_inx], l, u, compare)
        # qsort recursively
        self.qsort(nuts, bolts, l, part_inx - 1, compare)
        self.qsort(nuts, bolts, part_inx + 1, u, compare)

    def partition(self, alist, pivot, l, u, compare):
        m = l
        i = l + 1
        while i <= u:
            if compare.cmp(alist[i], pivot) == -1 or \
               compare.cmp(pivot, alist[i]) == 1:
                m += 1
                alist[i], alist[m] = alist[m], alist[i]
                i += 1
            elif compare.cmp(alist[i], pivot) == 0 or \
                 compare.cmp(pivot, alist[i]) == 0:
                # swap nuts[l]/bolts[l] with pivot
                alist[i], alist[l] = alist[l], alist[i]
            else:
                i += 1
        # move pivot to proper index
        alist[l], alist[m] = alist[m], alist[l]
```

```
        return m
```

C++

```
/**
 * class Comparator {
 *     public:
 *      int cmp(string a, string b);
 * };
 * You can use compare.cmp(a, b) to compare nuts "a" and bolts "b",
 * if "a" is bigger than "b", it will return 1, else if they are equal,
 * it will return 0, else if "a" is smaller than "b", it will return -1.
 * When "a" is not a nut or "b" is not a bolt, it will return 2, which is not valid.
*/
class Solution {
public:
    /**
     * @param nuts: a vector of integers
     * @param bolts: a vector of integers
     * @param compare: a instance of Comparator
     * @return: nothing
     */
    void sortNutsAndBolts(vector<string> &nuts, vector<string> &bolts, Comparator compare
        if (nuts.empty() || bolts.empty()) return;
        if (nuts.size() != bolts.size()) return;

        qsort(nuts, bolts, compare, 0, nuts.size() - 1);
    }

private:
    void qsort(vector<string>& nuts, vector<string>& bolts, Comparator compare,
               int l, int u) {

        if (l >= u) return;
        // find the partition index for nuts with bolts[l]
        int part_inx = partition(nuts, bolts[l], compare, l, u);
        // partition bolts with nuts[part_inx]
        partition(bolts, nuts[part_inx], compare, l, u);
        // qsort recursively
        qsort(nuts, bolts, compare, l, part_inx - 1);
        qsort(nuts, bolts, compare, part_inx + 1, u);
    }

    int partition(vector<string>& str, string& pivot, Comparator compare,
                  int l, int u) {

        int m = l;
        for (int i = l + 1; i <= u; ++i) {
            if (compare.cmp(str[i], pivot) == -1 ||
                compare.cmp(pivot, str[i]) == 1) {

                ++m;
                std::swap(str[m], str[i]);
            } else if (compare.cmp(str[i], pivot) == 0 ||
                       compare.cmp(pivot, str[i]) == 0) {
                // swap nuts[l]/bolts[l] with pivot
                std::swap(str[i], str[l]);
```

```
                --i;
            }
        }
        // move pivot to proper index
        std::swap(str[m], str[l]);

        return m;
    }
};
```

Java

```
/**
 * public class NBCompare {
 *     public int cmp(String a, String b);
 * }
 * You can use compare.cmp(a, b) to compare nuts "a" and bolts "b",
 * if "a" is bigger than "b", it will return 1, else if they are equal,
 * it will return 0, else if "a" is smaller than "b", it will return -1.
 * When "a" is not a nut or "b" is not a bolt, it will return 2, which is not valid.
*/
public class Solution {
    /**
     * @param nuts: an array of integers
     * @param bolts: an array of integers
     * @param compare: a instance of Comparator
     * @return: nothing
     */
    public void sortNutsAndBolts(String[] nuts, String[] bolts, NBComparator compare) {
        if (nuts == null || bolts == null) return;
        if (nuts.length != bolts.length) return;

        qsort(nuts, bolts, compare, 0, nuts.length - 1);
    }

    private void qsort(String[] nuts, String[] bolts, NBComparator compare,
                       int l, int u) {
        if (l >= u) return;
        // find the partition index for nuts with bolts[l]
        int part_inx = partition(nuts, bolts[l], compare, l, u);
        // partition bolts with nuts[part_inx]
        partition(bolts, nuts[part_inx], compare, l, u);
        // qsort recursively
        qsort(nuts, bolts, compare, l, part_inx - 1);
        qsort(nuts, bolts, compare, part_inx + 1, u);
    }

    private int partition(String[] str, String pivot, NBComparator compare,
                          int l, int u) {
        //
        int m = l;
        for (int i = l + 1; i <= u; i++) {
            if (compare.cmp(str[i], pivot) == -1 ||
                compare.cmp(pivot, str[i]) == 1) {
                //
                m++;
```

```
                swap(str, i, m);
            } else if (compare.cmp(str[i], pivot) == 0 ||
                        compare.cmp(pivot, str[i]) == 0) {
                // swap nuts[l]/bolts[l] with pivot
                swap(str, i, l);
                i--;
            }
        }
        // move pivot to proper index
        swap(str, m, l);

        return m;
    }

    private void swap(String[] str, int l, int r) {
        String temp = str[l];
        str[l] = str[r];
        str[r] = temp;
    }
}
```

源码分析

难以理解的可能在 `partition` 部分，不仅需要使用 `compare.cmp(alist[i], pivot)`，同时也需要使用 `compare.cmp(pivot, alist[i])`，否则答案有误。第二个在于 `alist[i] == pivot` 时，需要首先将其和 `alist[l]` 交换，因为 `i` 是从 `l+1` 开始处理的，将 `alist[l]` 换过来后可继续和 pivot 进行比较。在 while 循环退出后在将当前遍历到的小于 pivot 的元素 alist[m] 和 alist[l] 交换，此时基准元素正确归位。对这一块不是很清楚的举个例子就明白了。

复杂度分析

快排的思路，时间复杂度为 $O(2n\log n)$, 使用了一些临时变量，空间复杂度 $O(1)$.

Reference

- LintCode/Nuts & Bolts Problem.py at master · algorhythms/LintCode

String to Integer

Source

- leetcode: String to Integer (atoi) | LeetCode OJ
- lintcode: (54) String to Integer(atoi)

```
Implement function atoi to convert a string to an integer.

If no valid conversion could be performed, a zero value is returned.

If the correct value is out of the range of representable values,
INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

Example
"10" => 10

"-1" => -1

"123123123123123" => 2147483647

"1.0" => 1
```

题解

经典的字符串转整数题，边界条件比较多，比如是否需要考虑小数点，空白及非法字符的处理，正负号的处理，科学计数法等。最先处理的是空白字符，然后是正负号，接下来只要出现非法字符(包含正负号，小数点等，无需对这两类单独处理)即退出，否则按照正负号的整数进位加法处理。

Java

```java
public class Solution {
    /**
     * @param str: A string
     * @return An integer
     */
    public int atoi(String str) {
        if (str == null || str.length() == 0) return 0;

        // trim left and right spaces
        String strTrim = str.trim();
        int len = strTrim.length();
        // sign symbol for positive and negative
        int sign = 1;
        // index for iteration
        int i = 0;
        if (strTrim.charAt(i) == '+') {
            i++;
        } else if (strTrim.charAt(i) == '-') {
```

```
            sign = -1;
            i++;
        }

        // store the result as long to avoid overflow
        long result = 0;
        while (i < len) {
            if (strTrim.charAt(i) < '0' || strTrim.charAt(i) > '9') {
                break;
            }
            result = 10 * result + sign * (strTrim.charAt(i) - '0');
            // overflow
            if (result > Integer.MAX_VALUE) {
                return Integer.MAX_VALUE;
            } else if (result < Integer.MIN_VALUE) {
                return Integer.MIN_VALUE;
            }
            i++;
        }

        return (int)result;
    }
}
```

源码分析

符号位使用整型表示，便于后期相乘相加。在 while 循环中需要注意判断是否已经溢出，如果放在 while 循环外面则有可能超过 long 型范围。

复杂度分析

略

Reference

- String to Integer (atoi) 参考程序 Java/C++/Python

Insert Interval

Source

- leetcode: Insert Interval | LeetCode OJ
- lintcode: (30) Insert Interval

```
Given a non-overlapping interval list which is sorted by start point.
Insert a new interval into it,
make sure the list is still in order and non-overlapping
(merge intervals if necessary).

Example
Insert [2, 5] into [[1,2], [5,9]], we get [[1,9]].

Insert [3, 4] into [[1,2], [5,9]], we get [[1,2], [3,4], [5,9]].
```

题解

这道题看似简单，但其实实现起来不那么容易，因为若按照常规思路，需要分很多种情况考虑，如半边相等的情况。以返回新数组为例，首先，遍历原数组肯定是必须的，以 `[N]` 代表 `newInterval`，`[I]` 代表当前遍历到的 `interval`，那么有以下几种情况：

1. `[N], [I]` <==> `newInterval.end < interval.start`，由于 intervals 中的间隔数组已经为升序排列，那么遍历到的下一个间隔的左边元素必然也大于新间隔的右边元素。
2. `[NI]` <==> `newInterval.end == interval.start`，这种情况下需要进行合并操作。
3. `[IN]` <==> `newInterval.start == interval.end`，这种情况下也需要进行合并。
4. `[I], [N]` <==> `newInterval.start > interval.end`，这意味着 `newInterval` 有可能在此处插入，也有可能在其后面的间隔插入。故遍历时需要在这种情况下做一些标记以确定最终插入位置。

由于间隔都是互不重叠的，故其关系只可能为以上四种中的某几个。1和4两种情况很好处理，关键在于2和3的处理。由于2和3这种情况都将生成新的间隔，且这种情况一旦发生，**原来的 `newInterval` 即被新的合并间隔取代，这是一个非常关键的突破口。**

Java

```
/**
 * Definition of Interval:
 * public classs Interval {
 *     int start, end;
 *     Interval(int start, int end) {
 *         this.start = start;
 *         this.end = end;
 *     }
 */
```

```java
class Solution {
    /**
     * Insert newInterval into intervals.
     * @param intervals: Sorted interval list.
     * @param newInterval: A new interval.
     * @return: A new sorted interval list.
     */
    public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval
        ArrayList<Interval> result = new ArrayList<Interval>();
        if (intervals == null || intervals.isEmpty()) {
            if (newInterval != null) {
                result.add(newInterval);
            }
            return result;
        }

        int insertPos = 0;
        for (Interval interval : intervals) {
            if (newInterval.end < interval.start) {
                // case 1: [new], [old]
                result.add(interval);
            } else if (interval.end < newInterval.start) {
                // case 2: [old], [new]
                result.add(interval);
                insertPos++;
            } else {
                // case 3, 4: [old, new] or [new, old]
                newInterval.start = Math.min(newInterval.start, interval.start);
                newInterval.end = Math.max(newInterval.end, interval.end);
            }
        }

        result.add(insertPos, newInterval);

        return result;
    }
}
```

源码分析

源码的精华在case 3 和 case 4的处理，case 2用于确定最终新间隔的插入位置。

之所以不在 case 1立即返回，有两点考虑：一是代码的复杂性(需要用到 addAll 添加数组部分元素)；二是case2, case3, case 4有可能正好遍历到数组的最后一个元素，如果在 case 1就返回的话还需要单独做一判断。

复杂度分析

遍历一次，时间复杂度 $O(n)$. 不考虑作为结果返回占用的空间 result, 空间复杂度 $O(1)$.

Reference

- Insert Interval 参考程序 Java/C++/Python

Merge Intervals

Source

- leetcode: Merge Intervals | LeetCode OJ
- lintcode: (156) Merge Intervals

Problem

Given a collection of intervals, merge all overlapping intervals.

Example

Given intervals => merged intervals:

```
[                       [
  [1, 3],                 [1, 6],
  [2, 6],       =>        [8, 10],
  [8, 10],                [15, 18]
  [15, 18]              ]
]
```

Challenge

O(n log n) time and O(1) extra space.

题解1 - 排序后

初次接触这道题可能会先对 interval 排序，随后考虑相邻两个 interval 的 end 和 start 是否交叉，若交叉则合并之。

Java

```
/**
 * Definition of Interval:
 * public class Interval {
 *     int start, end;
 *     Interval(int start, int end) {
 *         this.start = start;
 *         this.end = end;
 *     }
 */

class Solution {
    /**
     * @param intervals: Sorted interval list.
     * @return: A new sorted interval list.
```

```
     */
    public List<Interval> merge(List<Interval> intervals) {
        if (intervals == null || intervals.isEmpty()) return intervals;

        List<Interval> result = new ArrayList<Interval>();
        // sort with Comparator
        Collections.sort(intervals, new IntervalComparator());
        Interval prev = intervals.get(0);
        for (Interval interval : intervals) {
            if (prev.end < interval.start) {
                result.add(prev);
                prev = interval;
            } else {
                prev.start = Math.min(prev.start, interval.start);
                prev.end = Math.max(prev.end, interval.end);
            }
        }
        result.add(prev);

        return result;
    }

    private class IntervalComparator implements Comparator<Interval> {
        public int compare(Interval a, Interval b) {
            return a.start - b.start;
        }
    }

}
```

源码分析

这里因为需要比较 interval 的 start, 所以需要自己实现 Comparator 接口并覆盖 compare 方法。这里取 prev 为前一个 interval。最后不要忘记加上 prev.

复杂度分析

排序 $O(n \log n)$, 遍历 $O(n)$, 所以总的时间复杂度为 $O(n \log n)$. 空间复杂度 $O(1)$.

题解2 - 插入排序

除了首先对 intervals 排序外，还可以使用类似插入排序的方法，插入的方法在题 Insert Interval 中已详述。这里将 result 作为 intervals 传进去即可，新插入的 interval 为 intervals 遍历得到的结果。

Java

```
/**
 * Definition of Interval:
 * public class Interval {
 *     int start, end;
 *     Interval(int start, int end) {
 *         this.start = start;
```

```
 *         this.end = end;
 *     }
 */

class Solution {
    /**
     * @param intervals: Sorted interval list.
     * @return: A new sorted interval list.
     */
    public List<Interval> merge(List<Interval> intervals) {
        if (intervals == null || intervals.isEmpty()) return intervals;

        List<Interval> result = new ArrayList<Interval>();
        for (Interval interval : intervals) {
            result = insert(result, interval);
        }

        return result;
    }

    private List<Interval> insert(List<Interval> intervals, Interval newInterval) {
        List<Interval> result = new ArrayList<Interval>();
        int insertPos = 0;
        for (Interval interval : intervals) {
            if (newInterval.end < interval.start) {
                result.add(interval);
            } else if (newInterval.start > interval.end) {
                result.add(interval);
                insertPos++;
            } else {
                newInterval.start = Math.min(newInterval.start, interval.start);
                newInterval.end = Math.max(newInterval.end, interval.end);
            }
        }
        result.add(insertPos, newInterval);

        return result;
    }
}
```

源码分析

关键在 insert 的理解， `result = insert(result, interval);` 作为迭代生成新的 result.

复杂度分析

每次添加新的 interval 都是线性时间复杂度，故总的时间复杂度为 $O(1+2+...+n)=O(n^2)$. 空间复杂度为 $O(n)$.

Reference

- Merge Intervals 参考程序 Java/C++/Python
- Soulmachine 的 leetcode 题解

Minimum Subarray

Source

- lintcode: (44) Minimum Subarray

```
Given an array of integers, find the subarray with smallest sum.

Return the sum of the subarray.

Example
For [1, -1, -2, 1], return -3

Note
The subarray should contain at least one integer.
```

题解

题 Maximum Subarray 的变形，使用区间和容易理解和实现。

Java

```java
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: A integer indicate the sum of minimum subarray
     */
    public int minSubArray(ArrayList<Integer> nums) {
        if (nums == null || nums.isEmpty()) return -1;

        int sum = 0, maxSum = 0, minSub = Integer.MAX_VALUE;
        for (int num : nums) {
            maxSum = Math.max(maxSum, sum);
            sum += num;
            minSub = Math.min(minSub, sum - maxSum);
        }

        return minSub;
    }
}
```

源码分析

略

复杂度分析

略

Matrix Zigzag Traversal

Source

- lintcode: (185) Matrix Zigzag Traversal

```
Given a matrix of m x n elements (m rows, n columns),
return all elements of the matrix in ZigZag-order.

Example
Given a matrix:

[
  [1, 2,  3,  4],
  [5, 6,  7,  8],
  [9,10, 11, 12]
]
return [1, 2, 5, 9, 6, 3, 4, 7, 10, 11, 8, 12]
```

题解

按之字形遍历矩阵，纯粹找下标规律。以题中所给范例为例，设 `(x, y)` 为矩阵坐标，按之字形遍历有如下规律：

```
(0, 0)
(0, 1), (1, 0)
(2, 0), (1, 1), (0, 2)
(0, 3), (1, 2), (2, 1)
(2, 2), (1, 3)
(2, 3)
```

可以发现其中每一行的坐标之和为常数，坐标和为奇数时 x 递增，为偶数时 x 递减。

Java - valid matrix index second

```java
public class Solution {
    /**
     * @param matrix: a matrix of integers
     * @return: an array of integers
     */
    public int[] printZMatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0) return null;

        int m = matrix.length - 1, n = matrix[0].length - 1;
        int[] result = new int[(m + 1) * (n + 1)];
        int index = 0;
        for (int i = 0; i <= m + n; i++) {
            if (i % 2 == 0) {
```

```
                for (int x = i; x >= 0; x--) {
                    // valid matrix index
                    if ((x <= m) && (i - x <= n)) {
                        result[index] = matrix[x][i - x];
                        index++;
                    }
                }
            } else {
                for (int x = 0; x <= i; x++) {
                    if ((x <= m) && (i - x <= n)) {
                        result[index] = matrix[x][i - x];
                        index++;
                    }
                }
            }
        }

        return result;
    }
}
```

Java - valid matrix index first

```
public class Solution {
    /**
     * @param matrix: a matrix of integers
     * @return: an array of integers
     */
    public int[] printZMatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0) return null;

        int m = matrix.length - 1, n = matrix[0].length - 1;
        int[] result = new int[(m + 1) * (n + 1)];
        int index = 0;
        for (int i = 0; i <= m + n; i++) {
            int upperBoundx = Math.min(i, m); // x <= m
            int lowerBoundx = Math.max(0, i - n); // lower bound i - x(y) <= n
            int upperBoundy = Math.min(i, n); // y <= n
            int lowerBoundy = Math.max(0, i - m); // i - y(x) <= m
            if (i % 2 == 0) {
                // column increment
                for (int y = lowerBoundy; y <= upperBoundy; y++) {
                    result[index] = matrix[i - y][y];
                    index++;
                }
            } else {
                // row increment
                for (int x = lowerBoundx; x <= upperBoundx; x++) {
                    result[index] = matrix[x][i - x];
                    index++;
                }
            }
        }

        return result;
    }
}
```

源码分析

矩阵行列和分奇偶讨论，奇数时行递增，偶数时列递增，一种是先循环再判断索引是否合法，另一种是先取的索引边界。

复杂度分析

后判断索引是否合法的实现遍历次数为 $1+2+...+(m+n)=O((m+n)^2)$, 首先确定上下界的每个元素遍历一次，时间复杂度 $O(m \cdot n)$. 空间复杂度都是 $O(1)$.

Reference

- LintCode/matrix-zigzag-traversal.cpp at master · kamyu104/LintCode

Valid Sudoku

Source

- leetcode: Valid Sudoku | LeetCode OJ
- lintcode: (389) Valid Sudoku

```
Determine whether a Sudoku is valid.

The Sudoku board could be partially filled,
where empty cells are filled with the character ..

Example
The following partially filed sudoku is valid.
```

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
Valid Sudoku

Note
A valid Sudoku board (partially filled) is not necessarily solvable.
Only the filled cells need to be validated.
Clarification
What is Sudoku?

http://sudoku.com.au/TheRules.aspx
https://zh.wikipedia.org/wiki/%E6%95%B8%E7%8D%A8
https://en.wikipedia.org/wiki/Sudoku
http://baike.baidu.com/subview/961/10842669.htm
```

题解

看懂数独的含义就好了，分为三点考虑，一是每行无重复数字；二是每列无重复数字；三是小的九宫格中无重复数字。

Java

```
class Solution {
    /**
      * @param board: the board
        @return: wether the Sudoku is valid
      */
    public boolean isValidSudoku(char[][] board) {
        if (board == null || board.length == 0) return false;

        // check row
        for (int i = 0; i < 9; i++) {
            boolean[] numUsed = new boolean[9];
            for (int j = 0; j < 9; j++) {
                if (isDuplicate(board[i][j], numUsed)) {
                    return false;
                }
            }
        }

        // check column
        for (int i = 0; i < 9; i++) {
            boolean[] numUsed = new boolean[9];
            for (int j = 0; j < 9; j++) {
                if (isDuplicate(board[j][i], numUsed)) {
                    return false;
                }
            }
        }

        // check sub box
        for (int i = 0; i < 9; i = i + 3) {
            for (int j = 0; j < 9; j = j + 3) {
                if (!isValidBox(board, i, j)) {
                    return false;
                }
            }
        }

        return true;
    }

    private boolean isValidBox(char[][] box, int x, int y) {
        boolean[] numUsed = new boolean[9];
        for (int i = x; i < x + 3; i++) {
            for (int j = y; j < y + 3; j++) {
                if (isDuplicate(box[i][j], numUsed)) {
                    return false;
                }
            }
        }
        return true;
    }

    private boolean isDuplicate(char c, boolean[] numUsed) {
        if (c == '.') {
            return false;
        } else if (numUsed[c - '1']) {
```

```
            return true;
        } else {
            numUsed[c - '1'] = true;
            return false;
        }
    }
}
```

源码分析

首先实现两个小的子功能模块判断是否有重复和小的九宫格是否重复。

复杂度分析

略

Reference

- Soulmachine 的 leetcode 题解

Add Binary

Source

- leetcode: Add Binary | LeetCode OJ
- lintcode: (408) Add Binary

```
Given two binary strings, return their sum (also a binary string).

For example,
a = "11"
b = "1"
Return "100".
```

题解

用字符串模拟二进制的加法，加法操作一般使用自后往前遍历的方法，不同位大小需要补零。

Java

```java
public class Solution {
    /**
     * @param a a number
     * @param b a number
     * @return the result
     */
    public String addBinary(String a, String b) {
        if (a == null || a.length() == 0) return b;
        if (b == null || b.length() == 0) return a;

        StringBuilder sb = new StringBuilder();
        int aLen = a.length(), bLen = b.length();

        int carry = 0;
        for (int ia = aLen - 1, ib = bLen - 1; ia >= 0 || ib >= 0; ia--, ib--) {
            // replace with 0 if processed
            int aNum = (ia < 0) ? 0 : a.charAt(ia) - '0';
            int bNum = (ib < 0) ? 0 : b.charAt(ib) - '0';

            int num = (aNum + bNum + carry) % 2;
            carry = (aNum + bNum + carry) / 2;
            sb.append(num);
        }
        if (carry == 1) sb.append(1);

        // important!
        sb.reverse();
        String result = sb.toString();
        return result;
    }
```

```
}
```

源码分析

用到的技巧主要有两点，一是两个数位数大小不一时用0补上，二是最后需要判断最高位的进位是否为1。最后需要反转字符串，因为我们是从低位往高位迭代的。虽然可以使用 insert 避免最后的 reverse 操作，但如此一来时间复杂度就从 $O(n)$ 变为 $O(n^2)$ 了。

复杂度分析

遍历两个字符串，时间复杂度 $O(n)$. reverse 操作时间复杂度 $O(n)$, 故总的时间复杂度 $O(n)$. 使用了 StringBuilder 作为临时存储对象，空间复杂度 $O(n)$.

Reverse Integer

Source

- leetcode: Reverse Integer | LeetCode OJ
- lintcode: (413) Reverse Integer

Problem

Reverse digits of an integer. Returns 0 when the reversed integer overflows (signed 32-bit integer).

Example

Given x = 123, return 321

Given x = -123, return -321

题解

初看这道题觉得先将其转换为字符串然后转置以下就好了，但是仔细一想这种方法存在两种缺陷，一是负号需要单独处理，而是转置后开头的0也需要处理。另一种方法是将原数字逐个弹出，然后再将弹出的数字组装为新数字，咋看以为需要用到栈，实际上却是队列... 所以根本不需要辅助数据结构。关于正负号的处理，我最开始是单独处理的，后来看其他答案时才发现根本就不用分正负考虑。因为`-1 / 10 = 0`.

Java

```
public class Solution {
    /**
     * @param n the integer to be reversed
     * @return the reversed integer
     */
    public int reverseInteger(int n) {
        long result = 0;
        while (n != 0) {
            result = n % 10 + 10 * result;
            n /= 10;
        }

        if (result < Integer.MIN_VALUE || result > Integer.MAX_VALUE) {
            return 0;
        }
        return (int)result;
    }
}
```

源码分析

注意 lintcode 和 leetcode 的方法名不一样。使用 long 型保存中间结果，最后判断是否溢出。

Reference

- LeetCode-Sol-Res/ReverseInt.java at master · FreeTymeKiyan/LeetCode-Sol-Res

Gray Code

Source

- leetcode: Gray Code | LeetCode OJ
- lintcode: (411) Gray Code

Problem

The gray code is a binary numeral system where two successive values differ in only one bit. Given a non-negative integer n representing the total number of bits in the code, find the sequence of gray code. A gray code sequence must begin with 0 and with cover all 2^n integers.

Example

Given `n = 2`, return `[0,1,3,2]`. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note

For a given n, a gray code sequence is not uniquely defined.

`[0,2,3,1]` is also a valid gray code sequence according to the above definition.

Challenge

$O(2^n)$ time.

题解

第一次遇到这个题是在腾讯的在线笔试中，当时找到了规律，用的是递归，但是实现似乎有点问题... 直接从 n 位的格雷码分析不太好分析，比如题中 `n = 2` 的格雷码，我们不妨试试从小到大分析，以 `n = 1` 往后递推。

从图中我们可以看出n 位的格雷码可由 n-1位的格雷码递推，在最高位前顺序加0，逆序加1即可。实际实现时我们可以省掉在最高位加0的过程，因为其在数值上和前 n-1位格雷码相同。另外一点则是初始化的处理，图中为从1开始，但若从0开始可进一步简化程序。而且根据 格雷码 的定义，n=0时确实应该返回0.

Java

```
public class Solution {
    /**
     * @param n a number
     * @return Gray code
     */
    public ArrayList<Integer> grayCode(int n) {
        if (n < 0) return null;

        ArrayList<Integer> currGray = new ArrayList<Integer>();
        currGray.add(0);
```

```
        for (int i = 0; i < n; i++) {
            int msb = 1 << i;
            // backward - symmetry
            for (int j = currGray.size() - 1; j >= 0; j--) {
                currGray.add(msb | currGray.get(j));
            }
        }

        return currGray;
    }
}
```

源码分析

加0 的那一部分已经在前一组格雷码中出现，故只需将前一组格雷码镜像后在最高位加1即可。第二重 for 循环中需要注意的是 `currGray.size() - 1` 并不是常量，只能用于给 j 初始化。本应该使用 2^n 和上一组格雷码相加，这里考虑到最高位为1的特殊性，使用位运算模拟加法更好。

复杂度分析

生成n 位的二进制码，时间复杂度 $O(2^n)$, 使用了 `msb` 代表最高位的值便于后续相加，空间复杂度 $O(1)$.

Reference

- Soulmachine 的 leetcode 题解

Part III - Contest

本节主要总结一些如 Google APAC 等比赛的题。

Google APAC

本章总结 Google APAC 的一些题。

APAC 2015 Round B

- Dashboard - Round B APAC Test - Google Code Jam

Problem A. Password Attacker

Source

- Dashboard - Round B APAC Test - Problem A. Password Attacker

Problem

Passwords are widely used in our lives: for ATMs, online forum logins, mobile device unlock and door access. Everyone cares about password security. However, attackers always find ways to steal our passwords. Here is one possible situation:

Assume that Eve, the attacker, wants to steal a password from the victim Alice. Eve cleans up the keyboard beforehand. After Alice types the password and leaves, Eve collects the fingerprints on the keyboard. Now she knows which keys are used in the password. However, Eve won't know how many times each key has been pressed or the order of the keystroke sequence.

To simplify the problem, let's assume that Eve finds Alice's fingerprints only occurs on M keys. And she knows, by another method, that Alice's password contains N characters. Furthermore, every keystroke on the keyboard only generates a single, unique character. Also, Alice won't press other irrelevant keys like 'left', 'home', 'backspace' and etc.

Here's an example. Assume that Eve finds Alice's fingerprints on M=3 key '3', '7' and '5', and she knows that Alice's password is N=4-digit in length. So all the following passwords are possible: 3577, 3557, 7353 and 5735. (And, in fact, there are 32 more possible passwords.)

However, these passwords are not possible:

```
1357  // There is no fingerprint on key '1'
3355  // There is fingerprint on key '7',
         so '7' must occur at least once.
357   // Eve knows the password must be a 4-digit number.
```

With the information, please count that how many possible passwords satisfy the statements above. Since the result could be large, please output the answer modulo 1000000007(109+7).

Input

The first line of the input gives the number of test cases, T. For the next T lines, each contains two space-separated numbers M and N, indicating a test case.

Output

For each test case, output one line containing "Case #x: y", where x is the test case number (starting from 1) and y is the total number of possible passwords modulo 1000000007(109+7).

Limits

Small dataset

T = 15. 1 ≤ M ≤ N ≤ 7.

Large dataset

T = 100. 1 ≤ M ≤ N ≤ 100.

Smaple

```
Input    Output

4
1 1      Case #1: 1
3 4      Case #2: 36
5 5      Case #3: 120
15 15    Case #4: 674358851
```

题解

题目看似很长，其实简单来讲就是用 M 个 不同的字符组成长度为 N 的字符串，问有多少种不同的排列。这里 M 小于 N，要是大于的话就是纯排列了。这道题我最开始想用纯数学方法推导公式一步到位，实践下来发现这种想法真是太天真了，这不是数学竞赛... 即使用推导也应该是推导类似动态规划的状态转移方程。

这里的动态规划不太明显，我们以状态 `dp[m][n]` 表示用 m 个不同的字符能组成长度为 n 的不同字符串的个数。这里需要注意的是最后长度为 n 的字符串中必须包含 m 个不同的字符，不多也不少。接下来就是寻找状态转移方程了，之前可能的状态为 `dp[m - 1][n -1], dp[m - 1][n], dp[m][n - 1]` . 现在问题来了，怎么解释这些状态以寻找状态转移方程？常规方法为正向分析，即分析 `m ==> n` , 但很快我们可以发现 `dp[m - 1][n]` 这个状态很难处理。既然正向分析比较麻烦，我们不妨试试反向从 `n ==> m` 分析，可以发现字符串个数由 n 变为 n-1，这减少的字符可以分为两种情况，一种是这个减少的字符就在前 n - 1个字符中，另一种则不在，如此一来便做到了不重不漏。相应的状态转移方程为：

```
dp[i][j] = dp[m][n-1] * m + dp[m - 1][n - 1] * m
```

第一种和第二种情况下字符串的第 n 位均可由 m 个字符中的一个填充。初始化分两种情况，第一种为索引为0时，其值显然为0；第二种则是 m 为1时，容易知道相应的排列为1。最后返回 `dp[M][N]` .

Java

```java
import java.util.*;

public class Solution {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
```

```
        int T = in.nextInt();
        // System.out.println("T = " + T);
        for (int t = 1; t <= T; t++) {
            int M = in.nextInt(), N = in.nextInt();
            long ans = solve(M, N);
            // System.out.printf("M = %d, N = %d\n", M, N);
            System.out.printf("Case #%d: %d\n", t, ans);
        }
    }

    public static long solve(int M, int N) {
        long[][] dp = new long[1 + M][1 + N];
        long mod = 1000000007;
        for (int j = 1; j <= N; j++) {
            dp[1][j] = 1;
        }
        for (int i = 2; i <= M; i++) {
            for (int j = i; j <= N; j++) {
                dp[i][j] = i * (dp[i][j - 1] + dp[i - 1][j - 1]);
                dp[i][j] %= mod;
            }
        }

        return dp[M][N];
    }
}
```

源码分析

Google Code Jam 上都是自己下载输入文件，上传结果，这里我们使用输入输出重定向的方法解决这个问题。举个例子，将这段代码保存为 `Solution.java`，将标准输入重定向至输入文件，标准输出重定向至输出文件。编译好之后以如下方式运行：

```
java Solution < A-large-practice.in > A-large-practice.out
```

这种方式处理各种不同 OJ 平台的输入输出较为方便。

复杂度分析

时间复杂度 $O(mn)$, 空间复杂度 $O(mn)$.

Reference

- Google-APAC2015-"Password Attacker" - dmsehuang的专栏

Appendix I Interview and Resume

本章主要总结一些技术面试和撰写简历方面的注意事项。

Interview

本小节主要总结一些面试相关的优质资源。

Facebook workshop - Crush Your Coding Interview

Facebook 每年的5月份左右会在中国大陆的清北复交浙等高校做技术讲座，基本模式是两到三个工程师进行现场分享，Frank 会着重介绍一些面试流程和简历撰写的细节，信息量非常大！其他几个工程师则是介绍自己在 Facebook 所做的产品和企业文化，全程约两个半小时，后面是 Q & A 环节，对提问者有各种小礼物送出。我会说我拿到了F的官方T恤了吗 :) 质地还不错，布料摸起来比较舒服，logo 也不太明显。强烈推荐在这五所高校附近的 CSer 们前去围观！本校的就更不要错过了啦~

咳咳，进入正题，以下为自己对当晚 Facebook 工程师经验分享的一些总结，部分参考自浙大一位童鞋的总结[Facebook 交流]。

大致的 slides 如下，没有在网上找到公开的，以下是自己根据照片总结的。

Resume

What to include on your resume

1. University, degree, expected graduation date
    - Highly recommended including GPA with scale/ranking
2. Projects
    - Industry experience (internships, competition, full-time)
    - Interesting projects
    - Links where applicable (github, apps, websites)

学校/学位/毕业时间(方便 HR 知道你何时毕业筛选简历)，GPA 最好能附上权重，不同的学校 GPA 总分不一样。

Writing a great resume

1. Focus on what you did
2. Focus on Impact(metrics and numbers are a plus)
3. Be specific and concise (1 page if at all possible)
4. Pro tip: alawys start with an active verb
    - example: built, optimized, improved, doubled, etc
5. Don't include
    - Age, photo, ID number

提供客观数据，具体且简短，多使用动词如『优化』、『提高』等，不要在简历中包含年龄，照片，ID 号，有些东西与法律相关。

Coding interview

Goals of a coding interview

Protip: Think out loud!

1. How you think and tackle technical problems
2. How you consider engineering trade offs (speed vs. time)
3. How you communicate in English about codes
4. Limits of what you know
    - Don't feel bad if you don't get all answers right

What is covered?

Use your comfortable coding language (C++ Java would be better)

之前听 Google 的工程师说是尽量使用 C++ 和 Java 实现。

1. Data structures and algorithms
    - implement, not memorize
    - discuss complexity (space and time trade-offs)
    - Common library functions are fair game
2. Specific questions about concepts are rare
    - Unless you claim to be an export or need the concept

During the interview

1. Clarify your understanding
    - ask questions until you fully understand problem space and constraints
    - validate or state any assumptions
    - draw pictures to help you better understand problems
2. Focus on getting a working solution first
    - handle corner cases
3. Iterate

1. 举一两个例子，有可能的话还可以在白板上画出来帮助理解。问题的限制不是那么明确，确定和面试官理解的是同一个问题。
2. 尝试获得一个能工作的 code
3. 进行迭代，寻找更好的方法。记住测试自己的代码，选择简单但是典型的测试案例。

不要立即写代码，先明确思路，再写代码。Done is better than perfect

能否修改原数组，空间限制，时间限制。

大体方案要和面试官讨论。一定要和面试官多交流，思考过程和方法。

be yourself, 坦白地说出自己不懂的地方，没什么不好的，把知道的地方说清楚。

最近做的/最喜欢的/最具挑战性的项目是什么，不只是要把项目背景说出来，还要说出为什么喜欢，有哪些挑战，推理过程。

项目讨论的框架

1. context: 简要描述项目背景，为什么要做，意义和影响何在。让面试官快速了解。
2. action: 你在这个项目中做了什么，贡献是什么。
3. result: 项目的结果，失败的项目也可以讲，在这个项目中学到了什么，得到了什么样的成长。

简历中提到的技术一定要熟悉。站在面试官的角度问自己会问自己什么问题。

面试之后，可以问面试官问题，着重问自己关心的问题。

behavior question

1. motivation：动机从何而来，整个过程中做了什么。
2. passion: 激情，哪种产品让你特别兴奋，为什么。
3. team pair: 团队合作? 这里忘了
4. disagreement: 怎么处理不同意见和冲突。

回答要具体，跟自己有关系，而不是泛泛而谈。

总结

1. Think out loud, 不用担心自己的英语，把主要意思表达清楚就好了.
2. 面试中多问问题，充分理解题意。
3. 不要写 shit code, 提供典型案例测试自己的代码
4. 多练习，可以找几个小伙伴进行模拟面试，交换角色，在白板上多写代码。
5. 电话面试找一个安静的地方，把双手解放出来，便于写代码。

Reference

本小节部分摘自九章微信的分享。

- www.geeksforgeeks.org - 非常著名的漏题网站之一。上面会时不时的有各种公司的面试真题漏出。有一些题也会有解法分析。
- Programming Interview Questions | CareerCup - CC150作者搞的网站，也是著名的漏题网站之一。大家会在上面讨论各个公司的面试题。
- Glassdoor – Get Hired. Love Your Job. - 一个给公司打分的网站，类似yelp的公司版。会有一些人在上面讨论面试题，适合你在面某个公司的时候专门去看一下。
- 面经网 | 汇集热气腾腾的求职咨询 - 面经网。应该是个人经营的一个积累面经的网站。面经来源主要是一亩三分地，mitbbs之类的地方。
- 一亩三分地论坛-美国加拿大留学申请|工作就业|英语考试|学习生活信噪比最高的网站 - 人气非常高的论坛。
- 待字闺中(JobHunting)版 | 未名空间(mitbbs.com) jobhunting版，美华人找工作必上。
- 程序员面试：电话面试问答Top 50 - 博客 - 伯乐在线 - 其实不仅仅只是 Top 50，扩展连接还给出了其他参考。
- 想加入硅谷顶级科技公司，你该知道这些 - 数据工程师董飞的求职分享，涵盖硅谷公司的招聘流程，简历的书写，面试中的考察内容，选拔标准等。Evernote 备份链接
- 求职在美国，面试攻略我知道 on Vimeo - Coursera 数据工程师董飞的视频分享。

- > Facebook 交流. Facebook学长交流分享 - biaobiaoqi - 博客园 - Facebook 工程师的经验分享，Frank 对面试和简历部分的分享极其详细，信息量很大。 ↩

Resume

本小节主要总结一些**技术简历**相关的优质资源。具体的还可以参考前一节中 Facebook 提供的简历撰写指南，除了这些简短的资源外还强烈推荐下 Gayle 写的 *The Google Resume*，极其详细！干货超多！

Resume Template

推荐使用 Markdown 或者 Latex 撰写简历，可以使用 sharelatex 在线写简历，从 CV or Resume 模板中挑，modernCV 的那些模板要写在一页里比较困难，这个 Professional CV 相对紧凑一些，LaTeX Templates » Curricula Vitae/Résumés 上还有更多更好的选择。另外推荐下自己写的一个还算简洁优雅的简历模板——billryan/resume, 同时支持中英文和 FontAwesome 字体，欢迎试用~ 中文的样式大概长成下面这个样子。

你的大名

xxx@yuanbin.me · (+86) 131-221-87xxx · http://www.yuanbin.me

教育背景

上海交通大学, 上海 2013 – 至今

在读硕士研究生 信息与通信工程, 预计 2016 年 3 月毕业

西安电子科技大学, 西安, 陕西 2009 – 2013

学士 通信工程

实习/项目经历

黑科技公司上海 2015 年 3 月 – 2015 年 5 月

实习 经理: 高富帅

xxx 后端开发

- 实现了 xxx 特性
- 后台资源占用率减少 8%
- xxx

分布式科学上网姿势 2014 年 6 月 – 至今

Golang, Linux 个人项目，和富帅糕合作开发

分布式负载均衡科学上网姿势, https://github.com/cyfdecyf/cow

- 修复了连接未正常关闭导致文件描述符耗尽的 bug
- 使用 Chord 哈希 URL, 实现稳定可靠地分流
- xxx (尽量使用量化的客观结果)

LaTeX 简历模板 2015 年 5 月 – 至今

LaTeX, Python 个人项目

优雅的 LaTeX 简历模板, https://github.com/billryan/resume

- 容易定制和扩展
- 完善的 Unicode 字体支持，使用 XeLaTeX 编译
- 支持 FontAwesome 4.3.0

IT 技能

- 编程语言: C == Python > C++ > Java
- 平台: Linux
- 开发: xxx

获奖情况

第一名, xxx 比赛 2013 年 6 月

其他奖项 2015

其他

- 技术博客: http://blog.yours.me
- GitHub: https://github.com/username
- 语言: 英语 - 熟练 (TOEFL xxx)

Reference

- *The Google Resume* - 书名虽为简历，但本书可不只是教你写简历那么简单，除了教你如何写优秀简历外还总结了技术面试以及找工作过程中的方方面面。甚至连职业规划都有涉及！**力荐！**
- 如何写好技术简历 —— 实例、模板及工具 | @Get社区 - 挺不错的技术简历实战。
- 精益技术简历之道——改善技术简历的47条原则 - Lucida - 『Effective 简历』系列。
- 如何把简历写进一页 - V2EX - 众人支招助萌妹纸优化简历。
- *Cracking the coding interview* 『写好简历』一节。

术语表

BFS

Breadth-First Search, 广度优先搜索

12.5. Binary Tree Level Order Traversal II 14. Exhaustive Search 16.1. Topological Sorting
16.2. Word Ladder

DFS

Depth-First Search, 深度优先搜索

15.22. Longest Increasing Continuous subsequence
15.23. Longest Increasing Continuous subsequence II 15.3. Minimum Path Sum 15.1. Triangle
14. Exhaustive Search 14.15. Minimum Depth of Binary Tree 14.11. Palindrome Partitioning
14.1. Subsets 14.7. Unique Binary Search Trees II 16.1. Topological Sorting
10.14. Print Numbers by Recursion 7.11. Wildcard Matching

DP_Matrix

根据动态规划解题的四要素，矩阵类动态规划问题通常可用 f[x][y] 表示从起点走到坐标(x,y)的值

15.3. Minimum Path Sum 15.4. Unique Paths 15.5. Unique Paths II

DP_Sequence

单序列动态规划，通常使用 f[i] 表示前i个位置/数字/字母... 使用 f[n-1] 表示最后返回结果。

15. Dynamic Programming 15.9. Longest Increasing Subsequence
15.10. Palindrome Partitioning II 15.8. Word Break

DP_Two_Sequence

一般有两个数组或者两个字符串，计算其匹配关系. 通常可用 `f[i][j]`表示第一个数组的前 i 位和第二个数组的前 j 位的关系。

15.11. Longest Common Subsequence 15. Dynamic Programming 15.18. Distinct Subsequences
15.12. Edit Distance

TLE

Time Limit Exceeded 的简称。你的程序在 OJ 上的运行时间太长了，超过了对应题目的时间限制。

15.17. Best Time to Buy and Sell Stock IV　15.7. Jump Game　15.13. Jump Game II
15.3. Minimum Path Sum　15.10. Palindrome Partitioning II　16.2. Word Ladder
8.2. Zero Sum Subarray　11.15. Copy List with Random Pointer　11.17. Insertion Sort List
11.13. Merge k Sorted Lists　11.14. Reorder List　10.10. Hash Function　10.19. Ugly Number
7.9. Longest Palindromic Substring　7.4. Anagrams