

目录

[Preface 3](#_Toc1)

[Part I - Basics 4](#_Toc2)

[Basics Data Structure 5](#_Toc3)

[String 5](#_Toc4)

[Linked List 5](#_Toc5)

[Binary Tree 5](#_Toc6)

[Huffman Compression 5](#_Toc7)

[Queue 5](#_Toc8)

[Heap 5](#_Toc9)

[Stack 5](#_Toc10)

[Set 5](#_Toc11)

[Map 5](#_Toc12)

[Graph 5](#_Toc13)

[Basics Sorting 6](#_Toc14)

[Bubble Sort 6](#_Toc15)

[Selection Sort 6](#_Toc16)

[Insertion Sort 6](#_Toc17)

[Merge Sort 6](#_Toc18)

[Quick Sort 6](#_Toc19)

[Heap Sort 6](#_Toc20)

[Bucket Sort 6](#_Toc21)

[Counting Sort 6](#_Toc22)

[Radix Sort 6](#_Toc23)

[Basics Algorithm 7](#_Toc24)

[Divide and Conquer 7](#_Toc25)

[Binary Search 7](#_Toc26)

[Math 7](#_Toc27)

[Greatest Common Divisor 7](#_Toc28)

[Prime 7](#_Toc29)

[Knapsack 7](#_Toc30)

[Probability 7](#_Toc31)

[Shuffle 7](#_Toc32)

[Basics Misc 8](#_Toc33)

[Bit Manipulation 8](#_Toc34)

[Part II - Coding 9](#_Toc35)

[String 10](#_Toc36)

[strStr 10](#_Toc37)

[Two Strings Are Anagrams 10](#_Toc38)

[Compare Strings 10](#_Toc39)

[Anagrams 10](#_Toc40)

[Longest Common Substring 10](#_Toc41)

[Rotate String 10](#_Toc42)

[Reverse Words in a String 10](#_Toc43)

[Valid Palindrome 10](#_Toc44)

[Longest Palindromic Substring 10](#_Toc45)

[Space Replacement 10](#_Toc46)

[Wildcard Matching 10](#_Toc47)

[Length of Last Word 10](#_Toc48)

[Count and Say 10](#_Toc49)

[Integer Array 11](#_Toc50)

[Remove Element 11](#_Toc51)

[Zero Sum Subarray 11](#_Toc52)

[Subarray Sum K 11](#_Toc53)

[Subarray Sum Closest 11](#_Toc54)

[Recover Rotated Sorted Array 11](#_Toc55)

[Product of Array Exclude Itself 11](#_Toc56)

[Partition Array 11](#_Toc57)

[First Missing Positive 11](#_Toc58)

[2 Sum 11](#_Toc59)

[3 Sum 11](#_Toc60)

[3 Sum Closest 11](#_Toc61)

[Remove Duplicates from Sorted Array 11](#_Toc62)

[Remove Duplicates from Sorted Array II 11](#_Toc63)

[Merge Sorted Array 11](#_Toc64)

[Merge Sorted Array II 11](#_Toc65)

[Median 11](#_Toc66)

[Partition Array by Odd and Even 11](#_Toc67)

[Kth Largest Element 11](#_Toc68)

[Binary Search 12](#_Toc69)

[Binary Search 12](#_Toc70)

[Search Insert Position 12](#_Toc71)

[Search for a Range 12](#_Toc72)

[First Bad Version 12](#_Toc73)

[Search a 2D Matrix 12](#_Toc74)

[Search a 2D Matrix II 12](#_Toc75)

[Find Peak Element 12](#_Toc76)

[Search in Rotated Sorted Array 12](#_Toc77)

[Search in Rotated Sorted Array II 12](#_Toc78)

[Find Minimum in Rotated Sorted Array 12](#_Toc79)

[Find Minimum in Rotated Sorted Array II 12](#_Toc80)

[Median of two Sorted Arrays 12](#_Toc81)

[Sqrt x 12](#_Toc82)

[Wood Cut 12](#_Toc83)

[Math and Bit Manipulation 13](#_Toc84)

[Single Number 13](#_Toc85)

[Single Number II 13](#_Toc86)

[Single Number III 13](#_Toc87)

[O1 Check Power of 2 13](#_Toc88)

[Convert Integer A to Integer B 13](#_Toc89)

[Factorial Trailing Zeroes 13](#_Toc90)

[Unique Binary Search Trees 13](#_Toc91)

[Update Bits 13](#_Toc92)

[Fast Power 13](#_Toc93)

[Hash Function 13](#_Toc94)

[Count 1 in Binary 13](#_Toc95)

[Fibonacci 13](#_Toc96)

[A plus B Problem 13](#_Toc97)

[Print Numbers by Recursion 13](#_Toc98)

[Majority Number 13](#_Toc99)

[Majority Number II 13](#_Toc100)

[Majority Number III 13](#_Toc101)

[Digit Counts 13](#_Toc102)

[Ugly Number 13](#_Toc103)

[Plus One 13](#_Toc104)

[Linked List 14](#_Toc105)

[Remove Duplicates from Sorted List 14](#_Toc106)

[Remove Duplicates from Sorted List II 14](#_Toc107)

[Remove Duplicates from Unsorted List 14](#_Toc108)

[Partition List 14](#_Toc109)

[Two Lists Sum 14](#_Toc110)

[Two Lists Sum Advanced 14](#_Toc111)

[Remove Nth Node From End of List 14](#_Toc112)

[Linked List Cycle 14](#_Toc113)

[Linked List Cycle II 14](#_Toc114)

[Reverse Linked List 14](#_Toc115)

[Reverse Linked List II 14](#_Toc116)

[Merge Two Sorted Lists 14](#_Toc117)

[Merge k Sorted Lists 14](#_Toc118)

[Reorder List 14](#_Toc119)

[Copy List with Random Pointer 14](#_Toc120)

[Sort List 14](#_Toc121)

[Insertion Sort List 14](#_Toc122)

[Check if a singly linked list is palindrome 14](#_Toc123)

[Delete Node in the Middle of Singly Linked List 14](#_Toc124)

[Rotate List 14](#_Toc125)

[Swap Nodes in Pairs 14](#_Toc126)

[Remove Linked List Elements 14](#_Toc127)

[Binary Tree 15](#_Toc128)

[Binary Tree Preorder Traversal 15](#_Toc129)

[Binary Tree Inorder Traversal 15](#_Toc130)

[Binary Tree Postorder Traversal 15](#_Toc131)

[Binary Tree Level Order Traversal 15](#_Toc132)

[Binary Tree Level Order Traversal II 15](#_Toc133)

[Maximum Depth of Binary Tree 15](#_Toc134)

[Balanced Binary Tree 15](#_Toc135)

[Binary Tree Maximum Path Sum 15](#_Toc136)

[Lowest Common Ancestor 15](#_Toc137)

[Invert Binary Tree 15](#_Toc138)

[Diameter of a Binary Tree 15](#_Toc139)

[Construct Binary Tree from Preorder and Inorder Traversal 15](#_Toc140)

[Construct Binary Tree from Inorder and Postorder Traversal 15](#_Toc141)

[Subtree 15](#_Toc142)

[Binary Tree Zigzag Level Order Traversal 15](#_Toc143)

[Binary Tree Serialization 15](#_Toc144)

[Binary Search Tree 16](#_Toc145)

[Insert Node in a Binary Search Tree 16](#_Toc146)

[Validate Binary Search Tree 16](#_Toc147)

[Search Range in Binary Search Tree 16](#_Toc148)

[Convert Sorted Array to Binary Search Tree 16](#_Toc149)

[Convert Sorted List to Binary Search Tree 16](#_Toc150)

[Binary Search Tree Iterator 16](#_Toc151)

[Exhaustive Search 17](#_Toc152)

[Subsets 17](#_Toc153)

[Unique Subsets 17](#_Toc154)

[Permutations 17](#_Toc155)

[Unique Permutations 17](#_Toc156)

[Next Permutation 17](#_Toc157)

[Previous Permuation 17](#_Toc158)

[Unique Binary Search Trees II 17](#_Toc159)

[Permutation Index 17](#_Toc160)

[Permutation Index II 17](#_Toc161)

[Permutation Sequence 17](#_Toc162)

[Palindrome Partitioning 17](#_Toc163)

[Combinations 17](#_Toc164)

[Combination Sum 17](#_Toc165)

[Combination Sum II 17](#_Toc166)

[Minimum Depth of Binary Tree 17](#_Toc167)

[Word Search 17](#_Toc168)

[Dynamic Programming 18](#_Toc169)

[Triangle 18](#_Toc170)

[Backpack 18](#_Toc171)

[Backpack II 18](#_Toc172)

[Minimum Path Sum 18](#_Toc173)

[Unique Paths 18](#_Toc174)

[Unique Paths II 18](#_Toc175)

[Climbing Stairs 18](#_Toc176)

[Jump Game 18](#_Toc177)

[Word Break 18](#_Toc178)

[Longest Increasing Subsequence 18](#_Toc179)

[Palindrome Partitioning II 18](#_Toc180)

[Longest Common Subsequence 18](#_Toc181)

[Edit Distance 18](#_Toc182)

[Jump Game II 18](#_Toc183)

[Best Time to Buy and Sell Stock 18](#_Toc184)

[Best Time to Buy and Sell Stock II 18](#_Toc185)

[Best Time to Buy and Sell Stock III 18](#_Toc186)

[Best Time to Buy and Sell Stock IV 18](#_Toc187)

[Distinct Subsequences 18](#_Toc188)

[Interleaving String 18](#_Toc189)

[Maximum Subarray 18](#_Toc190)

[Maximum Subarray II 18](#_Toc191)

[Longest Increasing Continuous subsequence 18](#_Toc192)

[Longest Increasing Continuous subsequence II 18](#_Toc193)

[Graph 19](#_Toc194)

[Find the Connected Component in the Undirected Graph 19](#_Toc195)

[Route Between Two Nodes in Graph 19](#_Toc196)

[Topological Sorting 19](#_Toc197)

[Word Ladder 19](#_Toc198)

[Bipartial Graph Part I 19](#_Toc199)

[Data Structure 20](#_Toc200)

[Implement Queue by Two Stacks 20](#_Toc201)

[Min Stack 20](#_Toc202)

[Sliding Window Maximum 20](#_Toc203)

[Longest Words 20](#_Toc204)

[Heapify 20](#_Toc205)

[Problem Misc 21](#_Toc206)

[Nuts and Bolts Problem 21](#_Toc207)

[String to Integer 21](#_Toc208)

[Insert Interval 21](#_Toc209)

[Merge Intervals 21](#_Toc210)

[Minimum Subarray 21](#_Toc211)

[Matrix Zigzag Traversal 21](#_Toc212)

[Valid Sudoku 21](#_Toc213)

[Add Binary 21](#_Toc214)

[Reverse Integer 21](#_Toc215)

[Gray Code 21](#_Toc216)

[Find the Missing Number 21](#_Toc217)

[Minimum Window Substring 21](#_Toc218)

[Continuous Subarray Sum 21](#_Toc219)

[Continuous Subarray Sum II 21](#_Toc220)

[Longest Consecutive Sequence 21](#_Toc221)

[Part III - Contest 22](#_Toc222)

[Google APAC 23](#_Toc223)

[APAC 2015 Round B 23](#_Toc224)

[Problem A. Password Attacker 23](#_Toc225)

[Microsoft 24](#_Toc226)

[Microsoft 2015 April 24](#_Toc227)

[Problem A. Magic Box 24](#_Toc228)

[Problem B. Professor Q's Software 24](#_Toc229)

[Problem C. Islands Travel 24](#_Toc230)

[Problem D. Recruitment 24](#_Toc231)

[Microsoft 2015 April 2 24](#_Toc232)

[Problem A. Lucky Substrings 24](#_Toc233)

[Problem B. Numeric Keypad 24](#_Toc234)

[Problem C. Spring Outing 24](#_Toc235)

[Microsoft 2015 September 2 24](#_Toc236)

[Problem A. Farthest Point 24](#_Toc237)

[Appendix I Interview and Resume 25](#_Toc238)

[Interview 25](#_Toc239)

[Resume 25](#_Toc240)

# Preface

# 数据结构与算法/leetcode/lintcode题解

## 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](https://www.gitbook.com/book/yuanbin/algorithm/details) 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](https://raw.githubusercontent.com/billryan/algorithm-exercise/deploy/epub/algorithm-ebook_zh-cn.epub), [Gitbook](https://www.gitbook.com/download/epub/book/yuanbin/algorithm?lang=zh-cn), [GitCafe(中国大陆用户适用)](https://gitcafe.com/billryan/algorithm-exercise/raw/deploy/epub/algorithm-ebook_zh-cn.epub) - 适合在 iPhone/iPad/MAC 上离线查看，实测效果极好。
2. PDF: [GitHub](https://raw.githubusercontent.com/billryan/algorithm-exercise/deploy/pdf/algorithm-ebook_zh-cn.pdf), [Gitbook](https://www.gitbook.com/download/pdf/book/yuanbin/algorithm?lang=zh-cn), [GitCafe(中国大陆用户适用)](https://gitcafe.com/billryan/algorithm-exercise/raw/deploy/pdf/algorithm-ebook_zh-cn.pdf) - 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](https://raw.githubusercontent.com/billryan/algorithm-exercise/deploy/mobi/algorithm-ebook_zh-cn.mobi), [Gitbook](https://www.gitbook.com/download/mobi/book/yuanbin/algorithm?lang=zh-cn), [GitCafe(中国大陆用户适用)](https://gitcafe.com/billryan/algorithm-exercise/raw/deploy/mobi/algorithm-ebook_zh-cn.mobi) - 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](http://algorithm.yuanbin.me/en/index.html) maintained by who?
* [简体中文](http://algorithm.yuanbin.me/zh-cn/index.html) maintained by [@billryan](https://github.com/billryan)
* [繁體中文](http://algorithm.yuanbin.me/zh-tw/index.html) maintained by [@CrossLuna](https://github.com/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](https://github.com/billryan/algorithm-exercise/blob/master/contributing_zh-cn.md).

## 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:

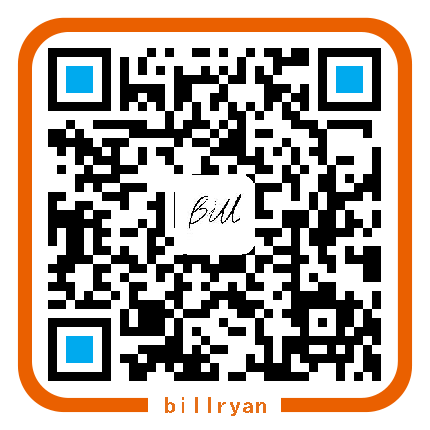
### 邮寄明信片

@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](https://leetcode.com/) - 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](http://www.lintcode.com) - 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](https://www.gitbook.com/book/siddontang/leetcode-solution/details) - 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](https://github.com/FreeTymeKiyan/LeetCode-Sol-Res) - Clean, Understandable Solutions and Resources on LeetCode Online Judge Algorithms Problems.
* [soulmachine/leetcode](https://github.com/soulmachine/leetcode) - 含C++和Java两个版本的题解。
* [Woodstock Blog](http://okckd.github.io/) - 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](http://www.itint5.com/) - 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](http://www.acmerblog.com/) - 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](http://www.nowcoder.com/) - 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](http://www.jiuzhang.com/) - 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](http://julyedu.com/) - july大神主导的在线算法辅导。
* [刷题 | 一亩三分地论坛](http://www.1point3acres.com/bbs/forum-84-1.html) - 时不时就会有惊喜放出。
* [VisuAlgo - visualising data structures and algorithms through animation](http://www.comp.nus.edu.sg/~stevenha/visualization/index.html) - 相当碉堡的数据结构和算法可视化。
* [Data Structure Visualization](http://www.cs.usfca.edu/~galles/visualization/Algorithms.html) - 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](http://blog.csdn.net/v_JULY_v) - 不得不服！
* [julycoding/The-Art-Of-Programming-By-July](https://github.com/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](http://blog.csdn.net/v_july_v/article/details/6543438)
* [专栏：算法笔记——《算法设计与分析》](http://blog.csdn.net/column/details/lf-algoritnote.html) - 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](http://zh.lucida.me/blog/on-learning-algorithms/) - 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)](http://book.douban.com/subject/1475870/)
* [The Algorithm Design Manual](http://www.amazon.com/exec/obidos/ASIN/1848000693/thealgorithmrepo), 作者还放出了自己上课的视频和slides - [Skiena's Audio Lectures](http://www3.cs.stonybrook.edu/~algorith/video-lectures/)，[The Algorithm Design Manual (豆瓣)](http://book.douban.com/subject/3072383/)
* 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

# 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。

## Reference

* [VisuAlgo - visualising data structures and algorithms through animation](http://www.comp.nus.edu.sg/~stevenha/visualization/index.html) - 相当碉堡的数据结构和算法可视化。
* [Data Structure Visualization](http://www.cs.usfca.edu/~galles/visualization/Algorithms.html) - 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。

# Basics Data Structure

# 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

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

# 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

#### 单向链表

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。

class ListNode {  
 int val;  
 ListNode next;  
 ListNode(int val) {  
 this.val = val;  
 }  
}  
  
public ListNode reverse(ListNode head) {  
 ListNode prev = null;  
 while (head != null) {  
 ListNode next = head.next;  
 head.next = prev;  
 prev = head;  
 head = next;  
 }  
 return prev;  
}

#### 双向链表

和单向链表的区别在于：双向链表的反转核心在于 next 和 prev 域的交换，还需要注意的是当前节点和上一个节点的递推。

class DListNode {  
 int val;  
 DListNode prev, next;  
 DListNode(int val) {  
 this.val = val;  
 this.prev = this.next = null;  
 }  
}  
  
public DListNode reverse(DListNode head) {  
 DListNode curr = null;  
 while (head != null) {  
 curr = head;  
 head = curr.next;  
 curr.next = curr.prev;  
 curr.prev = head;  
 }  
 return curr;  
}

### 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

# 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 2i−12^{i-1}2i−1 个结点；深度为k的二叉树至多有 2k−12^k-12k−1 个结点；对任何一棵二叉树T，如果其终端结点数为 n0n\_0n0, 度为2的结点数为 n2n\_2n2, 则 n0=n2+1n\_0=n\_2+1n0=n2+1。

一棵深度为 kkk, 且有 2k−12^k-12k−1 个节点称之为**满二叉树**；深度为 kk k，有 nnn 个节点的二叉树，当且仅当其每一个节点都与深度为 kkk 的满二叉树中序号为 111 至 nnn 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

class TreeNode:  
 def \_\_init\_\_(self, val):  
 self.val = val  
 self.left, self.right = None, None

### C++

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

### 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。

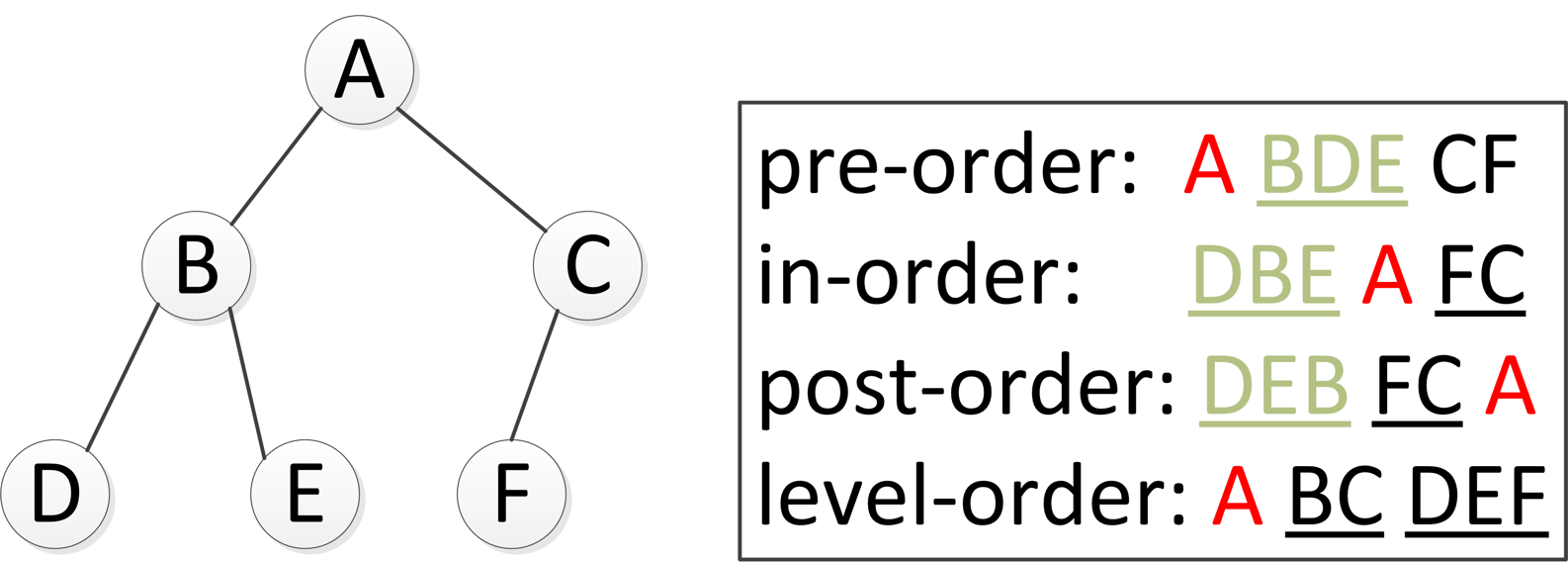
1. 前序(pre-order)：先根后左再右
2. 中序(in-order)：先左后根再右
3. 后序(post-order)：先左后右再根

* 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

# Huffman Compression - 霍夫曼压缩

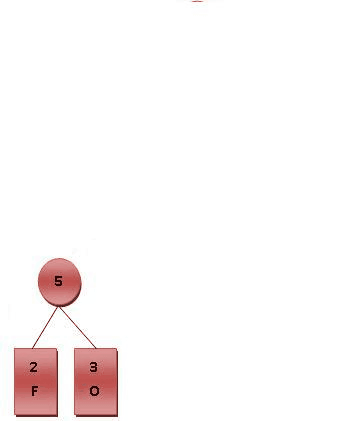
主要思想：放弃文本文件的普通保存方式：不再使用7位或8位二进制数表示每一个字符，而是**用较少的比特表示出现频率最高的字符，用较多的比特表示出现频率低的字符**。

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](http://coolshell.cn/articles/7459.html) 和 [霍夫曼编码 - 维基百科，自由的百科全书](http://zh.wikipedia.org/wiki/%E9%9C%8D%E5%A4%AB%E6%9B%BC%E7%BC%96%E7%A0%81)，清晰易懂。

## Queue

# 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(logn)O(\log n)O(logn)
* Object: remove(Object) and contains(Object) - O(n)O(n)O(n)
* retrieval: peek , element , and size - O(1)O(1)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](http://zh.wikipedia.org/zh/%E5%84%AA%E5%85%88%E4%BD%87%E5%88%97)
* [双端队列 - 维基百科，自由的百科全书](https://zh.wikipedia.org/wiki/%E5%8F%8C%E7%AB%AF%E9%98%9F%E5%88%97)

## Heap

# 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 2NlogN2N \log N2NlogN 次比较和恒定的额外空间。
3. 在索引从0开始的数组中：

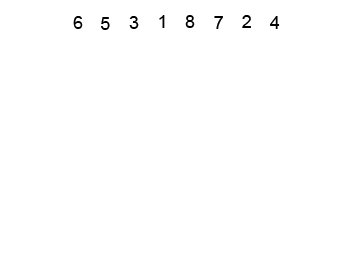
* 父节点 i 的左子节点在位置 (2\*i+1)
* 父节点 i 的右子节点在位置 (2\*i+2)
* 子节点 i 的父节点在位置 floor((i-1)/2)

### 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

# 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

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

# 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 (Map.Entry<String, Integer> entry : map.entrySet()) {  
 String key = entry.getKey();  
 int value = entry.getValue();  
 // do some thing  
}

## Graph

# Graph - 图

图的表示通常使用**邻接矩阵和邻接表**，前者易实现但是对于稀疏矩阵会浪费较多空间，后者使用链表的方式存储信息但是对于图搜索时间复杂度较高。

### 编程实现

### 邻接矩阵

设顶点个数为 V, 那么邻接矩阵可以使用 V × V 的二维数组来表示。

/\* Java Definition \*/  
int[][] g = new int[V][V]

g[i][j] 表示顶点 i 和顶点 j 的关系，对于无向图可以使用0/1表示是否有连接，对于带权图则需要使用 INF 来区分。有重边时保存边数或者权值最大/小的边即可。

### 邻接表

邻接表通过表示从顶点 i 出发到其他所有可能能到的边。

### Java

有向图。

/\* Java Definition \*/  
class DirectedGraphNode {  
 int label;  
 ArrayList<DirectedGraphNode> neighbors;  
 DirectedGraphNode(int x) {  
 label = x;  
 neighbors = new ArrayList<DirectedGraphNode>();  
 }  
}

无向图同上，只不过在建图时双向同时加。

class UndirectedGraphNode {  
 int label;  
 ArrayList<UndirectedGraphNode> neighbors;  
 UndirectedGraphNode(int x) {  
 this.label = x;  
 this.neighbors = new ArrayList<UndirectedGraphNode>();  
 }  
}

# Basics Sorting

# 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

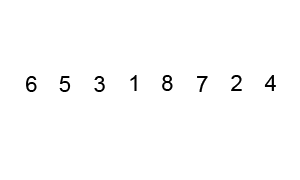
### Reference

* [常用排序算法总结（性能+代码） - SegmentFault](http://segmentfault.com/a/1190000002595152#articleHeader15)
* [Sorting algorithm - Wikipedia, the free encyclopedia](http://en.wikipedia.org/wiki/Sorting_algorithm) - 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](http://wuchong.me/blog/2014/02/09/algorithm-sort-summary/) - 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](http://blog.sina.com.cn/s/blog_eb52001d0102v1k8.html) - 总结了一些常用常问的排序算法。

## Bubble Sort

# 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。



### 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(n2)O(n^2)O(n2), 使用了 temp 作为临时交换变量，空间复杂度为 O(1)O(1)O(1).

### Reference

* [冒泡排序 - 维基百科，自由的百科全书](http://zh.wikipedia.org/wiki/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F)

## Selection Sort

# Selection Sort - 选择排序

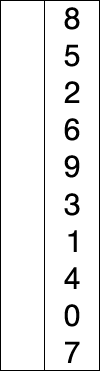
核心：不断地选择剩余元素中的最小者。

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](http://wiki.ibcsstudent.org/index.php?title=File:Selection-Sort-Animation.gif)



### 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](http://zh.wikipedia.org/wiki/%E9%80%89%E6%8B%A9%E6%8E%92%E5%BA%8F)
* [The Selection Sort — Problem Solving with Algorithms and Data Structures](http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html)

## Insertion Sort

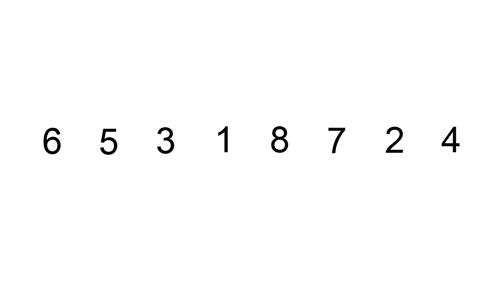
# Insertion Sort - 插入排序

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



### 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

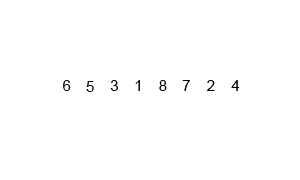
* [插入排序 - 维基百科，自由的百科全书](http://zh.wikipedia.org/wiki/%E6%8F%92%E5%85%A5%E6%8E%92%E5%BA%8F)
* [希尔排序 - 维基百科，自由的百科全书](http://zh.wikipedia.org/wiki/%E5%B8%8C%E5%B0%94%E6%8E%92%E5%BA%8F)
* [The Insertion Sort — Problem Solving with Algorithms and Data Structures](http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html)

## Merge Sort

# 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。



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

### Reference

* [Mergesort](http://algs4.cs.princeton.edu/22mergesort/) - Robert Sedgewick 的大作，非常清晰。

## Quick Sort

## Heap Sort

# Heap Sort - 堆排序

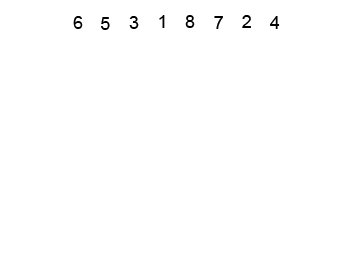
堆排序通常基于[二叉堆](http://algorithm.yuanbin.me/zh-cn/basics_data_structure/heap.html)实现，以大根堆为例，堆排序的实现过程分为两个子过程。第一步为取出大根堆的根节点(当前堆的最大值), 由于取走了一个节点，故需要对余下的元素重新建堆。重新建堆后继续取根节点，循环直至取完所有节点，此时数组已经有序。基本思想就是这样，不过实现上还是有些小技巧的。

### 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。



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)O(2N)O(2N), 堆排过程中重建的时间复杂度为 O(2NlogN)O(2N \log N)O(2NlogN). 故总的时间复杂度为 O(NlogN)O(N \log N)O(NlogN).

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](http://zh.wikipedia.org/wiki/%E5%A0%86%E6%8E%92%E5%BA%8F)
* [Priority Queues](http://algs4.cs.princeton.edu/24pq/) - 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](http://wuchong.me/blog/2014/02/09/algorithm-sort-summary/) - 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？](http://www.zhihu.com/question/20729324)
* [《大话数据结构》第9章 排序 9.7 堆排序（上） - 伍迷 - 博客园](http://www.cnblogs.com/cj723/archive/2011/04/21/2024261.html)
* [《大话数据结构》第9章 排序 9.7 堆排序（下） - 伍迷 - 博客园](http://www.cnblogs.com/cj723/archive/2011/04/22/2024269.html)

## Bucket Sort

# 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](http://www.cs.usfca.edu/~galles/visualization/BucketSort.html) - 动态演示。
* [桶排序 - 维基百科，自由的百科全书](http://zh.wikipedia.org/wiki/%E6%A1%B6%E6%8E%92%E5%BA%8F)

## Counting Sort

# 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](http://zh.wikipedia.org/wiki/%E8%AE%A1%E6%95%B0%E6%8E%92%E5%BA%8F) - 中文版的维基感觉比英文版的好理解些。
* [Counting Sort Visualization](https://www.cs.usfca.edu/~galles/visualization/CountingSort.html) - 动画真心不错~ 结合着看一遍就理解了。

## Radix Sort

# Radix Sort

# Basics Algorithm

# 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

# 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

## Binary Search

# Binary Search - 二分搜索

二分搜索是一种在有序数组中寻找目标值的经典方法，也就是说使用前提是『有序数组』。非常简单的题中『有序』特征非常明显，但更多时候可能需要我们自己去构造『有序数组』。下面我们从最基本的二分搜索开始逐步深入。

### 模板一 - lower/upper bound

定义 lower bound 为在给定升序数组中大于等于目标值的最小索引，upper bound 则为小于等于目标值的最大索引，下面上代码和测试用例。

### Java

import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 int[] nums = new int[]{1,2,2,3,4,6,6,6,13,18};  
 System.out.println(lowerBound(nums, 6)); // 5  
 System.out.println(upperBound(nums, 6)); // 7  
 System.out.println(lowerBound(nums, 7)); // 8  
 System.out.println(upperBound(nums, 7)); // 7  
 }  
  
 /\*  
 \* nums[index] >= target, min(index)  
 \*/  
 public static int lowerBound(int[] nums, int target) {  
 if (nums == null || nums.length == 0) return -1;  
 int lb = -1, ub = nums.length;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (nums[mid] < target) {  
 lb = mid;  
 } else {  
 ub = mid;  
 }  
 }  
  
 return lb + 1;  
 }  
  
 /\*  
 \* nums[index] <= target, max(index)  
 \*/  
 public static int upperBound(int[] nums, int target) {  
 if (nums == null || nums.length == 0) return -1;  
 int lb = -1, ub = nums.length;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (nums[mid] > target) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 }  
  
 return ub - 1;  
 }  
}

### 源码分析

以 lowerBound 的实现为例，以上二分搜索的模板有几个非常优雅的实现：

1. while 循环中 lb + 1 < ub , 而不是等号，因为取等号可能会引起死循环。初始化 lb < ub 时，最后循环退出时一定有 lb + 1 == ub .
2. mid = lb + (ub - lb) / 2 , 可有效防止两数相加后溢出。
3. lb 和 ub 的初始化，初始化为数组的两端以外，这种初始化方式比起 0 和 nums.length - 1 有不少优点，详述如下。

如果遇到有问插入索引的位置时，可以分三种典型情况：

1. 目标值在数组范围之内，最后返回值一定是 lb + 1
2. 目标值比数组最小值还小，此时 lb 一直为 -1 , 故最后返回 lb + 1 也没错，也可以将 -1 理解为数组前一个更小的值
3. 目标值大于等于数组最后一个值，由于循环退出条件为 lb + 1 == lb , 那么循环退出时一定有 lb = A.length - 1 , 应该返回 lb + 1

综上所述，返回 lb + 1 是非常优雅的实现。其实以上三种情况都可以统一为一种方式来理解，即索引 -1 对应于数组前方一个非常小的数，索引 ub 即对应数组后方一个非常大的数，那么要插入的数就一定在 lb 和 ub 之间了。

**有时复杂的边界条件处理可以通过『补项』这种优雅的方式巧妙处理。**

关于lb 和 ub 的初始化，由于 mid = lb + (ub - lb) / 2 , 且有 lb + 1 < ub ，故 mid 还是有可能为 ub - 1 或者 lb + 1 的，在需要访问 mid + 1 或者 mid - 1 处索引的元素时可能会越界。这时候就需要将初始化方式改为 lb = 0, ub = A.length - 1 了，最后再加一个关于 lb, ub 处索引元素的判断即可。如 [Search for a Range](http://algorithm.yuanbin.me/zh-cn/binary_search/search_for_a_range.html) 和 [Find Peak Element](http://algorithm.yuanbin.me/zh-cn/binary_search/find_peak_element.html). 尤其是 Find Peak Element 中 lb 和 ub 的初始值如果初始化为-1和数组长度会带来一些麻烦。

### 模板二 - 最优解

除了在有序数组中寻找目标值这种非常直接的二分搜索外，我们还可以利用二分搜索求最优解（最大值/最小值），通常这种题中只是隐含了『有序数组』，需要我们自己构造。

用数学语言来描述就是『求满足某条件 C(x)C(x)C(x) 的最小/大的 xxx』，以求最小值为例，对于任意满足条件的 xxx, 如果所有的 x≤x′≤UBx \leq x^\prime \leq UBx≤x′≤UB 对于 C(x′)C(x^\prime)C(x′) 都为真（其中 UB 可能为无穷大，也可能为满足条件的最大的解，如果不满足此条件就不能保证二分搜索的正确性），那么我们就能使用二分搜索进行求解，其中初始化时下界 lb 初始化为不满足条件的值 LB , 上界初始化为满足条件的上界 UB . 随后在 while 循环内部每次取中，满足条件就取 ub = mid , 否则 lb = mid , 那么最后 ub 就是要求的最小值。求最大值时类似，只不过处理的是 lb .

以 [POJ No.1064](http://poj.org/problem?id=1064) 为例。

### Problem

有 NNN 条绳子，它们的长度分别为 LiL\_iLi. 如果从它们中切割出 KKK 条长度相同的绳子的话，这 KKK 条绳子每条最长能有多长？答案保留到小数点后两位。

#### 输入

N = 4, L = {8.02, 7.43, 4.57, 5.39}, K = 11

#### 输出

2.00

### 题解

这道题看似是一个最优化问题，我们来尝试下使用模板二的思想求解，**令 C(x)C(x)C(x) 为『可以得到 KKK 条长度为 xxx 的绳子』。**根据题意，我们可以将上述条件进一步细化为：C(x)=∑i(floor(Li/x))≥KC(x) = \sum\_i(floor(L\_i / x)) \geq KC(x)=i∑(floor(Li/x))≥K

我们现在来分析下可行解的上下界。由于答案保留小数点后两位，显然绳子长度一定大于0，大于0的小数点后保留两位的最小值为 0.01 , 显然如果问题最后有解， 0.01 一定是可行解中最小的，且这个解可以分割出的绳子条数是最多的。一般在 OJ 上不同变量都是会给出范围限制，那么我们将上界初始化为 最大范围 + 0.01 , 它一定在可行解之外（也可以遍历一遍数组取数组最大值，但其实二分后复杂度相差不大）。使用二分搜索后最后返回 lb 即可。

### Java

import java.io.\*;  
import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner in = new Scanner(System.in);  
 int n = in.nextInt();  
 int k = in.nextInt();  
 double[] nums = new double[n];  
 for (int i = 0; i < n; i++) {  
 nums[i] = in.nextDouble();  
 }  
 System.out.printf("%.2f\n", Math.floor(solve(nums, k) \* 100) / 100);  
 }  
  
 public static double solve(double[] nums, int K) {  
 double lb = 0.00, ub = 10e5 + 0.01;  
 // while (lb + 0.001 < ub) {  
 for (int i = 0; i < 100; i++) {  
 double mid = lb + (ub - lb) / 2;  
 if (C(nums, mid, K)) {  
 lb = mid;  
 } else {  
 ub = mid;  
 }  
 }  
 return lb;  
 }  
  
 public static boolean C(double[] nums, double seg, int k) {  
 int count = 0;  
 for (double num : nums) {  
 count += Math.floor(num / seg);  
 }  
 return count >= k;  
 }  
}

### 源码分析

方法 C 只做一件事，给定数组 nums , 判断是否能切割出 K 条长度均为 seg 的绳子。 while 循环中使用 lb + 0.001 < ub , 不能使用 0.01 , 因为计算 mid 时有均值的计算，对于 double 型数值否则会有较大误差。

### 模板三 - 二分搜索的 while 结束条件判定

对于整型我们通常使用 lb + 1 < ub , 但对于 double 型数据来说会有些精度上的丢失，使得结束条件不是那么好确定。像上题中采用的方法是题目中使用的精度除10。但有时候这种精度可能还是不够，如果结束条件 lb + EPS < ub 中使用的 EPS 过小时 double 型数据精度有可能不够从而导致死循环的产生！这时候我们将 while 循环体替换为 for (int i = 0; i < 100; i++) , 100 次循环后可以达到 10−3010^{-30}10−30 精度范围，一般都没问题。

### Reference

* 《挑战程序设计竞赛》

## Math

# Math

本小节总结一些与数学（尤其是数论部分）有关的基础，主要总结了《挑战程序设计竞赛》第二章。主要包含以下内容：

1. Greatest Common Divisor(最大公约数)
2. Prime(素数基础理论)
3. Modulus(求模运算)
4. Fast Power(快速幂运算)

### Modulus - 求模运算

有时计算结果可能会溢出，此时往往需要对结果取余。如果有 a % m = c % m 和 b % m = d % m , 那么有以下模运算成立。

* (a + b) % m = (c + d) % m
* (a - b) % m = (c - d) % m
* (a × b) % m = (c × d) % m

需要注意的是没有除法运算，另外由于最终结果可能溢出，故需要使用更大范围的类型来保存求模之前的结果。另外若 a 是负数时往往需要改写为 a % m + m , 这样就保证结果在 [0, m - 1] 范围内了。

### Fast Power - 快速幂运算

快速幂运算的核心思想为反复平方法，将幂指数表示为2的幂次的和，等价于二进制进行移位计算（不断取幂的最低位），比如 x22=x16x4x2x^{22} = x^{16} x^4 x^2x22=x16x4x2.

### Java

import java.util.\*;  
  
public class FastPow {  
 public static long fastModPow(long x, long n, long mod) {  
 long res = 1;  
 while (n > 0) {  
 // if lowest bit is 1  
 if ((n & 1) != 0) res = res \* x % mod;  
 x = x \* x % mod;  
 n >>= 1;  
 }  
 return res;  
 }  
  
 public static void main(String[] args) {  
 if (args.length != 2 && args.length != 3) return;  
  
 long x = Long.parseLong(args[0]);  
 long n = Long.parseLong(args[1]);  
 long mod = Long.MAX\_VALUE;  
 if (args.length == 3) {  
 mod = Long.parseLong(args[2]);  
 }  
 System.out.println(fastModPow(x, n, mod));  
 }  
}

### Greatest Common Divisor

# Math

本小节总结一些与数学（尤其是数论部分）有关的基础，主要总结了《挑战程序设计竞赛》第二章。

### 最大公约数(GCD, Greatest Common Divisor)

常用的方法为辗转相除法，也称为欧几里得算法。不妨设函数 gcd(a, b) 是自然是 a , b 的最大公约数，不妨设 a > b , 则有 a=b×p+qa = b \times p + qa=b×p+q, 那么对于 gcd(b, q) 则是 b 和 q 的最大公约数，也就是说 gcd(b, q) 既能整除 b , 又能整除 a (因为 a=b×p+qa = b \times p + qa=b×p+q, p 是整数)，如此反复最后得到 gcd(a, b) = gcd(c, 0) , 第二个数为0时直接返回 c . 如果最开始 a < b , 那么 gcd(b, a % b) = gcd(b, a) = gcd(a, b % a) .

关于时间复杂度的证明：可以分 a > b/2 和 a < b/2 证明，对数级别的时间复杂度，过程略。

与最大公约数相关的还有最小公倍数(LCM, Lowest Common Multiple), 它们两者之间的关系为 lcm(a,b)×gcd(a,b)=∣ab∣ lcm(a, b) \times gcd(a, b) = |ab|lcm(a,b)×gcd(a,b)=∣ab∣.

### Java

public static long gcd(long a, long b) {  
 return (b == 0) ? a : gcd(b, a % b);  
}

### Problem

给定平面上两个坐标 P1=(x1, y1), P2=(x2,y2), 问线段 P1P2 上除 P1, P2以外还有几个整数坐标点？

#### Solution

问的是线段 P1P2, 故除 P1,P2以外的坐标需在 x1,x2,y1,y2范围之内，且不包含端点。在两端点不重合的前提下有：

y−y1x−x1=y2−y1x2−x1\frac{y-y\_1}{x-x\_1}=\frac{y\_2 - y\_1}{x\_2 - x\_1}x−x1y−y1=x2−x1y2−y1那么若得知 M=gcd(x2−x1,y2−y1)M = gcd(x\_2 - x\_1, y\_2 - y\_1)M=gcd(x2−x1,y2−y1), 则有 x−x1x - x\_1x−x1 必为 x2−x1/Mx\_2 - x\_1 / Mx2−x1/M 的整数倍大小，又因为 x1<x<x2 x\_1 < x < x\_2x1<x<x2, 故最多有 M−1M - 1M−1个整数坐标点。

### 扩展欧几里得算法

求解整系数 xxx 和 yyy 满足 d=gcd(a,b)=ax+byd = gcd(a, b) = ax + byd=gcd(a,b)=ax+by, 仿照欧几里得算法，应该要寻找 gcd(b,a%b)=bx′+(a%b)y′gcd(b, a \% b) = bx^\prime + (a \% b)y^\primegcd(b,a%b)=bx′+(a%b)y′.

### Java

public class Solution {  
 public static int gcd(int a, int b) {  
 return b == 0 ? a : gcd(b, a % b);  
 }  
  
 public static int[] gcdExt(int a, int b) {  
 if (b == 0) {  
 return new int[] {a, 1, 0};  
 } else {  
 int[] vals = gcdExt(b, a % b);  
 int d = vals[0];  
 int x = vals[2];  
 int y = vals[1];  
 y -= (a / b) \* x;  
 return new int[] {d, x, y};  
 }  
 }  
  
 public static void main(String[] args) {  
 int a = 4, b = 11;  
 int[] result = gcdExt(a, b);  
 System.out.printf("d = %d, x = %d, y = %d.\n", result[0], result[1], result[2]);  
 }  
}

### Problem

求整数 xxx 和 yyy 使得 ax+by=1ax+by=1ax+by=1.

#### Solution

不妨设 gcd(a, b) = M , 那么有 M(a′x+b′y)=1M(a^\prime x+b^\prime y)=1M(a′x+b′y)=1 ==> a′x+b′y=1/Ma^\prime x+b^\prime y=1/Ma′x+b′y=1/M 如果 M 大于1，由于等式左边为整数，故等式不成立，所以要想题中等式有解，必有 gcd(a, b) = 1 .

**扩展提：题中等式右边为1，假如为2又会怎样？**

提示：此时c=k⋅gcd(a,b),x′=k⋅x==>c % gcd(a,b)==0c = k \cdot gcd(a, b), x^\prime = k\cdot x ==> c\ \%\ gcd(a, b) == 0c=k⋅gcd(a,b),x′=k⋅x==>c % gcd(a,b)==0, c 为等式右边的正整数值。详细推导见 [How to find solutions of linear Diophantine ax + by = c?](http://math.stackexchange.com/questions/20717/how-to-find-solutions-of-linear-diophantine-ax-by-c)

### Prime

# Prime

素数：恰好有两个约数的整数，一个是1，另一个则是它自己，比如整数3和5就是素数。素数的基本算法有**素性测试、埃氏筛法和整数分解。**

### 素性测试

如果 d 是 n 的约数，则易知 n=d⋅ndn = d \cdot \frac{n}{d}n=d⋅dn, 因此 n/d 也是 n 的约数，且这两个约数中的较小者 min(d,n/d)<=n\min(d, n/d) <= \sqrt{n}min(d,n/d)<=√n. 因此我们只需要对前 n\sqrt{n}√n 个数进行处理。

### 埃氏筛法

素性测试针对的是单个整数，如果需要枚举整数 n 以内的素数就需要埃氏筛法了。核心思想是枚举从小到大的素数并将素数的整数倍依次从原整数数组中删除，余下的即为全部素数。

### 区间筛法

求区间 [a, b) 内有多少素数？

埃氏筛法得到的是 [1, n) 内的素数，求区间素数时不太容易直接求解，我们采取以退为进的思路先用埃氏筛法求得 [1, b) 内的素数，然后截取为 [a, b) 即可。

### Implementation

### Java

import java.util.\*;  
  
public class Prime {  
 // test if n is prime  
 public static boolean isPrime(int n) {  
 for (int i = 2; i \* i <= n; i++) {  
 if (n % i == 0) return false;  
 }  
 return n != 1; // 1 is not prime  
 }  
  
 // enumerate all the divisor for n  
 public static List<Integer> getDivisor(int n) {  
 List<Integer> result = new ArrayList<Integer>();  
 for (int i = 1; i \* i <= n; i++) {  
 if (n % i == 0) {  
 result.add(i);  
 // i \* i <= n ==> i <= n / i  
 if (i != n / i) result.add(n / i);  
 }  
 }  
 Collections.sort(result);  
 return result;  
 }  
  
 // 12 = 2 \* 2 \* 3, the number of prime factor, small to big  
 public static Map<Integer, Integer> getPrimeFactor(int n) {  
 Map<Integer, Integer> result = new HashMap<Integer, Integer>();  
 for (int i = 2; i \* i <= n; i++) {  
 // if i is a factor of n, repeatedly divide it out  
 while (n % i == 0) {  
 if (result.containsKey(i)) {  
 result.put(i, result.get(i) + 1);  
 } else {  
 result.put(i, 1);  
 }  
 n = n / i;  
 }  
 }  
 // if n is not 1 at last  
 if (n != 1) result.put(n, 1);  
 return result;  
 }  
  
 // sieve all the prime factor less equal than n  
 public static List<Integer> sieve(int n) {  
 List<Integer> prime = new ArrayList<Integer>();  
 // flag if i is prime  
 boolean[] isPrime = new boolean[n + 1];  
 Arrays.fill(isPrime, true);  
 isPrime[0] = false;  
 isPrime[1] = false;  
 for (int i = 2; i <= n; i++) {  
 if (isPrime[i]) {  
 prime.add(i);  
 for (int j = 2 \* i; j <= n; j += i) {  
 isPrime[j] = false;  
 }  
 }  
 }  
 return prime;  
 }  
  
 // sieve between [a, b)  
 public static List<Integer> sieveSegment(int a, int b) {  
 List<Integer> prime = new ArrayList<Integer>();  
 boolean[] isPrime = new boolean[b];  
 Arrays.fill(isPrime, true);  
 isPrime[0] = false;  
 isPrime[1] = false;  
 for (int i = 2; i < b; i++) {  
 if (isPrime(i)) {  
 for (int j = 2 \* i; j < b; j += i) isPrime[j] = false;  
 if (i >= a) prime.add(i);  
 }  
 }  
 return prime;  
 }  
  
 public static void main(String[] args) {  
 if (args.length == 1) {  
 int n = Integer.parseInt(args[0]);  
 if (isPrime(n)) {  
 System.out.println("Integer " + n + " is prime.");  
 } else {  
 System.out.println("Integer " + n + " is not prime.");  
 }  
 System.out.println();  
  
 List<Integer> divisor = getDivisor(n);  
 System.out.print("Divisor of integer " + n + ":");  
 for (int d : divisor) System.out.print(" " + d);  
 System.out.println();  
 System.out.println();  
  
 Map<Integer, Integer> primeFactor = getPrimeFactor(n);  
 System.out.println("Prime factor of integer " + n + ":");  
 for (Map.Entry<Integer, Integer> entry : primeFactor.entrySet()) {  
 System.out.println("prime: " + entry.getKey() + ", times: " + entry.getValue());  
 }  
  
 System.out.print("Sieve prime of integer " + n + ":");  
 List<Integer> sievePrime = sieve(n);  
 for (int i : sievePrime) System.out.print(" " + i);  
 System.out.println();  
 } else if (args.length == 2) {  
 int a = Integer.parseInt(args[0]);  
 int b = Integer.parseInt(args[1]);  
 List<Integer> primeSegment = sieveSegment(a, b);  
 System.out.println("Prime of integer " + a + " to " + b + ":");  
 for (int i : primeSegment) System.out.print(" " + i);  
 System.out.println();  
 }  
 }  
}

## Knapsack

# 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 WWW kg, 有 nnn 种珠宝可供选择，这 nnn 种珠宝的重量分别为 ω1,⋯,ωn\omega\_1,\cdots,\omega\_nω1,⋯,ωn, 相应的价值为 v1,⋯,vnv\_1,\cdots,v\_nv1,⋯,vn. 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 ω≤W\omega \leq Wω≤W, 要么尝试更少的珠宝数目(如珠宝 1,2,⋯,j, for j≤n1, 2, \cdots , j, ~for~ j \leq n1,2,⋯,j, for j≤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\omegaω 时抢劫犯所能获得的最优值为 K(ω)K(\omega)K(ω).** 对应此状态的状态转移方程并不是那么直观，先从 K(ω)K(\omega)K(ω) 所包含的信息出发，K(ω)>0K(\omega) > 0K(ω)>0 时，背包中必然含有某件值钱的珠宝，不妨假设最优值 K(ω)K(\omega)K(ω) 包含某珠宝 iii, 那么将珠宝 iii 从背包中移除后，背包中剩余珠宝的价值加上珠宝 iii 的价值即为 K(ω)K(\omega)K(ω). 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(ω)=F(ω−ωi)+vi (ωi∈Ω)K(\omega) = F(\omega - \omega\_i) + v\_i ~(\omega\_i \in \Omega)K(ω)=F(ω−ωi)+vi (ωi∈Ω)

其中 F(ω−ωi)F(\omega - \omega\_i)F(ω−ωi) 为拿出珠宝 iii 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 ωi<ω\omega\_i < \omegaωi<ω(总不能取出大于背包重量的珠宝吧...), Ω\OmegaΩ 即为 K(ω)K(\omega)K(ω) 中 ωi\omega\_iωi 的所有可能取值。想了想好像哪里不对劲，K(ω)K(\omega)K(ω) 的转移关系没鼓捣出来，反而新添了个 F(ω−ωi)F(\omega - \omega\_i)F(ω−ωi), 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 iii 后，其价值 viv\_ivi 就可以认为是一个定值了，故要想 K(ω)K(\omega)K(ω) 为最大值，F(ω−ωi)F(\omega - \omega\_i)F(ω−ωi) 也理应是背包容量为 ω−ωi\omega - \omega\_iω−ωi 时的包内珠宝的最大价值，如若不是，则必然存在 F(ω−ωi)<K(ω−ωi)F(\omega - \omega\_i) < K(\omega - \omega\_i)F(ω−ωi)<K(ω−ωi), 即有K(ω)=F(ω−ωi)+vi<K(ω−ωi)+vi=K′(ω)K(\omega) = F(\omega - \omega\_i) + v\_i < K(\omega - \omega\_i) + v\_i = K^{\prime}(\omega)K(ω)=F(ω−ωi)+vi<K(ω−ωi)+vi=K′(ω)与 K(ω)K(\omega)K(ω) 为在背包容量为 ω\omegaω 时的最大值的定义不符，故假设不成立，F(ω−ωi)=K(ω−ωi)F(\omega - \omega\_i) = K(\omega - \omega\_i)F(ω−ωi)=K(ω−ωi). 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(ω)=K(ω−ωi)+viK(\omega) = K(\omega - \omega\_i) + v\_iK(ω)=K(ω−ωi)+vi

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 iii 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 viv\_ivi 是从已知背包容量为 ω\omegaω 时取出来的珠宝 iii, 重量为 ωi\omega\_iωi. 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\omegaω 中的某一个。既然是这样，我们把所有小于背包容量 ω\omegaω 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 ωi\omega\_iωi 不一定是最大值 K(ω)K(\omega)K(ω)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(ω)K(\omega)K(ω)(如果不是，反证法证之), 用数学语言表示就是：K(ω)≥K(ω−ωj)+vj (ωj∉Ω)K(\omega) \geq K(\omega - \omega\_j) + v\_j ~(\omega\_j \notin \Omega)K(ω)≥K(ω−ωj)+vj (ωj∉Ω)

整理一下思路，用优雅的数学语言来表示就是：K(ω)=maxi: ωi≤ω{K(ω−ωi)+vi}K(\omega) = \max\_{i:~\omega\_i \leq \omega} {K(\omega - \omega\_i) + v\_i}K(ω)=maxi: ωi≤ω{K(ω−ωi)+vi}

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...

令 dp[i + 1][j] 表示从前 i 种物品中选出总重量不超过 j 时总价值的最大值。那么有转移方程：

dp[i + 1][j] = max{dp[i][j - k × w[i]] + k × v[i] | 0 ≤ k}

最坏情况下时间复杂度为 O(kW2)O(kW^2)O(kW2). 对上式进一步变形可得：

dp[i + 1][j] = max{dp[i][j - k × w[i]] + k × v[i] | 0 ≤ k}  
 = max{dp[i][j], max{dp[i][j - k × w[i]] + k × v[i] | 1 ≤ k}}  
 = max{dp[i][j], max{dp[i][(j - w[i]) - k × w[i]] + k × v[i] | 0 ≤ k} + v[i]}  
 = max{dp[i][j], dp[i + 1][j - w[i]] + v[j]}

**注意等式最后一行，咋看和01背包一样，实际上区别在于** dp[i + 1][] **, 01背包中为** dp[i][] **.** 此时时间复杂度简化为 O(nW)O(nW)O(nW).

### 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(ω)K(\omega)K(ω) 改为 K(i,ω)K(i,\omega)K(i,ω) 即可，**新的状态表示前 iii 件物品放入一个容量为 ω\omegaω 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,ω)K(i-1, \omega)K(i−1,ω)为 K(i,ω)K(i, \omega)K(i,ω) 的子问题，如果不放第 iii 件物品，那么问题即转化为「前 i−1i-1i−1 件物品放入容量为 ω\omegaω 的背包」，此时背包内获得的总价值为 K(i−1,ω)K(i-1, \omega)K(i−1,ω)；如果放入第 iii 件物品，那么问题即转化为「前 i−1i-1i−1 件物品放入容量为 ω−ωi\omega - \omega\_iω−ωi 的背包」，此时背包内获得的总价值为 K(i−1,ω−ωi)+viK(i-1, \omega - \omega\_i) + v\_iK(i−1,ω−ωi)+vi. 新的状态转移方程用数学语言来表述即为：K(i,ω)=max{K(i−1,ω),K(i−1,ω−ωi)+vi}K(i,\omega) = \max {K(i-1, \omega), K(i-1, \omega - \omega\_i) + v\_i}K(i,ω)=max{K(i−1,ω),K(i−1,ω−ωi)+vi}

这里的分析是以容量递推的，但是在容量特别大时，我们可能需要以价值作为转移方程。定义状态 dp[i + 1][j] 为前 i 个物品中挑选出价值总和为 j 时总重量的最小值（所以对于不满足条件的索引应该用充分大的值而不是最大值替代，防止溢出）。相应的转移方程为：前 i - 1 个物品价值为 j , 要么为 j - v[i] (选中第 i 个物品). 即 dp[i + 1][j] = min{dp[i][j], dp[i][j - v[i]] + w[i]} . 最终返回结果为 dp[n][j] ≤ W 中最大的 j.

### 扩展

以上我们只是求得了最终的最大获利，假如还需要输出选择了哪些项如何破？

以普通的01背包为例，如果某元素被选中，那么其必然满足 w[i] > j 且大于之前的 dp[i][j] , 这还只是充分条件，因为有可能被后面的元素代替。保险起见，我们需要跟踪所有可能满足条件的项，然后反向计算有可能满足条件的元素，有可能最终输出不止一项。

### Java

import java.util.\*;  
  
public class Backpack {  
 // 01 backpack with small datasets(O(nW), W is small)  
 public static int backpack(int W, int[] w, int[] v, boolean[] itemTake) {  
 int N = w.length;  
 int[][] dp = new int[N + 1][W + 1];  
 boolean[][] matrix = new boolean[N + 1][W + 1];  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j <= W; j++) {  
 if (w[i] > j) {  
 // backpack cannot hold w[i]  
 dp[i + 1][j] = dp[i][j];  
 } else {  
 dp[i + 1][j] = Math.max(dp[i][j], dp[i][j - w[i]] + v[i]);  
 // pick item i and get value j  
 matrix[i][j] = (dp[i][j - w[i]] + v[i] > dp[i][j]);  
 }  
 }  
 }  
  
 // determine which items to take  
 for (int i = N - 1, j = W; i >= 0; i--) {  
 if (matrix[i][j]) {  
 itemTake[i] = true;  
 j -= w[i];  
 } else {  
 itemTake[i] = false;  
 }  
 }  
  
 return dp[N][W];  
 }  
  
 // 01 backpack with big datasets(O(n\sigma{v}), W is very big)  
 public static int backpack2(int W, int[] w, int[] v) {  
 int N = w.length;  
 // sum of value array  
 int V = 0;  
 for (int i : v) {  
 V += i;  
 }  
 // initialize  
 int[][] dp = new int[N + 1][V + 1];  
 for (int[] i : dp) {  
 // should avoid overflow for dp[i][j - v[i]] + w[i]  
 Arrays.fill(i, Integer.MAX\_VALUE >> 1);  
 }  
 dp[0][0] = 0;  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j <= V; j++) {  
 if (v[i] > j) {  
 // value[i] > j  
 dp[i + 1][j] = dp[i][j];  
 } else {  
 // should avoid overflow for dp[i][j - v[i]] + w[i]  
 dp[i + 1][j] = Math.min(dp[i][j], dp[i][j - v[i]] + w[i]);  
 }  
 }  
 }  
  
 // search for the largest i dp[N][i] <= W  
 for (int i = V; i >= 0; i--) {  
 // if (dp[N][i] <= W) return i;  
 if (dp[N][i] <= W) return i;  
 }  
 return 0;  
 }  
  
 // repeated backpack  
 public static int backpack3(int W, int[] w, int[] v) {  
 int N = w.length;  
 int[][] dp = new int[N + 1][W + 1];  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j <= W; j++) {  
 if (w[i] > j) {  
 // backpack cannot hold w[i]  
 dp[i + 1][j] = dp[i][j];  
 } else {  
 dp[i + 1][j] = Math.max(dp[i][j], dp[i + 1][j - w[i]] + v[i]);  
 }  
 }  
 }  
  
 return dp[N][W];  
 }  
  
 public static void main(String[] args) {  
 int[] w1 = new int[]{2, 1, 3, 2};  
 int[] v1 = new int[]{3, 2, 4, 2};  
 int W1 = 5;  
 boolean[] itemTake = new boolean[w1.length + 1];  
 System.out.println("Testcase for 01 backpack.");  
 int bp1 = backpack(W1, w1, v1, itemTake); // bp1 should be 7  
 System.out.println("Maximum value: " + bp1);  
 for (int i = 0; i < itemTake.length; i++) {  
 if (itemTake[i]) {  
 System.out.println("item " + i + ", weight " + w1[i] + ", value " + v1[i]);  
 }  
 }  
  
 System.out.println("Testcase for 01 backpack with large W.");  
 int bp2 = backpack2(W1, w1, v1); // bp2 should be 7  
 System.out.println("Maximum value: " + bp2);  
  
 int[] w3 = new int[]{3, 4, 2};  
 int[] v3 = new int[]{4, 5, 3};  
 int W3 = 7;  
 System.out.println("Testcase for repeated backpack.");  
 int bp3 = backpack3(W3, w3, v3); // bp3 should be 10  
 System.out.println("Maximum value: " + bp3);  
 }  
}

### Reference

* 《挑战程序设计竞赛》第二章
* 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](http://blog.csdn.net/liufeng_king/article/details/8683136)
* [崔添翼 § 翼若垂天之云 › 《背包问题九讲》2.0 alpha1](http://cuitianyi.com/blog/%E3%80%8A%E8%83%8C%E5%8C%85%E9%97%AE%E9%A2%98%E4%B9%9D%E8%AE%B2%E3%80%8B2-0-alpha1/)
* [Knapsack.java](http://introcs.cs.princeton.edu/java/96optimization/Knapsack.java.html)

## Probability

# Probability

### Shuffle

# Shuffle and Sampling - 随机抽样和洗牌

### 洗牌算法

* [Shuffle a given array - GeeksforGeeks](http://www.geeksforgeeks.org/shuffle-a-given-array/)

Given an array, write a program to generate a random permutation of array elements. This question is also asked as “shuffle a deck of cards” or “randomize a given array”.

### 题解

这里以 Fisher–Yates shuffle 算法为例，伪代码如下：

To shuffle an array a of n elements (indices 0..n-1):  
 for i from 0 downto i do  
 j ← random integer such that 0 ≤ j ≤ i  
 exchange a[j] and a[i]

转化为代码为：

/\*  
 \* shuffle cards  
 \*/  
 public static void shuffleCard(int[] cards) {  
 if (cards == null || cards.length == 0) return;  
  
 Random rand = new Random();  
 for (int i = 0; i < cards.length; i++) {  
 int k = rand.nextInt(i + 1); // 0~i (inclusive)  
 int temp = cards[i];  
 cards[i] = cards[k];  
 cards[k] = temp;  
 }  
 }

看了算法和代码后让我们来使用归纳法简单证明下这个洗牌算法的正确性。我们要证明的问题是：**数组中每个元素在每个索引处出现的概率均相等。**

对于单个元素来讲，以上算法显然正确，因为交换后仍然只有一个元素。接下来我们不妨假设其遍历到数组索引为 i-1 时满足随机排列特性，那么当遍历到数组索引为 i 时，随机数 k 为 i 的概率为 1/i , 为 0~i-1 的概率为 (i-1)/i . 接下来与索引为 i 的值交换，可以得知 card[i] 出现在索引 i 的位置的概率为 1/i , 在其他索引位置的概率也为 1/i ; 而对于 card[i] 之前的元素，以索引 j 处的元素 card[j] 为例进行分析可知其在位置 j 的概率为 1/(i-1) \* (i-1)/i = 1/i , 具体含义为遍历到索引 i-1 时 card[j] 位于索引 j 的概率( 1/(i-1) )乘以遍历到索引 i 时随机数未选择与索引 j 的数进行交换的概率( (i-1)/i ).

需要注意的是前面的 j <= i-1 , 那么 card[j] 位于索引 i 的概率又是多少呢？要位于索引 i ，则随机数 k 须为 i , 这种概率为 1/i .

综上，以上算法可以实现完美洗牌（等概率）。

### Random sampling - 随机抽样

随机抽样也称为水池抽样，Randomly choosing a sample of k items from a list S containing n items. 大意是从大小为 n 的数组中随机选出 m 个整数，要求每个元素被选中的概率相同。

### 题解

比较简洁的有算法 Algorithm R, 伪代码如下：

/\*  
 S has items to sample, R will contain the result  
\*/  
ReservoirSample(S[1..n], R[1..k])  
 // fill the reservoir array  
 for i = 1 to k  
 R[i] := S[i]  
  
 // replace elements with gradually decreasing probability  
 for i = k+1 to n  
 j := random(1, i) // important: inclusive range  
 if j <= k  
 R[j] := S[i]

转化为代码为：

/\*  
 \* random sample  
 \*/  
 public static int[] randomSample(int[] nums, int m) {  
 if (nums == null || nums.length == 0 || m <= 0) return new int[]{};  
  
 int[] sample = new int[m];  
 for (int i = 0; i < m; i++) {  
 sample[i] = nums[i];  
 }  
  
 Random random = new Random();  
 for (int i = m; i < nums.length; i++) {  
 int k = random.nextInt(i + 1); // 0~i(inclusive)  
 if (k < m) {  
 sample[k] = nums[i];  
 }  
 }  
  
 return sample;  
 }

和洗牌算法类似，我们要证明的问题是：**数组中每个元素在最终采样的数组中出现的概率均相等且为** m/n **.** 洗牌算法中是排列，而对于随机抽样则为组合。

维基百科上的证明相对容易懂一些，这里我稍微复述下。首先将数组前 m 个元素填充进新数组 sample , 然后从 m 开始遍历直至数组最后一个索引。随机数 k 的范围为 0~i , 如果 k < m ，新数组的索引为 k 的元素则和原数组索引为 i 的元素交换；如果 k >= m , 则不进行交换。 i == m 时，以原数组中第 j 个元素为例，它被 nums[m] 替换的概率为 1/(m+1) , 也就是说保留在 sample 数组中的概率为 m/(m+1) . 对与第 m+1 个元素 nums[m] 来说，其位于 sample 数组中的概率则为 m\*1/(m+1) (可替换 m 个不同的元素).

接下来仍然使用数学归纳法证明，若 i 遍历到 r 时，其之前的元素出现的概率为 m/(r-1) , 那么其之前的元素中任一元素 nums[j] 被替换的概率为 m/r \* 1/m = 1/r , 不被替换的概率则为 (r-1)/r . 故元素 nums[j] 在 i 遍历完 r 后仍然保留的概率为 m/(r-1) \* (r-1)/r = m/r . 而对于元素 nums[r] 来说，其要被替换至 sample 数组中的概率则为 m/r (随机数小于m 的个数为 m).

综上，以上算法在遍历完长度为 n 的数组后每个数出现在最终 sample 数组中的概率都为 m/n .

### Implementation and Test case

**Talk is cheap, show me the code!**

### Java

import java.util.\*;  
import java.util.Random;  
  
public class Probability {  
 public static void main(String[] args) {  
 int[] cards = new int[10];  
 for (int i = 0; i < 10; i++) {  
 cards[i] = i;  
 }  
 // 100000 times test  
 final int times = 100000;  
 final int m = 5;  
 int[][] count = new int[cards.length][cards.length];  
 int[][] count2 = new int[cards.length][m];  
 for (int i = 0; i < times; i++) {  
 shuffleCard(cards);  
 shuffleTest(cards, count);  
 int[] sample = randomSample(cards, m);  
 shuffleTest(sample, count2);  
 }  
 System.out.println("Shuffle cards");  
 shufflePrint(count);  
 System.out.println();  
 System.out.println("Random sample");  
 shufflePrint(count2);  
 }  
  
 /\*  
 \* shuffle cards  
 \*/  
 public static void shuffleCard(int[] cards) {  
 if (cards == null || cards.length == 0) return;  
  
 Random rand = new Random();  
 for (int i = 0; i < cards.length; i++) {  
 int k = rand.nextInt(i + 1);  
 int temp = cards[i];  
 cards[i] = cards[k];  
 cards[k] = temp;  
 }  
 }  
  
 /\*  
 \* random sample  
 \*/  
 public static int[] randomSample(int[] nums, int m) {  
 if (nums == null || nums.length == 0 || m <= 0) return new int[]{};  
  
 m = Math.min(m, nums.length);  
 int[] sample = new int[m];  
 for (int i = 0; i < m; i++) {  
 sample[i] = nums[i];  
 }  
  
 Random random = new Random();  
 for (int i = m; i < nums.length; i++) {  
 int k = random.nextInt(i + 1);  
 if (k < m) {  
 sample[k] = nums[i];  
 }  
 }  
  
 return sample;  
 }  
  
 /\*  
 \* nums[i] = j, num j appear in index i ==> count[j][i]  
 \*/  
 public static void shuffleTest(int[] nums, int[][] count) {  
 if (nums == null || nums.length == 0) return;  
  
 for (int i = 0; i < nums.length; i++) {  
 count[nums[i]][i]++;  
 }  
 }  
  
 /\*  
 \* print shuffle test  
 \*/  
 public static void shufflePrint(int[][] count) {  
 if (count == null || count.length == 0) return;  
  
 // print index  
 System.out.print(" ");  
 for (int i = 0; i < count[0].length; i++) {  
 System.out.printf("%-7d", i);  
 }  
 System.out.println();  
 // print num appear in index i in total  
 for (int i = 0; i < count.length; i++) {  
 System.out.print(i + ": ");  
 for (int j = 0; j < count[i].length; j++) {  
 System.out.printf("%-7d", count[i][j]);  
 }  
 System.out.println();  
 }  
 }  
}

以十万次试验为例，左侧是元素 i , 列代表在相应索引位置出现的次数。可以看出分布还是比较随机的。

Shuffle cards  
 0 1 2 3 4 5 6 7 8 9  
0: 10033 9963 10043 9845 9932 10020 9964 10114 10043 10043  
1: 9907 9951 9989 10071 10059 9966 10054 10023 10015 9965  
2: 10042 10046 9893 10080 10050 9994 10024 9852 10098 9921  
3: 10039 10023 10039 10024 9919 10057 10188 9916 9907 9888  
4: 9944 9913 10196 10059 9838 10205 9899 9945 9850 10151  
5: 10094 9971 10054 9958 10022 9922 10047 9978 9965 9989  
6: 9995 10147 9824 10015 10023 9804 10050 10192 9939 10011  
7: 9941 10131 9902 9920 10040 10121 10010 9928 9984 10023  
8: 10010 9926 9883 10098 10083 10028 9801 9936 10200 10035  
9: 9995 9929 10177 9930 10034 9883 9963 10116 9999 9974  
  
Random sample  
 0 1 2 3 4  
0: 9966 10026 10078 9966 9891  
1: 9958 9806 10066 10022 10039  
2: 9923 9936 9964 10051 10083  
3: 10165 10088 10184 9928 9916  
4: 9998 9990 9973 9931 9832  
5: 10026 9932 9873 10085 10035  
6: 9942 9972 9990 10030 10026  
7: 9903 10153 9997 10051 10044  
8: 10082 10066 9804 9899 10147  
9: 10037 10031 10071 10037 9987

### Reference

* [Fisher–Yates shuffle](https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm) - 洗牌算法的详述，比较简洁的算法
* [Reservoir sampling](https://en.wikipedia.org/wiki/Reservoir_sampling)  - 水池抽样算法
* [如何测试洗牌程序 | 酷 壳 - CoolShell.cn](http://coolshell.cn/articles/8593.html) - 借鉴了其中的一些测试方法
* 《计算机程序设计艺术》第二卷（半数值算法） - 3.4.2 随机抽样和洗牌
* 《编程珠玑》第十二章 - 抽样问题

# Basics Misc

# Basics Miscellaneous

杂项部分，涉及「位操作」等。

## Bit Manipulation

# 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1/321/321/32.  
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](http://noalgo.info/344.html)
* [位运算应用技巧（2） » NoAlGo博客](http://noalgo.info/353.html)
* [位运算简介及实用技巧（一）：基础篇 | Matrix67: The Aha Moments](http://www.matrix67.com/blog/archives/263)
* *cc150* chapter 8.5 and chapter 9.5
* 《编程珠玑2》
* 《Elementary Algorithms》 Larry LIU Xinyu

# Part II - Coding

# 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

# 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

# strStr

### Source

* leetcode: [Implement strStr() | LeetCode OJ](https://leetcode.com/problems/implement-strstr/)
* lintcode: [lintcode - (13) strstr](http://www.lintcode.com/zh-cn/problem/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

/\*\*  
 \* 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

# Two Strings Are Anagrams

### Source

* CC150: [(158) Two Strings Are Anagrams](http://www.lintcode.com/en/problem/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++

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)O(2n)O(2n), 使用了额外的数组，空间复杂度 O(256)O(256)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)O(n)O(n) 和 O(n2)O(n^2)O(n2)之间，判断 s == t 时间复杂度最坏为 O(n)O(n)O(n).

### Reference

* *CC150 Chapter 9.1* 中文版 p109

## Compare Strings

# Compare Strings

### Source

* lintcode: [(55) Compare Strings](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/string/two_strings_are_anagrams.html) 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)O(2n)O(2n), 空间复杂度为 O(26)O(26)O(26).

## Anagrams

## Longest Common Substring

# Longest Common Substring

### Source

* lintcode: [(79) Longest Common Substring](http://www.lintcode.com/en/problem/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⋅lcs)O(mn \cdot lcs)O(mn⋅lcs).

### Reference

* [Longest Common Substring | 九章算法](http://www.jiuzhang.com/solutions/longest-common-substring/)

## Rotate String

# Rotate String

### Source

* lintcode: [(8) Rotate String](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 原地交换，空间复杂度为 O(1)O(1)O(1). 总共翻转3次，总的时间复杂度为 O(n)O(n)O(n), 空间复杂度为 O(1)O(1)O(1).

### Reference

* [Reverse a string in Python - Stack Overflow](http://stackoverflow.com/questions/931092/reverse-a-string-in-python)

## Reverse Words in a String

# Reverse Words in a String

### Source

* lintcode: [(53) Reverse Words in a String](http://www.lintcode.com/en/problem/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++

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

# Valid Palindrome

* tags: [palindrome]

### Source

* leetcode: [Valid Palindrome | LeetCode OJ](https://leetcode.com/problems/valid-palindrome/)
* lintcode: [(415) Valid Palindrome](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/linked_list/check_if_a_singly_linked_list_is_palindrome.html).

### 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  
 \*/  
 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)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Longest Palindromic Substring

## Space Replacement

# Space Replacement

### Source

* lintcode: [(212) Space Replacement](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 使用了 r 作为标记，空间复杂度 O(1)O(1)O(1).

## Wildcard Matching

## Length of Last Word

# Length of Last Word

### Source

* leetcode: [Length of Last Word | LeetCode OJ](https://leetcode.com/problems/length-of-last-word/)
* lintcode: [(422) Length of Last Word](http://www.lintcode.com/en/problem/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

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)O(n)O(n).

## Count and Say

# Count and Say

### Source

* leetcode: [Count and Say | LeetCode OJ](https://leetcode.com/problems/count-and-say/)
* lintcode: [(420) Count and Say](http://www.lintcode.com/en/problem/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

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](http://huntfor.iteye.com/blog/2059877)

# Integer Array

# Integer Array - 整型数组

本章主要总结与整型数组相关的题。

## Remove Element

# Remove Element

### Source

* leetcode: [Remove Element | LeetCode OJ](https://leetcode.com/problems/remove-element/)
* lintcode: [(172) Remove Element](http://www.lintcode.com/en/problem/remove-element/)

Given an array and a value, remove all occurrences of that value in place and return the new length.  
  
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++

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)O(n)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(n2)O(n^2)O(n2)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)O(n)O(n)。

### Reference

* [Remove Element | 九章算法](http://www.jiuzhang.com/solutions/remove-element/)

## Zero Sum Subarray

## Subarray Sum K

# Subarray Sum K

### Source

* GeeksforGeeks: [Find subarray with given sum - GeeksforGeeks](http://www.geeksforgeeks.org/find-subarray-with-given-sum/)

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](http://algorithm.yuanbin.me/zh-cn/integer_array/zero_sum_subarray.html) 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(i1)−f(i2)=0f(i\_1) - f(i\_2) = 0f(i1)−f(i2)=0，这道题则变为 f(i1)−f(i2)=kf(i\_1) - f(i\_2) = kf(i1)−f(i2)=k

### 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> 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](http://algorithm.yuanbin.me/zh-cn/integer_array/zero_sum_subarray.html)

### 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)f(i)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 i2>i1i\_2 > i\_1i2>i1, 题中的解等价于寻找 f(i2)−f(i1)=kf(i\_2) - f(i\_1) = kf(i2)−f(i1)=k, 则必有 f(i2)≥kf(i\_2) \geq kf(i2)≥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)f(i).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **f(i)f(i)f(i)** | **1** | **5** | **25** | **28** | **38** |
| iii | 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)O(2n)O(2n), 空间复杂度为 O(1)O(1)O(1).

### Reference

* [Find subarray with given sum - GeeksforGeeks](http://www.geeksforgeeks.org/find-subarray-with-given-sum/)

## Subarray Sum Closest

# Subarray Sum Closest

### Source

* lintcode: [(139) Subarray Sum Closest](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/integer_array/zero_sum_subarray.html) 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(nlogn)O(n \log n)O(nlogn) 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++

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

总的时间复杂度为 O(nlogn)O(n \log n)O(nlogn), 空间复杂度为 O(n)O(n)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](http://stackoverflow.com/questions/16388930/how-to-find-the-subarray-that-has-sum-closest-to-zero-or-a-certain-value-t-in-o)

## Recover Rotated Sorted Array

# Recover Rotated Sorted Array

### Source

* lintcode: [(39) Recover Rotated Sorted Array](http://lintcode.com/en/problem/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 [1,2,3,4], [2,3,4,1], [3,4,1,2], [4,1,2,3]

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

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 rotate  
 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 end) {  
 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

# Product of Array Exclude Itself

### Source

* lintcode: [(50) Product of Array Exclude Itself](http://www.lintcode.com/en/problem/product-of-array-exclude-itself/)
* GeeksforGeeks: [A Product Array Puzzle - GeeksforGeeks](http://www.geeksforgeeks.org/a-product-array-puzzle/)

Given an integers array A.  
  
Define B[i] = A[0] \* ... \* A[i-1] \* A[i+1] \* ... \* A[n-1], calculate B WITHOUT divide operation.  
  
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]⋅right[i]result[i] = left[i] \cdot right[i]result[i]=left[i]⋅right[i], 其中 left[i]=∏j=0i−1A[j]left[i] = \prod *{j = 0} ^{i - 1} A[j]left[i]=∏j=0i−1A[j], right[i]=∏j=i+1n−1A[j]right[i] = \prod* {j = i + 1} ^{n - 1} A[j]right[i]=∏j=i+1n−1A[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++

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)O(n)O(n). 使用了左右两半部分辅助空间，空间复杂度 O(2n)O(2n)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)O(1)O(1).

## Partition Array

# Partition Array

### Source

* [(31) Partition Array](http://www.lintcode.com/en/problem/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++

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)O(n)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](http://algorithm.yuanbin.me/zh-cn/basics_sorting/quick_sort.html) 的做这道题自然是分分钟的事，使用左右两根指针 left,rightleft, rightleft,right 分别代表小于、大于等于 k 的索引，左右同时开工，直至 left>rightleft > rightleft>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<=rightleft <= rightleft<=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)O(n)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](http://www.jiuzhang.com/solutions/partition-array/)

## First Missing Positive

# First Missing Positive

### Source

* leetcode: [First Missing Positive | LeetCode OJ](https://leetcode.com/problems/first-missing-positive/)
* lintcode: [(189) First Missing Positive](http://www.lintcode.com/en/problem/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(nlogn)O(n \log n)O(nlogn), 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](http://zh.wikipedia.org/wiki/%E5%9F%BA%E6%95%B0%E6%8E%92%E5%BA%8F)，[计数排序](http://algorithm.yuanbin.me/zh-cn/basics_sorting/counting_sort.html) 和 [桶排序](http://algorithm.yuanbin.me/zh-cn/basics_sorting/bucket_sort.html)。

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)O(n)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)O(n)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]=xA[i] = xA[i]=x, 则将 x 放到它该去的地方 - A[x−1]=xA[x - 1] = xA[x−1]=x, 同时将原来 A[x−1]A[x - 1]A[x−1] 地方的值交换给 A[i]A[i]A[i].

排好序后遍历桶，如果不满足 f[i]=i+1f[i] = i + 1f[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)O(n)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)O(n)O(n), 用到了 temp 作为中间交换变量，空间复杂度为 O(1)O(1)O(1).

### Reference

* [Find First Missing Positive | N00tc0d3r](http://n00tc0d3r.blogspot.com/2013/03/find-first-missing-positive.html)
* [LeetCode: First Missing Positive 解题报告 - Yu's Garden - 博客园](http://www.cnblogs.com/yuzhangcmu/p/4200096.html)
* [First Missing Positive | 九章算法](http://www.jiuzhang.com/solutions/first-missing-positive/)

## 2 Sum

# 2 Sum

### Source

* leetcode: [Two Sum | LeetCode OJ](https://leetcode.com/problems/two-sum/)
* lintcode: [(56) 2 Sum](http://www.lintcode.com/en/problem/2-sum/)

### Problem

Given an array of integers, find two numbers such that they add up to aspecific target number.

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

#### Example

numbers= [2, 7, 11, 15] , target= 9

return [1, 2]

#### Note

You may assume that each input would have exactly one solution

#### Challenge

Either of the following solutions are acceptable:

* O(n) Space, O(nlogn) Time
* O(n) Space, O(n) Time

### 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(n2)O(n^2)O(n2), 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—— xi+xj=targetx\_i + x\_j = targetxi+xj=target, 可进一步转化为 xi=target−xjx\_i = target - x\_jxi=target−xj, 其中 iii 和 jjj 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 xjx\_jxj(**注意这里并不能先初始化哈希表，否则无法排除两个相同的元素相加为 target 的情况**), 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！

### 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)

### 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 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;  
 }  
};

### Java

public class Solution {  
 /\*  
 \* @param numbers : An array of Integer  
 \* @param target : target = numbers[index1] + numbers[index2]  
 \* @return : [index1 + 1, index2 + 1] (index1 < index2)  
 \*/  
 public int[] twoSum(int[] numbers, int target) {  
 if (numbers == null || numbers.length == 0) return new int[]{0, 0};  
  
 Map<Integer, Integer> hashmap = new HashMap<Integer, Integer>();  
 int index1 = 0, index2 = 0;  
 for (int i = 0; i < numbers.length; i++) {  
 if (hashmap.containsKey(target - numbers[i])) {  
 index1 = hashmap.get(target - numbers[i]);  
 index2 = i;  
 return new int[]{1 + index1, 1 + index2};  
 } else {  
 hashmap.put(numbers[i], i);  
 }  
 }  
  
 return new int[]{0, 0};  
 }  
}

### 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。Python 中的 dict 就是天然的哈希表。
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)O(n)O(n), 遍历一次数组，时间复杂度为 O(n)O(n)O(n).

### 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)O(n)O(n), 空间复杂度也为 O(n)O(n)O(n). 标准库中的排序方法时间复杂度近似为 O(nlogn)O(n \log n)O(nlogn), 两根指针遍历数组时间复杂度为 O(n)O(n)O(n).

## 3 Sum

# 3 Sum

### Source

* leetcode: [3Sum | LeetCode OJ](https://leetcode.com/problems/3sum/)
* lintcode: [(57) 3 Sum](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/integer_array/2_sum.html), 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

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(nlogn)O(n \log n)O(nlogn), 两重 for 循环，时间复杂度近似为 O(n2)O(n^2)O(n2)，使用哈希表(字典)实现，空间复杂度为 O(n)O(n)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S[k]只减小）  
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(n2)O(n^2)O(n2)。最终运行时间为52ms

### Reference

* [3Sum | 九章算法](http://www.jiuzhang.com/solutions/3sum/)
* [A simply Python version based on 2sum - O(n^2) - Leetcode Discuss](https://leetcode.com/discuss/32455/a-simply-python-version-based-on-2sum-o-n-2)

## 3 Sum Closest

# 3 Sum Closest

### Source

* leetcode: [3Sum Closest | LeetCode OJ](https://leetcode.com/problems/3sum-closest/)
* lintcode: [(59) 3 Sum Closest](http://www.lintcode.com/en/problem/3-sum-closest/)

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target.   
Return the sum of the three integers. You may assume that each input would have exactly one solution.  
  
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

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(nlogn)O(n \log n)O(nlogn), 两重 for 循环，由于有两处优化，故最坏的时间复杂度才是 O(n2)O(n^2)O(n2), 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)O(1)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(n2)O(n^2)O(n2)运行时间 16ms

### Reference

* [3Sum Closest | 九章算法](http://www.jiuzhang.com/solutions/3sum-closest/)

## Remove Duplicates from Sorted Array

# Remove Duplicates from Sorted Array

### Source

* leetcode: [Remove Duplicates from Sorted Array | LeetCode OJ](https://leetcode.com/problems/remove-duplicates-from-sorted-array/)
* lintcode: [(100) Remove Duplicates from Sorted Array](http://www.lintcode.com/en/problem/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++

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)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Remove Duplicates from Sorted Array II

# Remove Duplicates from Sorted Array II

### Source

* leetcode: [Remove Duplicates from Sorted Array II | LeetCode OJ](https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/)
* lintcode: [(101) Remove Duplicates from Sorted Array II](http://www.lintcode.com/en/problem/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++

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)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Merge Sorted Array

# Merge Sorted Array

### Source

* leetcode: [Merge Sorted Array | LeetCode OJ](https://leetcode.com/problems/merge-sorted-array/)
* lintcode: [(6) Merge Sorted Array](http://www.lintcode.com/en/problem/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(n2)O(n^2)O(n2) 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

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)O(n)O(n). 空间复杂度 O(1)O(1)O(1).

## Merge Sorted Array II

# Merge Sorted Array II

### Source

* lintcode: [(64) Merge Sorted Array II](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 空间复杂度 O(1)O(1)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

# Median

### Source

* lintcode: [(80) Median](http://www.lintcode.com/en/problem/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(nlogn)O(n \log n)O(nlogn), 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](http://algorithm.yuanbin.me/zh-cn/basics_sorting/quick_sort.html), 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

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)O(n (1 + 1/2 + 1/4 + ...)) = O(2n)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)O(1)O(1), 满足题目要求。

## Partition Array by Odd and Even

# Partition Array by Odd and Even

### Source

* lintcode: [(373) Partition Array by Odd and Even](http://www.lintcode.com/en/problem/partition-array-by-odd-and-even/)
* [Segregate Even and Odd numbers - GeeksforGeeks](http://www.geeksforgeeks.org/segregate-even-and-odd-numbers/)

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

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)O(n)O(n), 使用了两根指针，空间复杂度 O(1)O(1)O(1).

## Kth Largest Element

# Kth Largest Element

### Source

* leetcode: [Kth Largest Element in an Array | LeetCode OJ](https://leetcode.com/problems/kth-largest-element-in-an-array/)
* lintcode: [(5) Kth Largest Element](http://www.lintcode.com/en/problem/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)O(n)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](http://algorithm.yuanbin.me/zh-cn/basics_sorting/quick_sort.html) 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

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(n2) n + n - 1 + ... + 1 = O(n^2)n+n−1+...+1=O(n2), 平均情况下 n+n/2+n/4+...+1=O(2n)=O(n)n + n/2 + n/4 + ... + 1 = O(2n)=O(n)n+n/2+n/4+...+1=O(2n)=O(n). 故平均情况时间复杂度为 O(n)O(n)O(n). 交换数组的值时使用了额外空间，空间复杂度 O(1)O(1)O(1).

# Binary Search

# 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

# Binary Search - 二分查找

### Source

* lintcode: [lintcode - (14) Binary Search](http://www.lintcode.com/en/problem/binary-search/)

### Problem

For a given sorted array (ascending order) and a target number, find thefirst index of this number in O(log n) time complexity.

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 2322^{32}232, can your code work properly?

### 题解

对于已排序升序(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，所以需要使用类似 lower\_bound 中提到的方法。

### Java

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

### 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，这里 start 初始化为 -1 主要是考虑到 end 为 1 。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。**由于这里初始化时 start < end ，所以一定是 start + 1 == end 时退出循环。
5. 迭代终止时有两种情况，一种是在原数组中找到了，这种情况下一定是 end , 因为 start 的更新只在 nums[mid] < target .
6. 最后判断 end 和 target 的关系，先排除 end 为数组长度这种会引起越界的情况，然后再判断和目标值是否相等。

### 复杂度分析

时间复杂度 O(logn)O(\log n)O(logn), 空间复杂度 (1)(1)(1).对于题中的 follow up, Java 中数组不允许使用 long 型，如果使用 long 型，那么数组大小可大 17GB 之巨！！几乎没法用。

### Reference

* 《挑战程序设计竞赛》3.1节

## Search Insert Position

# Search Insert Position

### Source

* lintcode: [(60) Search Insert Position](http://www.lintcode.com/en/problem/search-insert-position/)

### Problem

Given a sorted array and a target value, return the index if the target isfound. If not, return the index where it would be if it were inserted inorder.

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

#### Challenge

O(log(n)) time

### 题解

仍然是 [Binary Search](http://algorithm.yuanbin.me/zh-cn/basics_algorithm/binary_search.html) 中 lower\_bound 的变形，两大关键点： start 和 end 的初始化；最终插入位置和 start 以及 end 之间的关系，由于 start 对应的索引一定是小于目标值的，那么 start + 1 就是要求的值了，再检查下两端的边界，DONE

### 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 || A.length == 0) {  
 return -1;  
 }  
  
 int start = -1, end = A.length;  
 while (start + 1 < end) {  
 int mid = start + (end - start) / 2;  
 if (A[mid] == target) {  
 return mid; // no duplicates  
 } else if (A[mid] < target) {  
 start = mid;  
 } else {  
 end = mid;  
 }  
 }  
  
 return start + 1;  
 }  
}

### 源码分析

分析三种典型情况：

1. 目标值在数组范围之内，最后返回值一定是 start + 1
2. 目标值比数组最小值还小，此时 start 一直为 -1 , 故最后返回 start + 1 也没错，也可以将 -1 理解为数组前一个更小的值
3. 目标值大于等于数组最后一个值，由于循环退出条件为 start + 1 == end , 那么循环退出时一定有 start = A.length - 1 , 应该返回 start + 1

综上所述，返回 start + 1 是非常优雅的实现。其实以上三种情况都可以统一为一种方式来理解，即索引 -1 对应于在数组前方插入一个非常小的数，索引 end 即对应数组后方插入一个非常大的数，那么要插入的数就一定在 start 和 end 之间了。

有时复杂的边界条件处理可以通过『补项』这种优雅的方式巧妙处理。

### 复杂度分析

时间复杂度 O(logn)O(\log n)O(logn), 空间复杂度 O(1)O(1)O(1).

## Search for a Range

# Search for a Range

### Source

* leetcode: [Search for a Range | LeetCode OJ](https://leetcode.com/problems/search-for-a-range/)
* lintcode: [(61) Search for a Range](http://www.lintcode.com/en/problem/search-for-a-range/)

### Problem

Given a sorted array of *n* integers, find the starting and ending position ofa given target value.

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] .

#### Challenge

O(log *n*) time.

### 题解

lower/upper bound 的结合，做两次搜索即可。

### Java

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 int[] searchRange(int[] A, int target) {  
 int[] result = new int[]{-1, -1};  
 if (A == null || A.length == 0) return result;  
  
 int lb = -1, ub = A.length;  
 // lower bound  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (A[mid] < target) {  
 lb = mid;  
 } else {  
 ub = mid;  
 }  
 }  
 // whether A[lb + 1] == target, check lb + 1 first  
 if ((lb + 1 < A.length) && (A[lb + 1] == target)) {  
 result[0] = lb + 1;  
 } else {  
 result[0] = -1;  
 result[1] = -1;  
 // target is not in the array  
 return result;  
 }  
  
 // upper bound, since ub >= lb, we do not reset lb  
 ub = A.length;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (A[mid] > target) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 }  
 // target must exist in the array  
 result[1] = ub - 1;  
  
 return result;  
 }  
}

### 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. 分 lower/upper bound 两次搜索，注意如果在 lower bound 阶段未找到目标值时，upper bound 也一定找不到。
3. 取 A[lb + 1] 时一定要注意判断索引是否越界！

### 复杂度分析

两次二分搜索，时间复杂度仍为 O(logn)O(\log n)O(logn).

## First Bad Version

# First Bad Version

### Source

* lintcode: [(74) First Bad Version](http://www.lintcode.com/en/problem/first-bad-version/)

### Problem

The code base version is an integer start from 1 to n. One day, someonecommitted a bad version in the code case, so it caused this version and thefollowing versions are all failed in the unit tests. Find the first badversion.

You can call isBadVersion to help you determine which version is the firstbad one. The details interface can be found in the code's annotation part.

#### Example

Given n = 5 :

isBadVersion(3) -> false  
isBadVersion(5) -> true  
isBadVersion(4) -> true

Here we are 100% sure that the 4th version is the first bad version.

#### Note

Please read the annotation in code area to get the correct way to callisBadVersion in different language. For example, Java is VersionControl.isBadVersion(v)

#### Challenge

You should call *isBadVersion* as few as possible.

### 题解

基础算法中 [Binary Search](http://algorithm.yuanbin.me/zh-cn/basics_algorithm/binary_search.html) 的 lower bound. 找出满足条件的下界即可。

### 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):  
 lb, ub = 0, n + 1  
 while lb + 1 < ub:  
 mid = lb + (ub - lb) / 2  
 if VersionControl.isBadVersion(mid):  
 ub = mid  
 else:  
 lb = mid  
  
 return lb + 1

### C++

/\*\*  
 \* class VersionControl {  
 \* public:  
 \* static bool isBadVersion(int k);  
 \* }  
 \* you can use VersionControl::isBadVersion(k) to judge whether  
 \* 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) {  
 int lb = 0, ub = n + 1;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (VersionControl::isBadVersion(mid)) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 }  
  
 return lb + 1;  
 }  
};

### Java

/\*\*  
 \* public class VersionControl {  
 \* public static boolean isBadVersion(int k);  
 \* }  
 \* you can use VersionControl.isBadVersion(k) to judge whether  
 \* 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) {  
 int lb = 0, ub = n + 1;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (VersionControl.isBadVersion(mid)) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 }  
  
 return lb + 1;  
 }  
}

### 源码分析

lower bound 的实现，这里稍微注意下lb 初始化为 0，因为 n 从1开始。ub 和 lb 分别都在什么条件下更新就好了。另外这里并未考虑 n <= 0 的情况。

### 复杂度分析

二分搜索，O(logn)O(\log n)O(logn).

## Search a 2D Matrix

# Search a 2D Matrix

### Source

* leetcode: [Search a 2D Matrix | LeetCode OJ](https://leetcode.com/problems/search-a-2d-matrix/)
* lintcode: [(28) Search a 2D Matrix](http://www.lintcode.com/en/problem/search-a-2d-matrix/)

### Problem

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))O(log(mn))=O(log(m)+log(n))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。

显然我们应该选择一次二分搜索，直接上 lower bound 二分模板。

### Java

public class Solution {  
 /\*\*  
 \* @param matrix, a list of lists of integers  
 \* @param target, an integer  
 \* @return a boolean, indicate whether matrix contains target  
 \*/  
 public boolean searchMatrix(int[][] matrix, int target) {  
 if (matrix == null || matrix.length == 0 || matrix[0] == null) {  
 return false;  
 }  
  
 int ROW = matrix.length, COL = matrix[0].length;  
 int lb = -1, ub = ROW \* COL;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (matrix[mid / COL][mid % COL] < target) {  
 lb = mid;  
 } else {  
 if (matrix[mid / COL][mid % COL] == target) {  
 return true;  
 }  
 ub = mid;  
 }  
 }  
  
 return false;  
 }  
}

### 源码分析

仍然可以使用经典的二分搜索模板(lower bound)，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null，还要考虑到matrix[0]的长度也为0。
2. 由于 lb 的变化处一定小于 target, 故在 else 中判断。

### 复杂度分析

二分搜索，O(logn)O(\log n)O(logn).

## Search a 2D Matrix II

# Search a 2D Matrix II

### Source

* leetcode: [Search a 2D Matrix II | LeetCode OJ](https://leetcode.com/problems/search-a-2d-matrix-ii/)
* lintcode: [(38) Search a 2D Matrix II](http://lintcode.com/en/problem/search-a-2d-matrix-ii/)

### Problem

Write an efficient algorithm that searches for a value in an m x n matrix, return the occurrence of it.

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 occurrence = 0;  
 if (matrix == null || matrix[0] == null) {  
 return occurrence;  
 }  
  
 int row = 0, col = matrix[0].length - 1;  
 while (row >= 0 && row < matrix.length && col >= 0 && col < matrix[0].length) {  
 if (matrix[row][col] == target) {  
 occurrence++;  
 col--;  
 } else if (matrix[row][col] > target) {  
 col--;  
 } else {  
 row++;  
 }  
 }  
  
 return occurrence;  
 }  
}

### 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。

**严格来讲每次取二维矩阵元素前都应该进行 null 检测。**

### 复杂度分析

由于每行每列遍历一次，故时间复杂度为 O(m+n)O(m + n)O(m+n).

### Reference

[Searching a 2D Sorted Matrix Part II | LeetCode](http://articles.leetcode.com/2010/10/searching-2d-sorted-matrix-part-ii.html)

## Find Peak Element

## Search in Rotated Sorted Array

# Search in Rotated Sorted Array

### Source

* leetcode: [Search in Rotated Sorted Array | LeetCode OJ](https://leetcode.com/problems/search-in-rotated-sorted-array/)
* lintcode: [(62) Search in Rotated Sorted Array](http://www.lintcode.com/en/problem/search-in-rotated-sorted-array/)

### Problem

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 itsindex, otherwise return -1.

You may assume no duplicate exists in the array.

#### Example

For [4, 5, 1, 2, 3] and target=1 , return 2 .

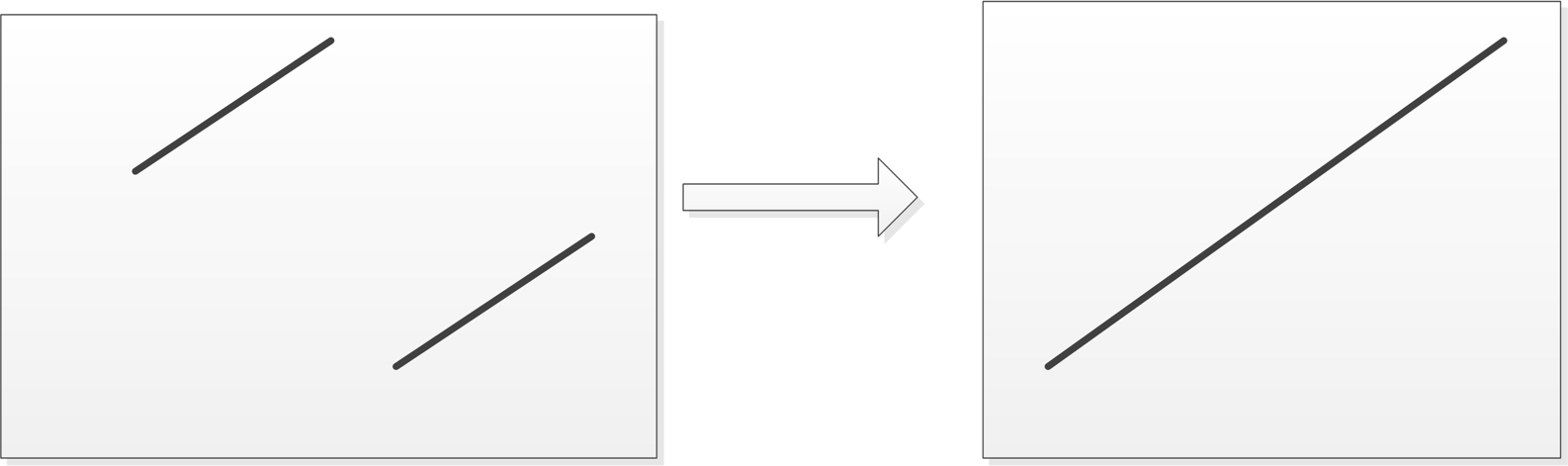
For [4, 5, 1, 2, 3] and target=0 , return -1 .

#### Challenge

O(logN) time

### 题解 - 找到有序数组

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。



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。刚开始做这道题时可能会去比较 target 和 A[mid] , 但分析起来异常复杂。**该题较为巧妙的地方在于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;  
 }  
};

### 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) {  
 if (A == null || A.length == 0) return -1;  
  
 int lb = 0, ub = A.length - 1;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (A[mid] == target) return mid;  
  
 if (A[mid] > A[lb]) {  
 // case1: numbers between lb and mid are sorted  
 if (A[lb] <= target && target <= A[mid]) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 } else {  
 // case2: numbers between mid and ub are sorted  
 if (A[mid] <= target && target <= A[ub]) {  
 lb = mid;  
 } else {  
 ub = mid;  
 }  
 }  
 }  
  
 if (A[lb] == target) {  
 return lb;  
 } else if (A[ub] == target) {  
 return ub;  
 }  
 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.

### 复杂度分析

分两段二分，时间复杂度仍近似为 O(logn)O(\log n)O(logn).

## Search in Rotated Sorted Array II

# Search in Rotated Sorted Array II

### Source

* leetcode: [Search in Rotated Sorted Array II | LeetCode OJ](https://leetcode.com/problems/search-in-rotated-sorted-array-ii/)
* lintcode: [(63) Search in Rotated Sorted Array II](http://www.lintcode.com/en/problem/search-in-rotated-sorted-array-ii/)

### Problem

Follow up for "Search in Rotated Sorted Array":What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

### 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;  
 }  
};

### Java

public class Solution {  
 /\*\*  
 \* param A : an integer ratated sorted array and duplicates are allowed  
 \* param target : an integer to be search  
 \* return : a boolean  
 \*/  
 public boolean search(int[] A, int target) {  
 if (A == null || A.length == 0) return false;  
  
 int lb = 0, ub = A.length - 1;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (A[mid] == target) return true;  
  
 if (A[mid] > A[lb]) {  
 // case1: numbers between lb and mid are sorted  
 if (A[lb] <= target && target <= A[mid]) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 } else if (A[mid] < A[lb]) {  
 // case2: numbers between mid and ub are sorted  
 if (A[mid] <= target && target <= A[ub]) {  
 lb = mid;  
 } else {  
 ub = mid;  
 }  
 } else {  
 // case3: A[mid] == target  
 lb++;  
 }  
 }  
  
 if (target == A[lb] || target == A[ub]) {  
 return true;  
 }  
 return false;  
 }  
}

### 源码分析

在 A[start] == A[mid] 时递增start序号即可。

### 复杂度分析

最差情况下 O(n)O(n)O(n), 平均情况下 O(logn)O(\log n)O(logn).

## Find Minimum in Rotated Sorted Array

# Find Minimum in Rotated Sorted Array

### Source

* leetcode: [Find Minimum in Rotated Sorted Array | LeetCode OJ](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/)
* lintcode: [(159) Find Minimum in Rotated Sorted Array](http://www.lintcode.com/en/problem/find-minimum-in-rotated-sorted-array/)

### Problem

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.

#### Example

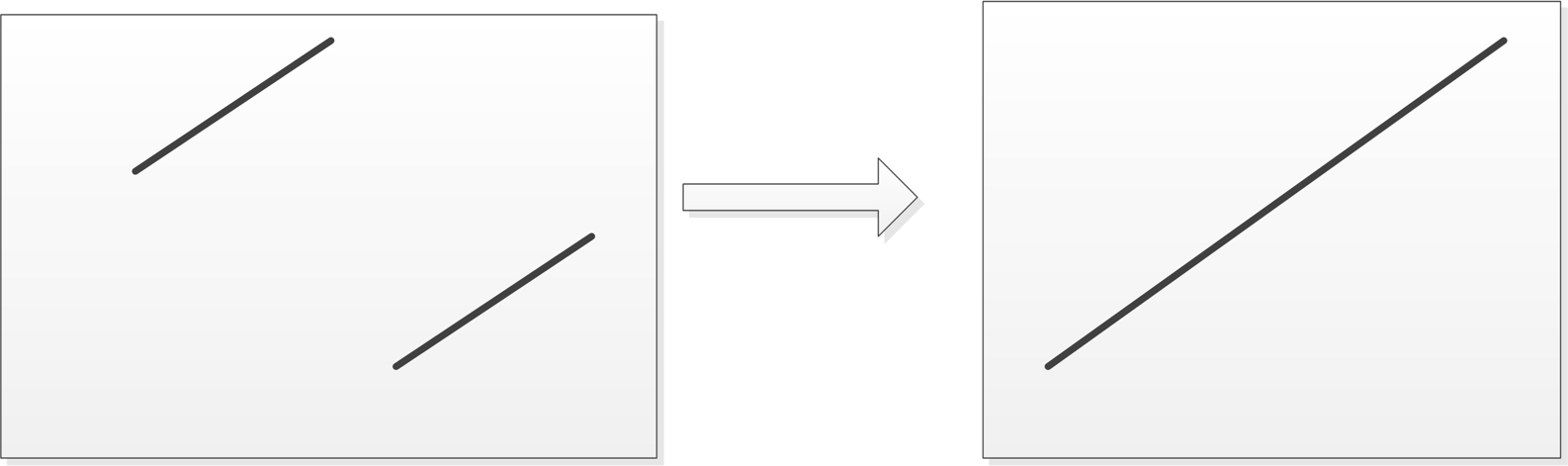
Given [4, 5, 6, 7, 0, 1, 2] return 0

#### Note

You may assume no duplicate exists in the array.

### 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。**不过考虑在内的话也算是一种优化。

### 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];  
 }  
 }  
};

### Java

public class Solution {  
 /\*\*  
 \* @param num: a rotated sorted array  
 \* @return: the minimum number in the array  
 \*/  
 public int findMin(int[] num) {  
 if (num == null || num.length == 0) return Integer.MIN\_VALUE;  
  
 int lb = 0, ub = num.length - 1;  
 // case1: num[0] < num[num.length - 1]  
 // if (num[lb] < num[ub]) return num[lb];  
  
 // case2: num[0] > num[num.length - 1] or num[0] < num[num.length - 1]  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (num[mid] < num[ub]) {  
 ub = mid;  
 } else {  
 lb = mid;  
 }  
 }  
  
 return Math.min(num[lb], num[ub]);  
 }  
}

### 源码分析

仅需注意使用 num[end] (使用 num[lb]不是那么直观)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。

### 复杂度分析

由于无重复元素，平均情况下复杂度为 O(logn)O(\log n)O(logn).

## Find Minimum in Rotated Sorted Array II

# Find Minimum in Rotated Sorted Array II

### Source

* leetcode: [Find Minimum in Rotated Sorted Array II | LeetCode OJ](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/)
* lintcode: [(160) Find Minimum in Rotated Sorted Array II](http://www.lintcode.com/en/problem/find-minimum-in-rotated-sorted-array-ii/)

### Problem

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.

The array may contain duplicates.

#### Example

Given [4,4,5,6,7,0,1,2] return 0

### 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];  
 }  
 }  
};

### Java

public class Solution {  
 /\*\*  
 \* @param num: a rotated sorted array  
 \* @return: the minimum number in the array  
 \*/  
 public int findMin(int[] num) {  
 if (num == null || num.length == 0) return Integer.MIN\_VALUE;  
  
 int lb = 0, ub = num.length - 1;  
 // case1: num[0] < num[num.length - 1]  
 // if (num[lb] < num[ub]) return num[lb];  
  
 // case2: num[0] > num[num.length - 1] or num[0] < num[num.length - 1]  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (num[mid] < num[ub]) {  
 ub = mid;  
 } else if (num[mid] > num[ub]){  
 lb = mid;  
 } else {  
 ub--;  
 }  
 }  
  
 return Math.min(num[lb], num[ub]);  
 }  
}

### 源码分析

注意 num[mid] > num[ub] 时应递减 ub 或者递增 lb.

### 复杂度分析

最坏情况下 O(n)O(n)O(n), 平均情况下 O(logn)O(\log n)O(logn).

## Median of two Sorted Arrays

# Median of two Sorted Arrays

### Source

* leetcode: [Median of Two Sorted Arrays | LeetCode OJ](https://leetcode.com/problems/median-of-two-sorted-arrays/)
* lintcode: [(65) Median of two Sorted Arrays](http://www.lintcode.com/en/problem/median-of-two-sorted-arrays/)

### Problem

There are two sorted arrays *A* and *B* of size *m* and *n* respectively. Find the **median** of the two sorted arrays.

#### Example

Given A=[1,2,3,4,5,6] and B=[2,3,4,5] , the median is 3.5 .

Given A=[1,2,3] and B=[4,5] , the median is 3 .

#### Challenge

The overall run time complexity should be O(log (m+n)).

### 题解1 - 归并排序

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](http://zh.wikipedia.org/wiki/%E4%B8%AD%E4%BD%8D%E6%95%B8)。简单粗暴的方法就是使用归并排序的思想，挨个比较两个数组的值，取小的，最后分奇偶长度返回平均值或者中位值。

### Java1 - merge sort with equal length

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) {  
 if ((A == null || A.length == 0) && (B == null || B.length == 0)) {  
 return -1.0;  
 }  
 int lenA = (A == null) ? 0 : A.length;  
 int lenB = (B == null) ? 0 : B.length;  
 int len = lenA + lenB;  
  
 /\* merge sort \*/  
 int indexA = 0, indexB = 0, indexC = 0;  
 int[] C = new int[len];  
 // case1: both A and B have elements  
 while (indexA < lenA && indexB < lenB) {  
 if (A[indexA] < B[indexB]) {  
 C[indexC++] = A[indexA++];  
 } else {  
 C[indexC++] = B[indexB++];  
 }  
 }  
 // case2: only A has elements  
 while (indexA < lenA) {  
 C[indexC++] = A[indexA++];  
 }  
 // case3: only B has elements  
 while (indexB < lenB) {  
 C[indexC++] = B[indexB++];  
 }  
  
 // return median for even and odd cases  
 int indexM1 = (len - 1) / 2, indexM2 = len / 2;  
 if (len % 2 == 0) {  
 return (C[indexM1] + C[indexM2]) / 2.0;  
 } else {  
 return C[indexM2];  
 }  
 }  
}

### Java2 - space optimization

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) {  
 if ((A == null || A.length == 0) && (B == null || B.length == 0)) {  
 return -1.0;  
 }  
 int lenA = (A == null) ? 0 : A.length;  
 int lenB = (B == null) ? 0 : B.length;  
 int len = lenA + lenB;  
 int indexM1 = (len - 1) / 2, indexM2 = len / 2;  
 int m1 = 0, m2 = 0;  
  
 /\* merge sort \*/  
 int indexA = 0, indexB = 0, indexC = 0;  
 // case1: both A and B have elements  
 while (indexA < lenA && indexB < lenB) {  
 if (indexC > indexM2) {  
 break;  
 }  
 if (indexC == indexM1) {  
 m1 = Math.min(A[indexA], B[indexB]);  
 }  
 if (indexC == indexM2) {  
 m2 = Math.min(A[indexA], B[indexB]);  
 }  
 if (A[indexA] < B[indexB]) {  
 indexA++;  
 } else {  
 indexB++;  
 }  
 indexC++;  
 }  
 // case2: only A has elements  
 while (indexA < lenA) {  
 if (indexC > indexM2) {  
 break;  
 }  
 if (indexC == indexM1) {  
 m1 = A[indexA];  
 }  
 if (indexC == indexM2) {  
 m2 = A[indexA];  
 }  
 indexA++;  
 indexC++;  
 }  
 // case3: only B has elements  
 while (indexB < lenB) {  
 if (indexC > indexM2) {  
 break;  
 }  
 if (indexC == indexM1) {  
 m1 = B[indexB];  
 }  
 if (indexC == indexM2) {  
 m2 = B[indexB];  
 }  
 indexB++;  
 indexC++;  
 }  
  
 // return median for even and odd cases  
 if (len % 2 == 0) {  
 return (m1 + m2) / 2.0;  
 } else {  
 return m2;  
 }  
 }  
}

### 源码分析

使用归并排序的思想做这道题不难，但是边界条件的处理比较闹心，使用归并排序带辅助空间的做法实现起来比较简单，代码也短。如果不使用额外空间并做一定优化的话需要多个 if 语句进行判断，需要注意的是多个 if 之间不能使用 else ，因为 indexM1 和 indexM2 有可能相等。

### 复杂度分析

时间复杂度 O(m+n)O(m + n)O(m+n), 空间复杂度为 (m+n)(m + n)(m+n)(使用额外数组), 或者 O(1)O(1)O(1)(不使用额外数组).

### 题解2 - 二分搜索

题中已有信息两个数组均为有序，找中位数的关键在于找到第一半大的数，显然可以使用二分搜索优化。本题是找中位数，其实可以泛化为一般的找第 k 大数，这个辅助方法的实现非常有意义！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...

由于是找第k大数(从1开始)，使用二分法则需要比较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++

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)) / 2.0);  
 } 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<int>::size\_type B\_start, int k) {  
 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);  
 }  
 }  
};

### 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) {  
 if ((A == null || A.length == 0) && (B == null || B.length == 0)) {  
 return -1.0;  
 }  
 int lenA = (A == null) ? 0 : A.length;  
 int lenB = (B == null) ? 0 : B.length;  
 int len = lenA + lenB;  
  
 // return median for even and odd cases  
 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);  
 }  
 }  
  
 private int findKth(int[] A, int indexA, int[] B, int indexB, int k) {  
  
 int lenA = (A == null) ? 0 : A.length;  
 if (indexA > lenA - 1) {  
 return B[indexB + k - 1];  
 }  
 int lenB = (B == null) ? 0 : B.length;  
 if (indexB > lenB - 1) {  
 return A[indexA + k - 1];  
 }  
  
 // avoid infilite loop if k == 1  
 if (k == 1) return Math.min(A[indexA], B[indexB]);  
  
 int keyA = Integer.MAX\_VALUE, keyB = Integer.MAX\_VALUE;  
 if (indexA + k/2 - 1 < lenA) keyA = A[indexA + k/2 - 1];  
 if (indexB + k/2 - 1 < lenB) keyB = B[indexB + k/2 - 1];  
  
 if (keyA > keyB) {  
 return findKth(A, indexA, B, indexB + k/2, k - k/2);  
 } else {  
 return findKth(A, indexA + k/2, B, indexB, k - k/2);  
 }  
 }  
}

### 源码分析

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。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。以 Java 的代码为例。

1. 首先在主程序中排除 A, B 均为空的情况。
2. 排除 A 或者 B 中有一个为空或者长度为0的情况。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。
3. k为1时，无需再递归调用，直接返回较小值。如果 k 为1不返回将导致后面的无限循环。
4. 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。
5. 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。
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

### 复杂度分析

找中位数，K 为数组长度和的一半，故总的时间复杂度为 O(log(m+n))O(\log (m+n))O(log(m+n)).

### Reference

* [九章算法 | Median of Two Sorted Arrays](http://www.jiuzhang.com/solutions/median-of-two-sorted-arrays/)
* [LeetCode: Median of Two Sorted Arrays 解题报告 - Yu's Garden - 博客园](http://www.cnblogs.com/yuzhangcmu/p/4138184.html)

## Sqrt x

# Sqrt x

### Source

* leetcode: [Sqrt(x) | LeetCode OJ](https://leetcode.com/problems/sqrtx/)
* lintcode: [(141) Sqrt(x)](http://www.lintcode.com/en/problem/sqrtx/)

### 题解 - 二分搜索

由于只需要求整数部分，故对于任意正整数 xxx, 设其整数部分为 kkk, 显然有 1≤k≤x1 \leq k \leq x1≤k≤x, 求解 kkk 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≤k≤end start \leq k \leq endstart≤k≤end 关系一直存在，也就是说在没有找到 mid2==xmid^2 == xmid2==x 时，循环退出时有 start<k<endstart < k < endstart<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(logn)O(\log n)O(logn), 使用了 start , end , mid 变量，空间复杂度为 O(1)O(1)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](http://www.nowamagic.net/algorithm/algorithm_EfficacyOfFunctionSqrt.php)，不得不感叹算法的魔力！

## Wood Cut

# Wood Cut

### Source

* lintcode: [(183) Wood Cut](http://www.lintcode.com/en/problem/wood-cut/)

### Problem

Given n pieces of wood with length L[i] (integer array). Cut them into smallpieces to guarantee you could have equal or more than k pieces with the samelength. 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素其实不容易，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：∑i=1nL[i]l≥k\sum \_{i = 1} ^{n} \frac {L[i]}{l} \geq k∑i=1nlL[i]≥k

其中 lll 为单段最大长度，显然有 1≤l≤max(L[i])1 \leq l \leq max(L[i])1≤l≤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](http://algorithm.yuanbin.me/zh-cn/binary_search/sqrt_x.html) 差不多一样了，要求的是 lll 的最大可能取值，同时 lll 可以看做是从有序序列 [1, max(L[i])] 的一个元素，典型的二分搜素！

P.S. 关于二分搜索总结在 [Binary Search](http://algorithm.yuanbin.me/zh-cn/basics_algorithm/binary_search.html) 一小节，直接套用『模板二——最优化』即可。

### 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

### Java

public 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.  
 \*/  
 public int woodCut(int[] L, int k) {  
 if (L == null || L.length == 0) return 0;  
  
 int lb = 0, ub = Integer.MAX\_VALUE;  
 while (lb + 1 < ub) {  
 int mid = lb + (ub - lb) / 2;  
 if (C(L, k, mid)) {  
 lb = mid;  
 } else {  
 ub = mid;  
 }  
 }  
  
 return lb;  
 }  
  
 // whether it cut with length x and get more than k pieces  
 private boolean C(int[] L, int k, int x) {  
 int sum = 0;  
 for (int l : L) {  
 sum += l / x;  
 }  
 return sum >= k;  
 }  
}

### 源码分析

定义私有方法 C 为切分为 x 长度时能否大于等于 k 段。若满足条件则更新 lb , 由于 lb 和 ub 的初始化技巧使得我们无需单独对最后的 lb 和 ub 单独求和判断。九章算法网站上的方法初始化为1和某最大值，还需要单独判断，虽然不会出bug, 但稍显复杂。这个时候lb, ub初始化为两端不满足条件的值的优雅之处就体现出来了。

### 复杂度分析

遍历求和时间复杂度为 O(n)O(n)O(n), 二分搜索时间复杂度为 O(logmax(L))O(\log max(L))O(logmax(L)). 故总的时间复杂度为 O(nlogmax(L))O(n \log max(L))O(nlogmax(L)). 空间复杂度 O(1)O(1)O(1).

### Reference

* [Binary Search](http://algorithm.yuanbin.me/zh-cn/basics_algorithm/binary_search.html)

# Math and Bit Manipulation

# 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](http://www.matrix67.com/blog/archives/263)
* *cc150* chapter 8.5 and chapter 9.5

## Single Number

# 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](http://www.lintcode.com/en/problem/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++

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

# Single Number II

### Source

* lintcode: [(83) Single Number II](http://www.lintcode.com/en/problem/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)⋅sizeof(int)⋅8)O((3n+1)\cdot sizeof(int) \cdot 8)O((3n+1)⋅sizeof(int)⋅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⋅sizeof(int)−18 \cdot sizeof(int) - 18⋅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](https://leetcode.com/discuss/857/constant-space-solution?show=2542) 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 single one which appears l times.

@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

# Single Number III

### Source

* lintcode: [(84) Single Number III](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/single_number.html) 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

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](http://algorithm.yuanbin.me/zh-cn/basics_misc/bit_manipulation.html)。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)O(n)O(n), 使用了部分额外空间，空间复杂度 O(1)O(1)O(1).

### Reference

* [Single Number III 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/single-number-iii/)

## O1 Check Power of 2

# O(1) Check Power of 2

### Source

* lintcode: [(142) O(1) Check Power of 2](http://www.lintcode.com/en/problem/o1-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)O(1)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

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)O(1)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](http://www.geeksforgeeks.org/next-power-of-2/)，有兴趣的可以去围观下。

## Convert Integer A to Integer B

# Convert Integer A to Integer B

### Source

* CC150, lintcode: [(181) Convert Integer A to Integer B](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/o1_check_power_of_2.html) 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

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](https://wiki.python.org/moin/BitManipulation)
* [5. Expressions — Python 2.7.10rc0 documentation](https://docs.python.org/2/reference/expressions.html#shifting)
* [Python之位移操作符所带来的困惑 - 旁观者 - 博客园](http://www.cnblogs.com/zhengyun_ustc/archive/2009/10/14/shifting.html)

## Factorial Trailing Zeroes

## Unique Binary Search Trees

## Update Bits

# Update Bits

### Source

* CTCI: [(179) Update Bits](http://www.lintcode.com/en/problem/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)O(1)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)O(1)O(1).

### Reference

* [c++ - logical shift right on signed data - Stack Overflow](http://stackoverflow.com/questions/13221369/logical-shift-right-on-signed-data)
* [Update Bits | 九章算法](http://www.jiuzhang.com/solutions/update-bits/)
* *CTCI 5th Chapter 9.5 中文版* p163

## Fast Power

# Fast Power

### Source

* lintcode: [(140) Fast Power](http://www.lintcode.com/en/problem/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 ana^nan 具体值求出来，太大了... 所以利用以上求模公式，可以改写 ana^nan 为：

an=an/2⋅an/2=an/4⋅an/4⋅an/4⋅an/4⋅=...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 = ...an=an/2⋅an/2=an/4⋅an/4⋅an/4⋅an/4⋅=...

至此递归模型建立。

### 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 a0a^0a0 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)O(1)O(1), 递归层数约为 logn\log nlogn, 时间复杂度为 O(logn)O(\log n)O(logn), 栈空间复杂度也为 O(logn)O(\log n)O(logn).

### Reference

* [Lintcode: Fast Power 解题报告 - Yu's Garden - 博客园](http://www.cnblogs.com/yuzhangcmu/p/4174781.html)
* [Fast Power 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/fast-power/)

## Hash Function

## Count 1 in Binary

# Count 1 in Binary

### Source

* lintcode: [(365) Count 1 in Binary](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/o1_check_power_of_2.html) 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

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)O(m)O(m). 空间复杂度 O(1)O(1)O(1).

### Reference

* [Number of 1 bits | LeetCode](http://articles.leetcode.com/2010/09/number-of-1-bits.html) - 评论中有关于不同算法性能的讨论

## Fibonacci

# Fibonacci

### Source

* lintcode: [(366) Fibonacci](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 使用了两个额外变量，空间复杂度为 O(1)O(1)O(1).

## A plus B Problem

# A plus B Problem

### Source

* lintcode: [(1) A + B Problem](http://www.lintcode.com/en/problem/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

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)O(1)O(1). 使用了部分额外变量，空间复杂度为 O(1)O(1)O(1).

## Print Numbers by Recursion

## Majority Number

# Majority Number

### Source

* leetcode: [Majority Element | LeetCode OJ](https://leetcode.com/problems/majority-element/)
* lintcode: [(46) Majority Number](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Majority Number II

# Majority Number II

### Source

* leetcode: [Majority Element II | LeetCode OJ](https://leetcode.com/problems/majority-element-ii/)
* lintcode: [(47) Majority Number II](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/majority_number.html) 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

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)O(n)O(n), 空间复杂度 O(2×2)=O(1)O(2 \times 2) = O(1)O(2×2)=O(1).

### Reference

* [Majority Number II 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/majority-number-ii/)

## Majority Number III

# Majority Number III

### Source

* lintcode: [(48) Majority Number III](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/majority_number_ii.html) 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

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)O(n)O(n), 使用了哈希表，空间复杂度 O(k)O(k)O(k).

### Reference

* [Majority Number III 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/majority-number-iii/)

## Digit Counts

# Digit Counts

### Source

* leetcode: [Number of Digit One | LeetCode OJ](https://leetcode.com/problems/number-of-digit-one/)
* lintcode: [(3) Digit Counts](http://www.lintcode.com/en/problem/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

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×L)O(n \times L)O(n×L), L 为n 的最大长度，拆成字符数组，空间复杂度 O(L)O(L)O(L).

## Ugly Number

## Plus One

# Plus One

### Source

* leetcode: [Plus One | LeetCode OJ](https://leetcode.com/problems/plus-one/)
* lintcode: [(407) Plus One](http://www.lintcode.com/en/problem/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)O(n)O(n), 空间复杂度在最后一位有进位时恶化为 O(n)O(n)O(n), 当然也可以通过两次循环使得空间复杂度为 O(1)O(1)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

# 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

# Remove Duplicates from Sorted List

### Source

* leetcode: [Remove Duplicates from Sorted List | LeetCode OJ](https://leetcode.com/problems/remove-duplicates-from-sorted-list/)
* lintcode: [(112) Remove Duplicates from Sorted List](http://www.lintcode.com/en/problem/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

# 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)O(n)O(n), 使用了一个中间变量进行遍历，空间复杂度为 O(1)O(1)O(1).

### Reference

* [Remove Duplicates from Sorted List 参考程序 | 九章](http://www.jiuzhang.com/solutions/remove-duplicates-from-sorted-list/)

## Remove Duplicates from Sorted List II

# ￼Remove Duplicates from Sorted List II

### Source

* leetcode: [Remove Duplicates from Sorted List II | LeetCode OJ](https://leetcode.com/problems/remove-duplicates-from-sorted-list-ii/)
* lintcode: [(113) Remove Duplicates from Sorted List II](http://www.lintcode.com/en/problem/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：

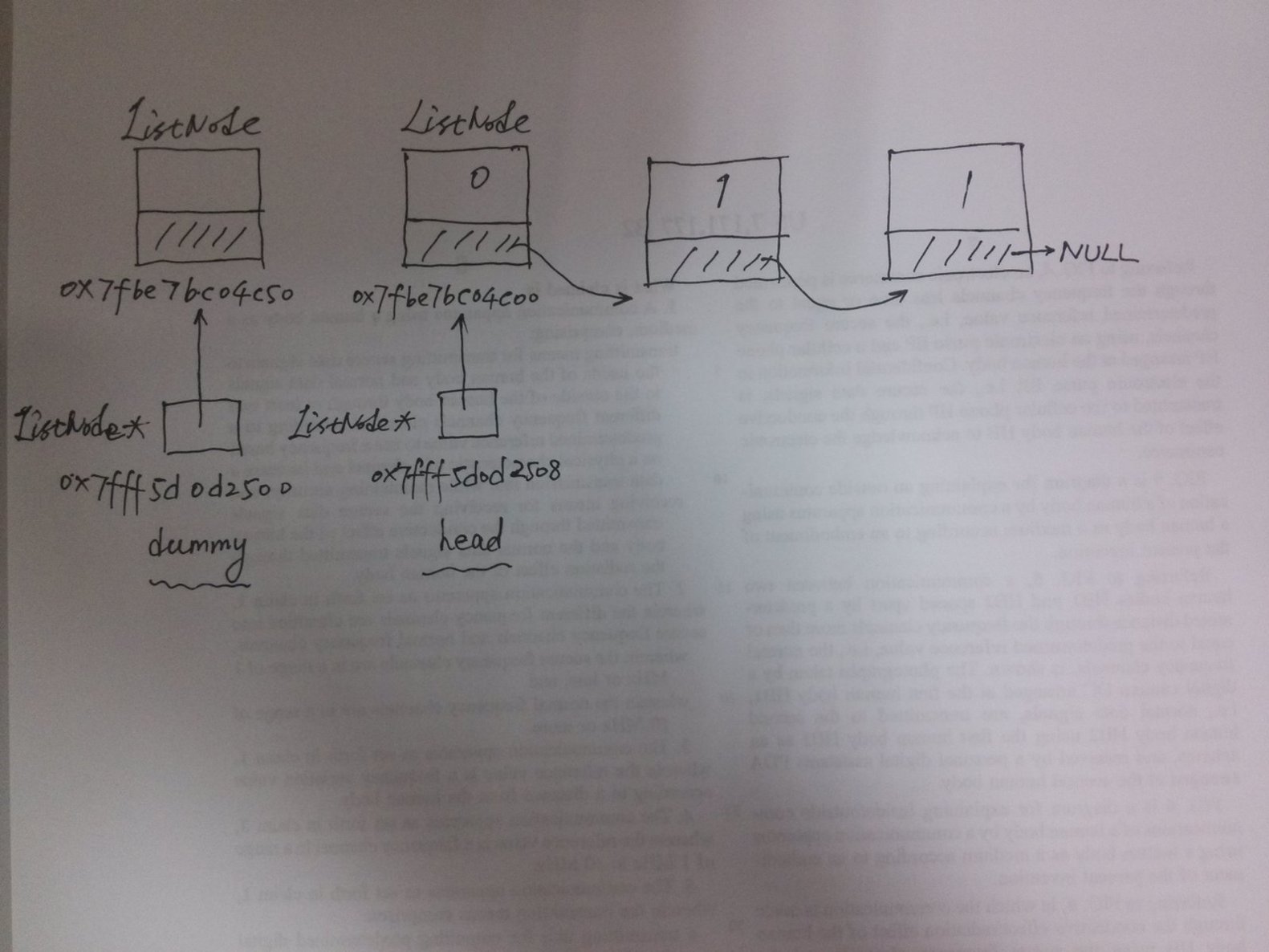
### C++ - Wrong

/\*\*  
 \* 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; }  
 \* }  
 \*/  
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)O(2n)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)O(1)O(1).

### Reference

* [Remove Duplicates from Sorted List II | 九章](http://www.jiuzhang.com/solutions/remove-duplicates-from-sorted-list-ii/)

## Remove Duplicates from Unsorted List

# Remove Duplicates from Unsorted List

### Source

* [Remove duplicates from an unsorted linked list - GeeksforGeeks](http://www.geeksforgeeks.org/remove-duplicates-from-an-unsorted-linked-list/)

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

"""  
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;  
 \* }  
 \* }  
 \*/  
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(12n2)O(\frac{1}{2}n^2)O(21n2), 空间复杂度近似为 O(1)O(1)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

"""  
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)O(n)O(n), 使用了额外的哈希表，空间复杂度近似为 O(n)O(n)O(n).

### Reference

* [Remove duplicates from an unsorted linked list - GeeksforGeeks](http://www.geeksforgeeks.org/remove-duplicates-from-an-unsorted-linked-list/)
* [ctci/Question.java at master · gaylemcd/ctci](https://github.com/gaylemcd/ctci/blob/master/java/Chapter%202/Question2_1/Question.java)

## Partition List

# Partition List

### Source

* leetcode: [Partition List | LeetCode OJ](https://leetcode.com/problems/partition-list/)
* lintcode: [(96) Partition List](http://www.lintcode.com/en/problem/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

"""  
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

/\*\*  
 \* 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)O(n)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)O(1)O(1).

## Two Lists Sum

# Two Lists Sum

### Source

* CC150 - [(167) Two Lists Sum](http://www.lintcode.com/en/problem/two-lists-sum/)

You have two numbers represented by a linked list, where each node contains a single digit.  
The digits are stored in reverse order, such that the 1’s digit is at the head of the list.  
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

/\*\*  
 \* 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))O(max(L1, L2))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](http://www.geeksforgeeks.org/add-two-numbers-represented-by-linked-lists/)

## Two Lists Sum Advanced

# Two Lists Sum Advanced

### Source

* CC150 - [Add two numbers represented by linked lists | Set 2 - GeeksforGeeks](http://www.geeksforgeeks.org/sum-of-two-linked-lists/)

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](http://algorithm.yuanbin.me/zh-cn/linked_list/two_lists_sum.html) 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](http://www.geeksforgeeks.org/sum-of-two-linked-lists/)

## Remove Nth Node From End of List

# Remove Nth Node From End of List

### Source

* lintcode: [(174) Remove Nth Node From End of List](http://www.lintcode.com/en/problem/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

# Linked List Cycle

### Source

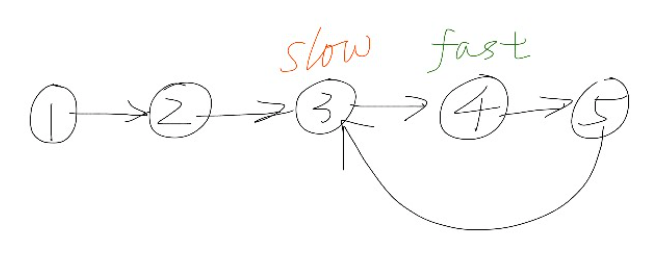
* leetcode: [Linked List Cycle | LeetCode OJ](https://leetcode.com/problems/linked-list-cycle/)
* lintcode: [(102) Linked List Cycle](http://www.lintcode.com/en/problem/linked-list-cycle/)

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)O(n/2)O(n/2).
2. 有环时，最坏的时间复杂度近似为 O(n)O(n)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)O(n)O(n).

### Reference

* [Linked List Cycle | 九章算法](http://www.jiuzhang.com/solutions/linked-list-cycle/)

## Linked List Cycle II

# Linked List Cycle II

### Source

* leetcode: [Linked List Cycle II | LeetCode OJ](https://leetcode.com/problems/linked-list-cycle-ii/)
* lintcode: [(103) Linked List Cycle II](http://www.lintcode.com/en/problem/linked-list-cycle-ii/)

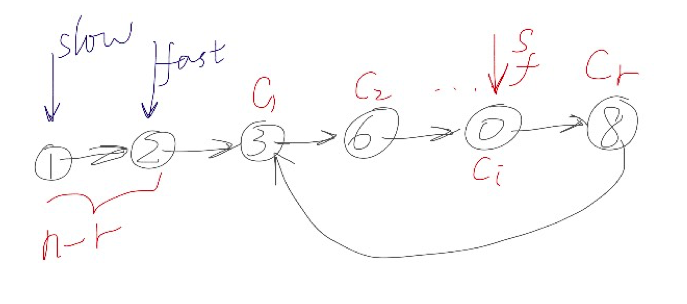
Given a linked list, return the node where the cycle begins. If there is no cycle, return null.  
  
Example  
Given -21->10->4->5, tail connects to node index 1，return node 10  
  
Challenge  
Follow up:  
Can you solve it without using extra space?

### 题解 - 快慢指针

题 [Linked List Cycle | Data Structure and Algorithm](http://algorithm.yuanbin.me/zh-cn/linked_list/linked_list_cycle.html) 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 rrr, 链表中节点个数为 nnn. 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 CiC\_iCi, 那么此时慢指针正好走了 n−r−1+in - r - 1 + in−r−1+i 步，快指针则走了 2⋅(n−r−1+i)2 \cdot (n - r - 1 + i)2⋅(n−r−1+i) 步，且存在[1](#): n−r−1+i+1=l⋅rn - r - 1 + i + 1= l \cdot rn−r−1+i+1=l⋅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⋅r−i+1)(l \cdot r - i + 1)(l⋅r−i+1) 个节点 (lll 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⋅r−in - r = l \cdot r - in−r=l⋅r−i. 从头节点走到环的入口节点所走的步数可用 n−rn - rn−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)O(n)O(n), 第二次相遇的最坏时间复杂度为 O(n)O(n)O(n). 故总的时间复杂度近似为 O(n)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

### Reference

* [Linked List Cycle II | 九章算法](http://www.jiuzhang.com/solutions/linked-list-cycle-ii/)

## Reverse Linked List

# ￼Reverse Linked List

### Source

* leetcode: [Reverse Linked List | LeetCode OJ](https://leetcode.com/problems/reverse-linked-list/)
* lintcode: [(35) Reverse Linked List](http://www.lintcode.com/en/problem/reverse-linked-list/)

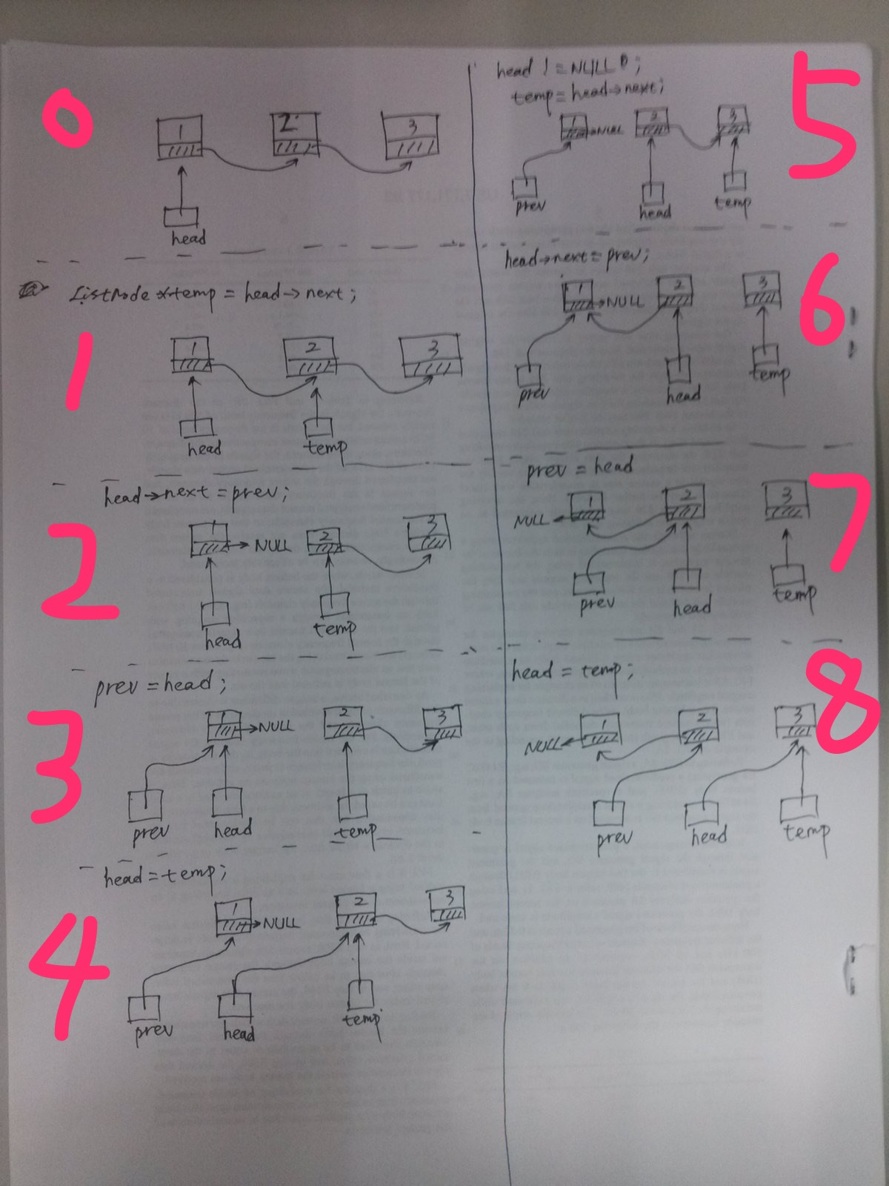
Reverse a linked list.  
  
Example  
For linked list 1->2->3, the reversed linked list is 3->2->1  
  
Challenge  
Reverse it in-place and in one-pass

### 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

# 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++

/\*\*  
 \* 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)O(n)O(n), 使用了辅助变量，空间复杂度 O(1)O(1)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)O(n)O(n), 时间复杂度为 O(n)O(n)O(n), 空间(不含栈空间)复杂度为 O(1)O(1)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](http://www.cnblogs.com/kubixuesheng/p/4394509.html)
* [data structures - Reversing a linked list in Java, recursively - Stack Overflow](http://stackoverflow.com/questions/354875/reversing-a-linked-list-in-java-recursively)
* [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](http://ceeji.net/blog/reserve-linked-list-cpp/)
* [iteratively and recursively Java Solution - Leetcode Discuss](https://leetcode.com/discuss/37804/iteratively-and-recursively-java-solution)

## Reverse Linked List II

# Reverse Linked List II

### Source

* leetcode: [Reverse Linked List II | LeetCode OJ](https://leetcode.com/problems/reverse-linked-list-ii/)
* lintcode: [(36) Reverse Linked List II](http://www.lintcode.com/en/problem/reverse-linked-list-ii/)

### Problem

Reverse a linked list from position m to n.

#### Example

Given **1->2->3->4->5->NULL**, m = 2 and n = 4, return1->4->3->2->5->NULL.

#### Note

Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length of list.

#### 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;  
 }  
};

### 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   
 \* @oaram m and n  
 \* @return: The head of the reversed ListNode  
 \*/  
 public ListNode reverseBetween(ListNode head, int m , int n) {  
 ListNode dummy = new ListNode(0);  
 dummy.next = head;  
  
 // find the mth node  
 ListNode premNode = dummy;  
 for (int i = 1; i < m; i++) {  
 premNode = premNode.next;  
 }  
  
 // reverse node between m and n  
 ListNode prev = null, curr = premNode.next;  
 while (curr != null && (m <= n)) {  
 ListNode nextNode = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = nextNode;  
 m++;  
 }  
  
 // join head and tail before m and after n  
 premNode.next.next = curr;  
 premNode.next = prev;  
  
 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

# Merge Two Sorted Lists

### Source

* lintcode: [(165) Merge Two Sorted Lists](http://www.lintcode.com/en/problem/merge-two-sorted-lists/)
* leetcode: [Merge Two Sorted Lists | LeetCode OJ](https://leetcode.com/problems/merge-two-sorted-lists/)

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++

/\*\*  
 \* 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)O(1)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)O(l1+l2)O(l1+l2). 空间复杂度近似为 O(1)O(1)O(1).

### Reference

* [Merge Two Sorted Lists | 九章算法](http://www.jiuzhang.com/solutions/merge-two-sorted-lists/)

## Merge k Sorted Lists

## Reorder List

## Copy List with Random Pointer

## Sort List

## Insertion Sort List

## Check if a singly linked list is palindrome

# Check if a singly linked list is palindrome

* tags: [palindrome, linked\_list]

### Source

* [Function to check if a singly linked list is palindrome - GeeksforGeeks](http://www.geeksforgeeks.org/function-to-check-if-a-singly-linked-list-is-palindrome/)

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

/\*\*  
 \* 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(12n)O(\frac{1}{2}n)O(21n), 分别遍历链表的前半部分和后半部分，时间复杂度为 O(n)O(n)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)O(n)O(n), 使用了几个临时遍历，空间复杂度为 O(1)O(1)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)O(n)O(n), 使用了几个临时变量，空间复杂度为 O(1)O(1)O(1).

### Reference

* [Function to check if a singly linked list is palindrome - GeeksforGeeks](http://www.geeksforgeeks.org/function-to-check-if-a-singly-linked-list-is-palindrome/)
* [回文判断 | The-Art-Of-Programming-By-July/01.04.md](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.04.md)
* [ctci/QuestionB.java at master · gaylemcd/ctci](https://github.com/gaylemcd/ctci/blob/master/java/Chapter%202/Question2_7/QuestionB.java)

## Delete Node in the Middle of Singly Linked List

# Delete Node in the Middle of Singly Linked List

### Source

* lintcode: [(372) Delete Node in the Middle of Singly Linked List](http://www.lintcode.com/en/problem/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

/\*\*  
 \* 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)O(1)O(1).

## Rotate List

# Rotate List

### Source

* leetcode: [Rotate List | LeetCode OJ](https://leetcode.com/problems/rotate-list/)
* lintcode: [(170) Rotate List](http://www.lintcode.com/en/problem/rotate-list/)

### Problem

Given a list, rotate the list to the right by *k* places, where *k* is non-negative.

#### Example

Given 1->2->3->4->5 and k = 2 , return 4->5->1->2->3 .

### 题解

旋转链表，链表类问题通常需要找到需要处理节点处的前一个节点。因此我们只需要找到旋转节点和最后一个节点即可。需要注意的细节是 k 有可能比链表长度还要大，此时需要取模，另一个 corner case 则是链表长度和 k 等长。

### Java

/\*\*  
 \* Definition for singly-linked list.  
 \* public class ListNode {  
 \* int val;  
 \* ListNode next;  
 \* ListNode(int x) {  
 \* val = x;  
 \* next = null;  
 \* }  
 \* }  
 \*/  
public class Solution {  
 /\*\*  
 \* @param head: the List  
 \* @param k: rotate to the right k places  
 \* @return: the list after rotation  
 \*/  
 public ListNode rotateRight(ListNode head, int k) {  
 if (head == null) return head;  
 ListNode fast = head, slow = head;  
 int len = 1;  
 for (len = 1; fast.next != null && len <= k; len++) {  
 fast = fast.next;  
 }  
 // k mod len if k > len  
 if (len <= k) {  
 k = k % len;  
 fast = head;  
 for (int i = 0; i < k; i++) {  
 fast = fast.next;  
 }  
 }  
 // forward slow and fast  
 while (fast.next != null) {  
 fast = fast.next;  
 slow = slow.next;  
 }  
 // return new head  
 fast.next = head;  
 head = slow.next;  
 slow.next = null;  
  
 return head;  
 }  
}

### 源码分析

由于需要处理的是节点的前一个节点，故最终的 while 循环使用 fast.next != null . k 与链表等长时包含在 len <= k 中。

### 复杂度分析

时间复杂度 O(n)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Swap Nodes in Pairs

# Swap Nodes in Pairs

### Source

* leetcode: [Swap Nodes in Pairs | LeetCode OJ](https://leetcode.com/problems/swap-nodes-in-pairs/)
* lintcode: [(451) Swap Nodes in Pairs](http://www.lintcode.com/en/problem/swap-nodes-in-pairs/)

### Problem

Given a linked list, swap every two adjacent nodes and return its head.

#### Example

Given 1->2->3->4 , you should return the list as 2->1->4->3 .

#### Challenge

Your algorithm should use only constant space. You may not modify the valuesin the list, only nodes itself can be changed.

### 题解1 - Iteration

直觉上我们能想到的是使用 dummy 处理不定头节点，但是由于这里是交换奇偶位置的链表节点，我们不妨首先使用伪代码来表示。大致可以分为如下几个步骤：

1. 保存 2.next
2. 将 2.next 赋值为 1
3. 将 1.next 赋值为1中保存的 2.next
4. 将前一个链表节点的 next 指向 1
5. 更新前一个链表节点为 1
6. 更新当前的链表节点为1中保存的 2.next

链表类题目看似容易，但要做到 bug-free 其实不容易，建议结合图像辅助分析，onsite 时不要急，把过程先用伪代码写出来。然后将伪代码逐行转化。

### Java

/\*\*  
 \* Definition for singly-linked list.  
 \* public class ListNode {  
 \* int val;  
 \* ListNode next;  
 \* ListNode(int x) { val = x; }  
 \* }  
 \*/  
public class Solution {  
 /\*\*  
 \* @param head a ListNode  
 \* @return a ListNode  
 \*/  
 public ListNode swapPairs(ListNode head) {  
 ListNode dummy = new ListNode(0);  
 dummy.next = head;  
 ListNode prev = dummy, curr = head;  
  
 while (curr != null && curr.next != null) {  
 ListNode after = curr.next;  
 ListNode nextCurr = after.next;  
 after.next = curr;  
 curr.next = nextCurr;  
 // link new node after prev  
 prev.next = after;  
 // update prev and curr  
 prev = curr;  
 curr = nextCurr;  
 }  
  
 return dummy.next;  
 }  
}

### 源码分析

这里使用 dummy 处理不定头节点，首先将 prev 初始化为 dummy , 然后按照题解中的几个步骤逐步转化，需要注意的是 while 循环中 curr 和 curr.next 都不能为 null .

### 复杂度分析

遍历链表一遍，时间复杂度 O(1)O(1)O(1). 使用了若干临时链表节点引用对象，空间复杂度 O(1)O(1)O(1).

### 题解2 - Recursion

在题解1 的分析过程中我们发现比较难处理的是 prev 和下一个头的连接，要是能直接得到链表后面新的头节点该有多好啊。首先我们可以肯定的是若 head == null || head.next == null 时应直接返回，如果不是则求得交换奇偶节点后的下一个头节点并链接到之前的奇数个节点。这种思想使用递归实现起来非常优雅！

### Java

/\*\*  
 \* Definition for singly-linked list.  
 \* public class ListNode {  
 \* int val;  
 \* ListNode next;  
 \* ListNode(int x) { val = x; }  
 \* }  
 \*/  
public class Solution {  
 /\*\*  
 \* @param head a ListNode  
 \* @return a ListNode  
 \*/  
 public ListNode swapPairs(ListNode head) {  
 if (head == null || head.next == null) return head;  
  
 ListNode after = head.next;  
 head.next = swapPairs(after.next);  
 after.next = head;  
  
 return after;  
 }  
}

### 源码分析

这个递归实现非常优雅，需要注意的是递归步的退出条件==> head == null || head.next == null) .

### 复杂度分析

每个节点最多被遍历若干次，时间复杂度 O(n)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Remove Linked List Elements

# Remove Linked List Elements

### Source

* leetcode: [Remove Linked List Elements | LeetCode OJ](https://leetcode.com/problems/remove-linked-list-elements/)
* lintcode: [(452) Remove Linked List Elements](http://www.lintcode.com/en/problem/remove-linked-list-elements/)

### Problem

Remove all elements from a linked list of integers that have value val .

#### Example

Given 1->2->3->3->4->5->3 , val = 3, you should return the list as 1->2->4->5

### 题解

删除链表中指定值，找到其前一个节点即可，将 next 指向下一个节点即可。

### Java

/\*\*  
 \* Definition for singly-linked list.  
 \* public class ListNode {  
 \* int val;  
 \* ListNode next;  
 \* ListNode(int x) { val = x; }  
 \* }  
 \*/  
public class Solution {  
 /\*\*  
 \* @param head a ListNode  
 \* @param val an integer  
 \* @return a ListNode  
 \*/  
 public ListNode removeElements(ListNode head, int val) {  
 ListNode dummy = new ListNode(0);  
 dummy.next = head;  
 ListNode curr = dummy;  
 while (curr.next != null) {  
 if (curr.next.val == val) {  
 curr.next = curr.next.next;  
 } else {  
 curr = curr.next;  
 }  
 }  
  
 return dummy.next;  
 }  
}

### 源码分析

while 循环中使用 curr.next 较为方便，if 语句中比较时也使用 curr.next.val 也比较简洁，如果使用 curr 会比较难处理。

### 复杂度分析

略

# Binary Tree

Maximum Depth of Binary Tree# Binary Tree - 二叉树

二叉树的基本概念在 [Binary Tree | Algorithm](http://algorithm.yuanbin.me/zh-cn/basics_data_structure/binary_tree.html) 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

# Binary Tree Preorder Traversal

### Source

* leetcode: [Binary Tree Preorder Traversal | LeetCode OJ](https://leetcode.com/problems/binary-tree-preorder-traversal/)
* lintcode: [(66) Binary Tree Preorder Traversal](http://www.lintcode.com/en/problem/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

"""  
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)O(n)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)O(n)O(n), 对每个节点遍历一次，时间复杂度近似为 O(n)O(n)O(n).

## Binary Tree Inorder Traversal

# Binary Tree Inorder Traversal

### Source

* leetcode: [Binary Tree Inorder Traversal | LeetCode OJ](https://leetcode.com/problems/binary-tree-inorder-traversal/)
* lintcode: [(67) Binary Tree Inorder Traversal](http://www.lintcode.com/en/problem/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

"""  
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)O(n)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)O(n)O(n), 时间复杂度 O(n)O(n)O(n).

### Reference

## Binary Tree Postorder Traversal

# Binary Tree Postorder Traversal

### Source

* leetcode: [Binary Tree Postorder Traversal | LeetCode OJ](https://leetcode.com/problems/binary-tree-postorder-traversal/)
* lintcode: [(68) Binary Tree Postorder Traversal](http://www.lintcode.com/en/problem/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

# 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)O(n)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

/\*\*  
 \* 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)O(n)O(n), 每个节点遍历两次或以上，时间复杂度近似为 O(n)O(n)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;  
 }  
};

### 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: Postorder in ArrayList which contains node values.  
 \*/  
 public ArrayList<Integer> postorderTraversal(TreeNode root) {  
 ArrayList<Integer> result = new ArrayList<Integer>();  
 if (root == null) return result;  
  
 Deque<TreeNode> stack = new ArrayDeque<TreeNode>();  
 stack.push(root);  
 while (!stack.isEmpty()) {  
 TreeNode node = stack.pop();  
 result.add(node.val);  
 if (node.left != null) stack.push(node.left);  
 if (node.right != null) stack.push(node.right);  
 }  
 Collections.reverse(result);  
  
 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](http://www.cnblogs.com/zuoyuan/p/3720846.html) - 解释清晰
* [更简单的非递归遍历二叉树的方法](http://zisong.me/post/suan-fa/geng-jian-dan-de-bian-li-er-cha-shu-de-fang-fa) - 比较新颖和简洁的实现

## Binary Tree Level Order Traversal

# Binary Tree Level Order Traversal

### Source

* lintcode: [(69) Binary Tree Level Order Traversal](http://www.lintcode.com/en/problem/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)O(n)O(n), 时间复杂度 O(n)O(n)O(n).

## Binary Tree Level Order Traversal II

## Maximum Depth of Binary Tree

# Maximum Depth of Binary Tree

### Source

* lintcode: [(97) Maximum Depth of Binary Tree](http://www.lintcode.com/en/problem/maximum-depth-of-binary-tree/)

### Problem

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the rootnode down to the farthest leaf node.

#### 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)O(n)O(n), 树的深度最大为 nnn, 最小为 log2n\log\_2 nlog2n, 故空间复杂度介于 O(logn)O(\log n)O(logn) 和 O(n)O(n)O(n) 之间。

### 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: 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

/\*\*  
 \* 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++

/\*\*  
 \* 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3 - 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。

### 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: 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

/\*\*  
 \* 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 depth = 0;  
 Queue<TreeNode> q = new LinkedList<TreeNode>();  
 q.offer(root);  
 while (!q.isEmpty()) {  
 depth++;  
 int qLen = q.size();  
 for (int i = 0; i < qLen; i++) {  
 TreeNode node = q.poll();  
 if (node.left != null) q.offer(node.left);  
 if (node.right != null) q.offer(node.right);  
 }  
 }  
  
 return depth;  
 }  
}

### 源码分析

广度优先中队列的使用中， qLen 需要在for 循环遍历之前获得，因为它是一个变量。

### 复杂度分析

最坏情况下空间复杂度为 O(n)O(n)O(n), 遍历每一个节点，时间复杂度为 O(n)O(n)O(n),

## Balanced Binary Tree

# Balanced Binary Tree

### Source

* lintcode: [(93) Balanced Binary Tree](http://www.lintcode.com/en/problem/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 depth of the two subtrees of every node never differ by more than 1.  
  
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](http://algorithm.yuanbin.me/zh-cn/binary_tree/maximum_depth_of_binary_tree.html) 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

## Lowest Common Ancestor

## Invert Binary Tree

# Invert Binary Tree

### Source

* leetcode: [Invert Binary Tree | LeetCode OJ](https://leetcode.com/problems/invert-binary-tree/)
* lintcode: [(175) Invert Binary Tree](http://www.lintcode.com/en/problem/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

/\*\*  
 \* 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)O(n)O(n), 使用了临时变量，空间复杂度为 O(1)O(1)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)O(n)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)O(n)O(n).

### Reference

* [0ms C++ Recursive/Iterative Solutions with Explanations - Leetcode Discuss](https://leetcode.com/discuss/42613/0ms-c-recursive-iterative-solutions-with-explanations)

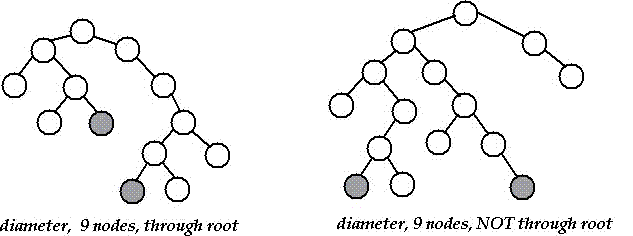
## Diameter of a Binary Tree

# Diameter of a Binary Tree

### Source

* [Diameter of a Binary Tree - GeeksforGeeks](http://www.geeksforgeeks.org/diameter-of-a-binary-tree/)

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).



### 题解

和题 [Lowest Common Ancestor](http://algorithm.yuanbin.me/zh-cn/binary_tree/lowest_common_ancestor.html) 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](http://www.geeksforgeeks.org/diameter-of-a-binary-tree/)
* [Diameter of a Binary Tree | Algorithms](http://algorithms.tutorialhorizon.com/diameter-of-a-binary-tree/)

## Construct Binary Tree from Preorder and Inorder Traversal

# Construct Binary Tree from Preorder and Inorder Traversal

### Source

* leetcode: [Construct Binary Tree from Preorder and Inorder Traversal | LeetCode OJ](https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/)
* lintcode: [(73) Construct Binary Tree from Preorder and Inorder Traversal](http://www.lintcode.com/en/problem/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

/\*\*  
 \* 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)O(n)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(n2)O(n^2)O(n2). 原地生成最终二叉树，空间复杂度为 O(1)O(1)O(1).

### Reference

* [Construct Binary Tree from Preorder and Inorder Traversal 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/construct-binary-tree-from-preorder-and-inorder-traversal/)

## Construct Binary Tree from Inorder and Postorder Traversal

# Construct Binary Tree from Inorder and Postorder Traversal

### Source

* lintcode: [(72) Construct Binary Tree from Inorder and Postorder Traversal](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/binary_tree/construct_binary_tree_from_preorder_and_inorder_traversal.html) 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

/\*\*  
 \* 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)O(n)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(n2)O(n^2)O(n2).

## Subtree

# Subtree

### Source

* lintcode: [(245) Subtree](http://www.lintcode.com/en/problem/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

/\*\*  
 \* 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)O(n)O(n), 查根节点的时间复杂度随机，平均为 O(m)O(m)O(m), 故总的时间复杂度可近似为 O(mn)O(mn)O(mn).

### Reference

* [LintCode: Subtree](http://cherylintcode.blogspot.com/2015/06/subtree.html)

## Binary Tree Zigzag Level Order Traversal

# Binary Tree Zigzag Level Order Traversal

### Source

* leetcode: [Binary Tree Zigzag Level Order Traversal | LeetCode OJ](https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/)
* lintcode: [(71) Binary Tree Zigzag Level Order Traversal](http://www.lintcode.com/en/problem/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)O(n)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)O(n/2)O(n/2).

总的时间复杂度为 O(n)O(n)O(n), 空间复杂度也为 O(n)O(n)O(n).

### Reference

* [Binary Tree Zigzag Level Order Traversal 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/binary-tree-zigzag-level-order-traversal/)
* [Printing a Binary Tree in Zig Zag Level-Order | LeetCode](http://articles.leetcode.com/2010/09/printing-binary-tree-in-zig-zag-level_18.html)

## Binary Tree Serialization

# Binary Tree Serialization

### Source

* lintcode: [(7) Binary Tree Serialization](http://www.lintcode.com/en/problem/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

/\*\*  
 \* 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).](https://gist.github.com/bittib/5620951)
* [Serialization/Deserialization of a Binary Tree | LeetCode](http://articles.leetcode.com/2010/09/serializationdeserialization-of-binary.html)

# Binary Search Tree

# Binary Search Tree - 二叉搜索树

二叉搜索树的定义及简介在 [Binary Search Trees](http://algorithm.yuanbin.me/zh-cn/basics_data_structure/binary_search_tree.html) 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

# Insert Node in a Binary Search Tree

### Source

* lintcode: [(85) Insert Node in a Binary Search Tree](http://www.lintcode.com/en/problem/insert-node-in-a-binary-search-tree/)

Given a binary search tree and a new tree node, insert the node into the tree. You should keep the tree still be a valid binary search tree.  
  
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

# Validate Binary Search Tree

### Source

* lintcode: [(95) Validate Binary Search Tree](http://www.lintcode.com/en/problem/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 = " + 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)O(n)O(n), 使用了部分额外空间，空间复杂度为 O(1)O(1)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](http://www.cnblogs.com/yuzhangcmu/p/4177047.html) - 提供了4种不同的方法，思路可以参考。

## Search Range in Binary Search Tree

# Search Range in Binary Search Tree

### Source

* lintcode: [(11) Search Range in Binary Search Tree](http://www.lintcode.com/en/problem/search-range-in-binary-search-tree/)

### Problem

Given two values k1 and k2 (where k1 < k2) and a root pointer to a BinarySearch Tree. Find all the keys of tree in range k1 to k2. i.e. print all xsuch that k1<=x<=k2 and x is a key of given BST. Return all the keys inascending order.

#### Example

If k1 = 10 and k2 = 22 , then your function should return [12, 20, 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++

/\*\*  
 \* 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);  
 }  
};

### 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 the binary search tree.  
 \* @param k1 and k2: range k1 to k2.  
 \* @return: Return all keys that k1<=key<=k2 in ascending order.  
 \*/  
 public ArrayList<Integer> searchRange(TreeNode root, int k1, int k2) {  
 ArrayList<Integer> result = new ArrayList<Integer>();  
 helper(root, k1, k2, result);  
  
 return result;  
 }  
  
 private void helper(TreeNode root, int k1, int k2, ArrayList<Integer> result) {  
 if (root == null) return;  
  
 // in-order binary tree iteration  
 helper(root.left, k1, k2, result);  
 if (k1 <= root.val && root.val <= k2) {  
 result.add(root.val);  
 }  
 helper(root.right, k1, k2, result);  
 }  
}

### 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

# Convert Sorted Array to Binary Search Tree

### Source

* leetcode: [Convert Sorted Array to Binary Search Tree | LeetCode OJ](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/)
* lintcode: [(177) Convert Sorted Array to Binary Search Tree With Minimal Height](http://www.lintcode.com/en/problem/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)O(n)O(n).

### Reference

* [Convert Sorted Array to Binary Search Tree | 九章算法](http://www.jiuzhang.com/solutions/convert-sorted-array-to-binary-search-tree/)

## Convert Sorted List to Binary Search Tree

# Convert Sorted List to Binary Search Tree

### Source

* leetcode - [Convert Sorted List to Binary Search Tree | LeetCode OJ](https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/)
* lintcode - [(106) Convert Sorted List to Binary Search Tree](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/binary_search_tree/convert_sorted_array_to_binary_search_tree.html) 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++

/\*\*  
 \* 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)O(n)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)O(n)O(n), 两者加起来总的时间复杂度仍为 O(n)O(n)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 solution above  
 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)O(nlogn)O(nlogn)，性能上还是比 O(n)O(n)O(n) 差一点。

### Reference

* [Convert Sorted List to Binary Search Tree | 九章算法](http://www.jiuzhang.com/solutions/convert-sorted-list-to-binary-search-tree/)

## Binary Search Tree Iterator

# Binary Search Tree Iterator

### Source

* lintcode: [(86) Binary Search Tree Iterator](http://www.lintcode.com/en/problem/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, 11, 12]  
  
 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

## Subsets

## Unique Subsets

# Unique Subsets

### Source

* leetcode: [Subsets II | LeetCode OJ](https://leetcode.com/problems/subsets-ii/)
* lintcode: [(18) Unique Subsets](http://www.lintcode.com/en/problem/unique-subsets/)

### Problem

Given a list of numbers that may has duplicate numbers, return all possible subsets.

#### Example

If ***S*** = [1,2,2] , a solution is:

[  
 [2],  
 [1],  
 [1,2,2],  
 [2,2],  
 [1,2],  
 []  
]

#### 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.

### 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](http://www.jiuzhang.com))。

以 [1,21,22][1, 2\_1, 2\_2][1,21,22] 为例，若不考虑重复，组合有 [],[1],[1,21],[1,21,22],[1,22],[21],[21,22],[22][], [1], [1, 2\_1], [1, 2\_1, 2\_2], [1, 2\_2], [2\_1], [2\_1, 2\_2], [2\_2][],[1],[1,21],[1,21,22],[1,22],[21],[21,22],[22]. 其中重复的有 [1,22],[22][1, 2\_2], [2\_2][1,22],[22]. 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] (不是 i + 1, 可能索引越界，而i 不等于 pos 已经能保证 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();  
 }  
 }  
};

### Java

class Solution {  
 /\*\*  
 \* @param S: A set of numbers.  
 \* @return: A list of lists. All valid subsets.  
 \*/  
 public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayList<Integer> S) {  
 ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
 if (S == null) return result;  
 //   
 Collections.sort(S);  
 List<Integer> list = new ArrayList<Integer>();  
 dfs(S, 0, list, result);  
 return result;  
 }  
  
 private void dfs(ArrayList<Integer> S, int pos, List<Integer> list,   
 ArrayList<ArrayList<Integer>> result) {  
  
 result.add(new ArrayList<Integer>(list));  
 for (int i = pos; i < S.size(); i++) {  
 // exlude duplicate  
 if (i != pos && S.get(i) == S.get(i - 1)) {  
 continue;  
 }  
 list.add(S.get(i));  
 dfs(S, i + 1, list, result);  
 list.remove(list.size() - 1);  
 }  
 }  
}

### 源码分析

相比前一道题多了去重的判断。

### 复杂度分析

和前一道题差不多，最坏情况下时间复杂度为 2n2^n2n. 空间复杂度为 O(n)O(n)O(n).

### Reference

* [Subsets II | 九章算法](http://www.jiuzhang.com/solutions/subsets-ii/)

## Permutations

# Permutations

### Source

* leetcode: [Permutations | LeetCode OJ](https://leetcode.com/problems/permutations/)
* lintcode: [(15) Permutations](http://www.lintcode.com/en/problem/permutations/)

### Problem

Given a list of numbers, return all possible permutations.

#### Example

For nums = [1,2,3] , the permutations 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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/subsets.html) 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

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

public class Solution {  
 public List<List<Integer>> permute(int[] nums) {  
 List<List<Integer>> result = new ArrayList<List<Integer>>();  
 if (nums == null || nums.length == 0) return result;  
  
 List<Integer> list = new ArrayList<Integer>();  
 dfs(nums, list, result);  
  
 return result;  
 }  
  
 private void dfs(int[] nums, List<Integer> list, List<List<Integer>> result) {  
 if (list.size() == nums.length) {  
 result.add(new ArrayList<Integer>(list));  
 return;  
 }  
  
 for (int i = 0; i < nums.length; i++) {  
 if (list.contains(nums[i])) continue;  
 list.add(nums[i]);  
 dfs(nums, list, result);  
 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)O(N)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)O(N)O(N), 查找的时间复杂度为 O(1)O(1)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!n!n!, 每个节点被遍历的次数为 (n−1)!(n-1)!(n−1)!, 故节点共被遍历的状态数为 O(n!)O(n!)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 nnn^nnn 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⋅n!)O(n \cdot n!)O(n⋅n!), 上界为 n⋅nnn \cdot n^nn⋅nn. 实测 helper 中 for 循环的遍历次数在 O(2n⋅n!)O(2n \cdot n!)O(2n⋅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!)O(n!)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>>();  
 if (nums == null || nums.length == 0) return result;  
  
 Arrays.sort(nums);  
 while (true) {  
 // step0: add nums into result  
 List<Integer> list = new ArrayList<Integer>();  
 for (int i : nums) {  
 list.add(i);  
 }  
 result.add(list);  
  
 // step2: find the first nums[k] < nums[k + 1] from the end to start  
 int k = -1;  
 for (int i = nums.length - 2; i >= 0; i--) {  
 if (nums[i] < nums[i + 1]) {  
 k = i;  
 break;  
 }  
 }  
 if (k == -1) break;  
  
 // step3: find the first nums[l] > nums[k] from the end to start  
 int l = nums.length - 1;  
 while (nums[l] <= nums[k]) {  
 l--;  
 }  
  
 // step3: swap between l and k  
 int temp = nums[l];  
 nums[l] = nums[k];  
 nums[k] = temp;  
  
 // step4: reverse between k + 1, nums.length - 1  
 reverse(nums, k + 1, nums.length - 1);  
 }  
  
 return result;  
 }  
  
 private void reverse(int[] nums, int lb, int ub) {  
 while (lb < ub) {  
 int temp = nums[lb];  
 nums[lb] = nums[ub];  
 nums[ub] = temp;  
 lb++;  
 ub--;  
 }  
 }  
}

### 源码分析

注意好步骤即可，其中对于数组的 reverse 操作不可在 while 循环中自增，极易出 bug! 对于 Java 来说其实可以首先将数组转化为 List, 相应的方法多一些。

### 复杂度分析

除了将 n!n!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(nlogn)O(n \log n)O(nlogn), 反转操作近似为 O(n)O(n)O(n), 故总的时间复杂度为 O(n!)O(n!)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](http://www.nayuki.io/page/next-lexicographical-permutation-algorithm) - 此题非递归方法更为详细的解释。
* [Permutation - Wikipedia, the free encyclopedia](https://en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order) - 字典序实现。
* [Programming Interview Questions 11: All Permutations of String | Arden DertatArden Dertat](http://www.ardendertat.com/2011/10/28/programming-interview-questions-11-all-permutations-of-string/)
* [algorithm - complexity of recursive string permutation function - Stack Overflow](http://stackoverflow.com/questions/5363619/complexity-of-recursive-string-permutation-function)
* [[leetcode]Permutations @ Python - 南郭子綦 - 博客园](http://www.cnblogs.com/zuoyuan/p/3758816.html)
* [[leetcode] permutations的讨论 - tuantuanls的专栏 - 博客频道 - CSDN.NET](http://blog.csdn.net/tuantuanls/article/details/8717262)
* [非递归排列算法（Permutation Generation）](http://arieshout.me/2012/04/non-recursive-permutation-generation.html)
* [闲谈permutations | HelloYou](http://helloyou2012.me/?p=133)
* [9.7. itertools — Functions creating iterators for efficient looping — Python 2.7.10 documentation](https://docs.python.org/2/library/itertools.html#itertools.permutations)

## Unique Permutations

# Permutations II

### Source

* leetcode: [Permutations II | LeetCode OJ](https://leetcode.com/problems/permutations-ii/)
* lintcode: [(16) Permutations II](http://www.lintcode.com/en/problem/permutations-ii/)

### Problem

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1 - backtracking

在上题的基础上进行剪枝，剪枝的过程和 [Unique Subsets](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/unique_subsets.html) 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 [1,21,22][1, 2\_1, 2\_2][1,21,22] 为例。按照上题 Permutations 的解法，我们可以得到如下全排列。

1. [1,21,22][1, 2\_1, 2\_2][1,21,22]
2. [1,22,21][1, 2\_2, 2\_1][1,22,21]
3. [21,1,22][2\_1, 1, 2\_2][21,1,22]
4. [21,22,1][2\_1, 2\_2, 1][21,22,1]
5. [22,1,21][2\_2, 1, 2\_1][22,1,21]
6. [22,21,1][2\_2, 2\_1, 1][22,21,1]

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 222\_222 在前，而第一个 212\_121 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 [1,21,22][1, 2\_1, 2\_2][1,21,22] 和 [1,22,21][1, 2\_2, 2\_1][1,22,21] 进行说明。

首先可以确定 [1,21,22][1, 2\_1, 2\_2][1,21,22] 是我们要的一个解，此时 list 为 [1,21,22][1, 2\_1, 2\_2][1,21,22], 经过两次 list.pop\_back() 之后， list 为 [1][1][1], 如果不进行剪枝，那么接下来要加入 list 的将为 222\_222, 那么我们剪枝要做的就是避免将 222\_222 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 222\_222 时，相对应的 visited[1] 为 true (对应 212\_121)，而在第二次加入 222\_222 时，相对应的 visited[1] 为 false ，因为在 list 为 [1,21][1, 2\_1][1,21] 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。

### 题解2 - 字典序

Permutation 的题使用字典序的做法其实更为简单，且为迭代的解法，效率也更高。代码和之前的 Permutations 那道题一模一样。

### Java

public class Solution {  
 public List<List<Integer>> permuteUnique(int[] nums) {  
 List<List<Integer>> result = new ArrayList<List<Integer>>();  
 if (nums == null || nums.length == 0) {  
 return result;  
 }  
  
 Arrays.sort(nums);  
 while (true) {  
 // step1: add list to result  
 List<Integer> list = new ArrayList<Integer>();  
 for (int i : nums) {  
 list.add(i);  
 }  
 result.add(list);  
  
 // step2: find nums[k] < nums[k + 1] backward  
 int k = -1;  
 for (int i = nums.length - 2; i >= 0; i--) {  
 if (nums[i] < nums[i + 1]) {  
 k = i;  
 break;  
 }  
 }  
 if (k == -1) break;  
  
 // step3: swap with nums[l]  
 int l = nums.length - 1;  
 while (l > k && nums[l] <= nums[k]) {  
 l--;  
 }  
 int temp = nums[l];  
 nums[l] = nums[k];  
 nums[k] = temp;  
  
 // step4: reverse between k+1, nums.length - 1  
 reverse(nums, k + 1, nums.length - 1);  
 }  
  
 return result;  
 }  
  
 private void reverse(int[] nums, int lb, int ub) {  
 while (lb < ub) {  
 int temp = nums[lb];  
 nums[lb] = nums[ub];  
 nums[ub] = temp;  
 lb++;  
 ub--;  
 }  
 }  
}

### 源码分析

见前一题，略。

### 复杂度分析

略

### Reference

* [Permutation II | 九章算法](http://www.jiuzhang.com/solutions/permutations-ii/)

## Next Permutation

# Next Permutation

### Source

* lintcode: [(52) Next Permutation](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutations.html) 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

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)O(n)O(n), 使用了 temp 临时变量，空间复杂度可认为是 O(1)O(1)O(1).

### Reference

* [Permutations](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutations.html)

## Previous Permuation

# Previous Permuation

### Source

* lintcode: [(51) Previous Permuation](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/next_permutation.html) 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

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)O(n)O(n), 使用了 temp 临时变量，空间复杂度可认为是 O(1)O(1)O(1).

### Reference

* [Permutations](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutations.html)

## Unique Binary Search Trees II

## Permutation Index

# Permutation Index

### Source

* lintcode: [(197) Permutation Index](http://www.lintcode.com/en/problem/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!)O(n!)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

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(n2)O(n^2)O(n2). 使用了部分额外空间，空间复杂度 O(1)O(1)O(1).

### Reference

* [Permutation Index](http://www.geekviewpoint.com/java/numbers/permutation_index)

## Permutation Index II

# Permutation Index II

### Source

* lintcode: [(198) Permutation Index II](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutation_index.html) 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

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(n2)O(n^2)O(n2), 使用了哈希表，空间复杂度为 O(n)O(n)O(n).

## Permutation Sequence

# Permutation Sequence

### Source

* leetcode: [Permutation Sequence | LeetCode OJ](https://leetcode.com/problems/permutation-sequence/)
* lintcode: [(388) Permutation Sequence](http://www.lintcode.com/en/problem/permutation-sequence/)

### Problem

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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutation_index.html) 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

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) {  
 // get factorial array  
 int[] fact = new int[n];  
 fact[0] = 1;  
 for (int i = 1; i < n; i++) {  
 fact[i] = fact[i - 1] \* i;  
 }  
 // generate nums 1 to n  
 List<Integer> nums = new ArrayList<Integer>();  
 for (int i = 1; i <= n; i++) {  
 nums.add(i);  
 }  
 // get the permutation digit  
 StringBuilder sb = new StringBuilder();  
 for (int i = 0; i < n; i++) {  
 // k begins from 1, so (1, 2) is a group  
 int rank = (k - 1) / fact[n - i - 1];  
 k = (k - 1) % fact[n - i - 1] + 1;  
 // ajust the mapping of rank to num  
 sb.append(nums.get(rank));  
 nums.remove(nums.get(rank));  
 }  
  
 return sb.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)O(n)O(n), 用了与 n 等长的一些数组，空间复杂度为 O(n)O(n)O(n).

### Reference

* [Permutation Sequence 解题报告](http://blog.sina.com.cn/s/blog_eb52001d0102v1ss.html)
* [Permutation Sequence 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/permutation-sequence/)
* [c++ - How to transform a vector into a string? - Stack Overflow](http://stackoverflow.com/questions/2518979/how-to-transform-a-vectorint-into-a-string)

## Palindrome Partitioning

## Combinations

# Combinations

### Source

* leetcode: [Combinations | LeetCode OJ](https://leetcode.com/problems/combinations/)
* lintcode: [(152) Combinations](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutations.html) 模板。

### 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 Cn2C\_n^2Cn2, 每组解有两个元素，故时间复杂度应为 O(n2)O(n^2)O(n2). list 只保留最多两个元素，空间复杂度 O(1)O(1)O(1).

## Combination Sum

# Combination Sum

### Source

* leetcode: [Combination Sum | LeetCode OJ](https://leetcode.com/problems/combination-sum/)
* lintcode: [(135) Combination Sum](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/permutations.html) 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

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!)O(n!)O(n!), 使用了list 保存中间结果，空间复杂度 O(n)O(n)O(n).

### Reference

* Soulmachine 的 leetcode 题解

## Combination Sum II

# Combination Sum II

### Source

* leetcode: [Combination Sum II | LeetCode OJ](https://leetcode.com/problems/combination-sum-ii/)
* lintcode: [(153) Combination Sum II](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/unique_subsets.html) 非常类似。在 [Combination Sum](http://algorithm.yuanbin.me/zh-cn/exhaustive_search/combination_sum.html) 的基础上改改就好了。

### 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)O(n)O(n), 空间复杂度 O(n)O(n)O(n).

## Minimum Depth of Binary Tree

## Word Search

# Dynamic Programming

## Triangle

# Triangle - Find the minimum path sum from top to bottom

### Source

* lintcode: [(109) Triangle](http://www.lintcode.com/en/problem/triangle/)

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.  
  
Note  
Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.  
  
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(2n)O(2^n)O(2n), 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(2n)O(2^n)O(2n)，变为 O(n2)O(n^2)O(n2), 每个坐标的元素仅计算一次，故共计算的次数为 1+2+...+n≈O(n2)1+2+...+n \approx O(n^2)1+2+...+n≈O(n2).

### 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]triangle[x][y]triangle[x][y] 的元素出发，其路径只可能为 triangle[x][y]−>triangle[x+1][y]triangle[x][y]->triangle[x+1][y]triangle[x][y]−>triangle[x+1][y] 或者 triangle[x][y]−>triangle[x+1][y+1]triangle[x][y]->triangle[x+1][y+1]triangle[x][y]−>triangle[x+1][y+1]. 以点 (x,y)(x,y)(x,y) 作为参考，那么可能的状态 f(x,y)f(x,y)f(x,y) 就可以是：

1. 从 (x,y)(x,y)(x,y) 出发走到最后一行的最短路径和
2. 从 (0,0)(0,0)(0,0) 走到 (x,y)(x,y)(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：f1(x,y)=min{f1(x+1,y),f1(x+1,y+1)}+triangle[x][y]f\_1(x,y) = min{f\_1(x+1, y), f\_1(x+1, y+1)} + triangle[x][y]f1(x,y)=min{f1(x+1,y),f1(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：f2(x,y)=min{f2(x−1,y),f2(x−1,y−1)}+triangle[x][y]f\_2(x,y) = min{f\_2(x-1, y), f\_2(x-1, y-1)} + triangle[x][y]f2(x,y)=min{f2(x−1,y),f2(x−1,y−1)}+triangle[x][y]

两个状态所对应的初始状态分别为 f1(n−1,y),0≤y≤n−1f\_1(n-1, y), 0 \leq y \leq n-1f1(n−1,y),0≤y≤n−1 和 f2(0,0)f\_2(0,0)f2(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[i][j];  
 }  
 }  
  
 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

# Backpack

### Source

* lintcode: [(92) Backpack](http://www.lintcode.com/en/problem/backpack/)

### Problem

Given *n* items with size AiA\_iAi, an integer *m* denotes the size of a backpack.How full you can fill this backpack?

#### Example

If we have 4 items with size [2, 3, 5, 7] , the backpack size is 11, we canselect [2, 3, 5] , so that the max size we can fill this backpack is 10 . Ifthe backpack size is 12 . we can select [2, 3, 7] so that we can fulfillthe backpack.

You function should return the max size we can fill in the given backpack.

#### Note

You can not divide any item into small pieces.

#### Challenge

O(n x m) time and O(m) memory.

O(n x m) memory is also acceptable if you do not know how to optimize memory.

### 题解1

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。参考前文 [Knapsack](http://algorithm.yuanbin.me/zh-cn/basics_algorithm/knapsack.html) 中总结的解法，这个题中可以将背包的 size 理解为传统背包中的重量；题目问的是能达到的最大 size, 故可将每个背包的 size 类比为传统背包中的价值。

考虑到数组索引从0开始，故定义状态 bp[i + 1][j] 为前 i 个物品中选出重量不超过 j 时总价值的最大值。状态转移方程则为分 A[i] > j 与否两种情况考虑。初始化均为0，相当于没有放任何物品。

### Java

public class Solution {  
 /\*\*  
 \* @param m: An integer m denotes the size of a backpack  
 \* @param A: Given n items with size A[i]  
 \* @return: The maximum size  
 \*/  
 public int backPack(int m, int[] A) {  
 if (A == null || A.length == 0) return 0;  
  
 final int M = m;  
 final int N = A.length;  
 int[][] bp = new int[N + 1][M + 1];  
  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j <= M; j++) {  
 if (A[i] > j) {  
 bp[i + 1][j] = bp[i][j];  
 } else {  
 bp[i + 1][j] = Math.max(bp[i][j], bp[i][j - A[i]] + A[i]);  
 }  
 }  
 }  
  
 return bp[N][M];  
 }  
}

### 源码分析

注意索引及初始化的值，尤其是 N 和 M 的区别，内循环处可等于 M。

### 复杂度分析

两重 for 循环，时间复杂度为 O(m×n)O(m \times n)O(m×n), 二维矩阵的空间复杂度为 O(m×n)O(m \times n)O(m×n), 一维矩阵的空间复杂度为 O(m)O(m)O(m).

### 题解2

接下来看看 [九章算法](http://www.jiuzhang.com/solutions/backpack/) 的题解，**这种解法感觉不是很直观，推荐使用题解1的解法。**

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]f[i][S] = f[i-1][S-A[i]] ~or~ f[i-1][S]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。

1. f[i−1][S−A[i]]f[i-1][S-A[i]]f[i−1][S−A[i]]: **放入第i个物品**，前 i−1i-1i−1 个物品能否取出一些体积和为 S−A[i]S-A[i]S−A[i] 的背包。
2. f[i−1][S]f[i-1][S]f[i−1][S]: **不放入第i个物品**，前 i−1i-1i−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⋯n][0]=true; f[0][1⋯m]=falsef[1 \cdots n][0]=true; ~f[0][1 \cdots m]=falsef[1⋯n][0]=true; f[0][1⋯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]f[n][S]f[n][S] 值为true的最大S (1≤S≤m1 \leq S \leq m1≤S≤m)

### C++ - 2D vector

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

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;  
 }  
};

### 复杂度分析

两重 for 循环，时间复杂度均为 O(m×n)O(m \times n)O(m×n), 二维矩阵的空间复杂度为 O(m×n)O(m \times n)O(m×n), 一维矩阵的空间复杂度为 O(m)O(m)O(m).

### Reference

* 《挑战程序设计竞赛》第二章
* [Lintcode: Backpack - neverlandly - 博客园](http://www.cnblogs.com/EdwardLiu/p/4269149.html)
* [九章算法 | 背包问题](http://www.jiuzhang.com/problem/58/)
* [崔添翼 § 翼若垂天之云 › 《背包问题九讲》2.0 alpha1](http://cuitianyi.com/blog/%E3%80%8A%E8%83%8C%E5%8C%85%E9%97%AE%E9%A2%98%E4%B9%9D%E8%AE%B2%E3%80%8B2-0-alpha1/)

## Backpack II

# Backpack II

### Source

* lintcode: [(125) Backpack II](http://www.lintcode.com/en/problem/backpack-ii/)

### Problem

Given *n* items with size AiAiAi and value Vi, and a backpack with size *m*.What's the maximum value can you put into the backpack?

#### Example

Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4] , and abackpack with size 10 . The maximum value is 9 .

#### Note

You cannot divide item into small pieces and the total size of items youchoose should smaller or equal to m.

#### Challenge

O(n x m) memory is acceptable, can you do it in O(m) memory?

### 题解

首先定义状态 K(i,w)K(i,w)K(i,w) 为前 iii 个物品放入size为 www 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−wi)+vi}K(i,w) = \max {K(i-1, w), K(i-1, w - w\_i) + v\_i}K(i,w)=max{K(i−1,w),K(i−1,w−wi)+vi}

详细分析过程见 [Knapsack](http://algorithm.yuanbin.me/zh-cn/basics_algorithm/knapsack.html)

### 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];  
 }  
};

### Java

public class Solution {  
 /\*\*  
 \* @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  
 \*/  
 public int backPackII(int m, int[] A, int V[]) {  
 if (A == null || V == null || A.length == 0 || V.length == 0) return 0;  
  
 final int N = A.length;  
 final int M = m;  
 int[][] bp = new int[N + 1][M + 1];  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j <= M; j++) {  
 if (A[i] > j) {  
 bp[i + 1][j] = bp[i][j];  
 } else {  
 bp[i + 1][j] = Math.max(bp[i][j], bp[i][j - A[i]] + V[i]);  
 }  
 }  
 }  
  
 return bp[N][M];  
 }  
}

### 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 II - neverlandly - 博客园](http://www.cnblogs.com/EdwardLiu/p/4272300.html)
* [九章算法 | 背包问题](http://www.jiuzhang.com/problem/58/)

## Minimum Path Sum

## Unique Paths

## Unique Paths II

## Climbing Stairs

# Climbing Stairs

### Source

* lintcode: [(111) Climbing Stairs](http://www.lintcode.com/en/problem/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++

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)O(n)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](http://www.jiuzhang.com/solutions/climbing-stairs/)

### 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

## Word Break

## Longest Increasing Subsequence

## Palindrome Partitioning II

## Longest Common Subsequence

## Edit Distance

## Jump Game II

## Best Time to Buy and Sell Stock

# Best Time to Buy and Sell Stock

### Source

* leetcode: [Best Time to Buy and Sell Stock | LeetCode OJ](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/)
* lintcode: [(149) Best Time to Buy and Sell Stock](http://www.lintcode.com/en/problem/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

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++

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)O(n)O(n), 使用了几个额外变量，空间复杂度为 O(1)O(1)O(1).

### Reference

* soulmachine 的卖股票系列

## Best Time to Buy and Sell Stock II

# Best Time to Buy and Sell Stock II

### Source

* leetcode: [Best Time to Buy and Sell Stock II | LeetCode OJ](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/)
* lintcode: [(150) Best Time to Buy and Sell Stock II](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 用了几个额外变量，空间复杂度为 O(1)O(1)O(1).

### Reference

* soulmachine 的卖股票系列

## Best Time to Buy and Sell Stock III

# Best Time to Buy and Sell Stock III

### Source

* leetcode: [Best Time to Buy and Sell Stock III | LeetCode OJ](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/)
* lintcode: [(151) Best Time to Buy and Sell Stock III](http://www.lintcode.com/en/problem/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(n2)O(n^2)O(n2) 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

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

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)O(n)O(n), 利用了若干和数组等长的数组，空间复杂度也为 O(n)O(n)O(n).

### Reference

* soulmachine 的卖股票系列

## Best Time to Buy and Sell Stock IV

## Distinct Subsequences

## Interleaving String

# Interleaving String

### Source

* leetcode: [Interleaving String | LeetCode OJ](https://leetcode.com/problems/interleaving-string/)
* lintcode: [(29) Interleaving String](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 空间复杂度 O(1)O(1)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 + i3));  
 // s1[i1:], s2[1+i2:], s3[1+i3:]  
 bool case2 = isInterleave(s1.substr(i1), s2.substr(1 + i2), s3.substr(1 + i3));  
 // 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.substring(1 + i3));  
 // s1[i1:], s2[1+i2:], s3[1+i3:]  
 boolean case2 = isInterleave(s1.substring(i1), s2.substring(1 + i2), s3.substring(1 + i3));  
 // 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 {  
 /\*\*  
 \* 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][i2];  
 boolean case2 = s2.charAt(i2 - 1) == s3.charAt(i1 + i2 - 1) && f[i1][i2 - 1];  
 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(n2)O(n^2)O(n2), 使用了二维矩阵，空间复杂度 O(n2)O(n^2)O(n2). 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](http://www.jiuzhang.com/solutions/interleaving-string/)

## Maximum Subarray

# Maximum Subarray

### Source

* leetcode: [Maximum Subarray | LeetCode OJ](https://leetcode.com/problems/maximum-subarray/)
* lintcode: [(41) Maximum Subarray](http://www.lintcode.com/en/problem/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 n2n^2n2 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

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)O(n)O(n), 使用了几个额外变量，空间复杂度 O(1)O(1)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)O(n)O(n), 使用了类似滚动数组的处理方式，空间复杂度 O(1)O(1)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)O(n)O(n), 空间复杂度也为 O(n)O(n)O(n).

### Reference

* 《剑指 Offer》第五章
* [Maximum Subarray 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/maximum-subarray/)

## Maximum Subarray II

# Maximum Subarray II

### Source

* lintcode: [(42) Maximum Subarray II](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/dynamic_programming/maximum_subarray.html) 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

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)O(2n)=O(n)O(2n)=O(n), 空间复杂度 O(n)O(n)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)O(n)O(n). 故总的时间复杂度为 O(n)O(n)O(n), 空间复杂度 O(n)O(n)O(n).

## Longest Increasing Continuous subsequence

## Longest Increasing Continuous subsequence II

# Graph

# Graph

本章主要总结图与搜索相关题目。

## Find the Connected Component in the Undirected Graph

## Route Between Two Nodes in Graph

## Topological Sorting

## Word Ladder

## Bipartial Graph Part I

# Data Structure

# 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

# Implement Queue by Two Stacks

### Source

* lintcode: [(40) Implement Queue by Two Stacks](http://www.lintcode.com/en/problem/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

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)O(1)O(1).

### Reference

* [Implement Queue by Two Stacks 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/implement-queue-by-two-stacks/)

## Min Stack

# Min Stack

### Source

* lintcode: [(12) Min Stack](http://www.lintcode.com/en/problem/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)O(1)O(1) 的时间内找出最小值，一般这种 O(1)O(1)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

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)O(1)O(1).

## Sliding Window Maximum

# Sliding Window Maximum

### Source

* leetcode: [Sliding Window Maximum | LeetCode OJ](https://leetcode.com/problems/sliding-window-maximum/)
* lintcode: [(362) Sliding Window Maximum](http://www.lintcode.com/en/problem/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)O(nk)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)O(1)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)O(1)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](https://zh.wikipedia.org/wiki/%E5%8F%8C%E7%AB%AF%E9%98%9F%E5%88%97)。可以自己试着以一个实际例子来帮助理解。

### Java

public class Solution {  
 /\*\*  
 \* @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)O(n)O(n), 空间复杂度 O(k)O(k)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)O(k)O(k).

### Reference

* 《剑指 Offer》
* [sliding-window-maximum 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/sliding-window-maximum/)
* [Maximum of all subarrays of size k (Added a O(n) method) - GeeksforGeeks](http://www.geeksforgeeks.org/maximum-of-all-subarrays-of-size-k/)

## Longest Words

# Longest Words

### Source

* lintcode: [(133) Longest Words](http://www.lintcode.com/en/problem/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

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)O(n)O(n), 最坏情况下需要保存 n - 1个字符串，空间复杂度 O(n)O(n)O(n).

### Reference

* [Lintcode: Longest Words | codesolutiony](https://codesolutiony.wordpress.com/2015/06/07/lintcode-longest-words/)

## Heapify

# Heapify

### Source

* lintcode: [(130) Heapify](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/basics_sorting/heap_sort.html) 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++

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)(N)(N), 证明待补充...

### Reference

* [Heap Sort](http://algorithm.yuanbin.me/zh-cn/basics_sorting/heap_sort.html)
* [Heapify 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/heapify/)

# Problem Misc

# 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

# Nuts and Bolts Problem

### Source

* lintcode: [(399) Nuts & Bolts Problem](http://www.lintcode.com/en/problem/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(n2)O(n^2)O(n2), 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(nlogn)O(n \log n)O(nlogn), 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(nlogn)O(n \log n)O(nlogn) 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)O(n)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(2nlogn)O(2n \log n)O(2nlogn), 使用了一些临时变量，空间复杂度 O(1)O(1)O(1).

### Reference

* [LintCode/Nuts & Bolts Problem.py at master · algorhythms/LintCode](https://github.com/algorhythms/LintCode/blob/master/Nuts%20%26%20Bolts%20Problem.py)

## String to Integer

# String to Integer

### Source

* leetcode: [String to Integer (atoi) | LeetCode OJ](https://leetcode.com/problems/string-to-integer-atoi/)
* lintcode: [(54) String to Integer(atoi)](http://www.lintcode.com/en/problem/string-to-integeratoi/)

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

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](http://www.jiuzhang.com/solutions/string-to-integer-atoi/)

## Insert Interval

# Insert Interval

### Source

* leetcode: [Insert Interval | LeetCode OJ](https://leetcode.com/problems/insert-interval/)
* lintcode: [(30) Insert Interval](http://www.lintcode.com/en/problem/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;  
 \* }  
 \*/  
  
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)O(n)O(n). 不考虑作为结果返回占用的空间 result, 空间复杂度 O(1)O(1)O(1).

### Reference

* [Insert Interval 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/insert-interval/)

## Merge Intervals

# Merge Intervals

### Source

* leetcode: [Merge Intervals | LeetCode OJ](https://leetcode.com/problems/merge-intervals/)
* lintcode: [(156) Merge Intervals](http://www.lintcode.com/en/problem/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(nlogn)O(n \log n)O(nlogn), 遍历 O(n)O(n)O(n), 所以总的时间复杂度为 O(nlogn)O(n \log n)O(nlogn). 空间复杂度 O(1)O(1)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](http://algorithm.yuanbin.me/zh-cn/problem_misc/insert_interval.html)  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(n2)O(1 + 2 + ... + n) = O(n^2)O(1+2+...+n)=O(n2). 空间复杂度为 O(n)O(n)O(n).

### Reference

* [Merge Intervals 参考程序 Java/C++/Python](http://www.jiuzhang.com/solutions/merge-intervals/)
* Soulmachine 的 leetcode 题解

## Minimum Subarray

# Minimum Subarray

### Source

* lintcode: [(44) Minimum Subarray](http://www.lintcode.com/en/problem/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](http://algorithm.yuanbin.me/zh-cn/dynamic_programming/maximum_subarray.html) 的变形，使用区间和容易理解和实现。

### 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

# Matrix Zigzag Traversal

### Source

* lintcode: [(185) Matrix Zigzag Traversal](http://www.lintcode.com/en/problem/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

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)1 + 2 + ... + (m + n) = O((m+n)^2)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⋅n)O(m \cdot n)O(m⋅n). 空间复杂度都是 O(1)O(1)O(1).

### Reference

* [LintCode/matrix-zigzag-traversal.cpp at master · kamyu104/LintCode](https://github.com/kamyu104/LintCode/blob/master/C++/matrix-zigzag-traversal.cpp)

## Valid Sudoku

# Valid Sudoku

### Source

* leetcode: [Valid Sudoku | LeetCode OJ](https://leetcode.com/problems/valid-sudoku/)
* lintcode: [(389) Valid Sudoku](http://www.lintcode.com/en/problem/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.



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

# Add Binary

### Source

* leetcode: [Add Binary | LeetCode OJ](https://leetcode.com/problems/add-binary/)
* lintcode: [(408) Add Binary](http://www.lintcode.com/en/problem/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

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)O(n)O(n) 变为 O(n2)O(n^2)O(n2) 了。

### 复杂度分析

遍历两个字符串，时间复杂度 O(n)O(n)O(n). reverse 操作时间复杂度 O(n)O(n)O(n), 故总的时间复杂度 O(n)O(n)O(n). 使用了 StringBuilder 作为临时存储对象，空间复杂度 O(n)O(n)O(n).

## Reverse Integer

# Reverse Integer

### Source

* leetcode: [Reverse Integer | LeetCode OJ](https://leetcode.com/problems/reverse-integer/)
* lintcode: [(413) Reverse Integer](http://www.lintcode.com/en/problem/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](https://github.com/FreeTymeKiyan/LeetCode-Sol-Res/blob/master/Easy/ReverseInt.java)

## Gray Code

# Gray Code

### Source

* leetcode: [Gray Code | LeetCode OJ](https://leetcode.com/problems/gray-code/)
* lintcode: [(411) Gray Code](http://www.lintcode.com/en/problem/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 2n2^n2n 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.

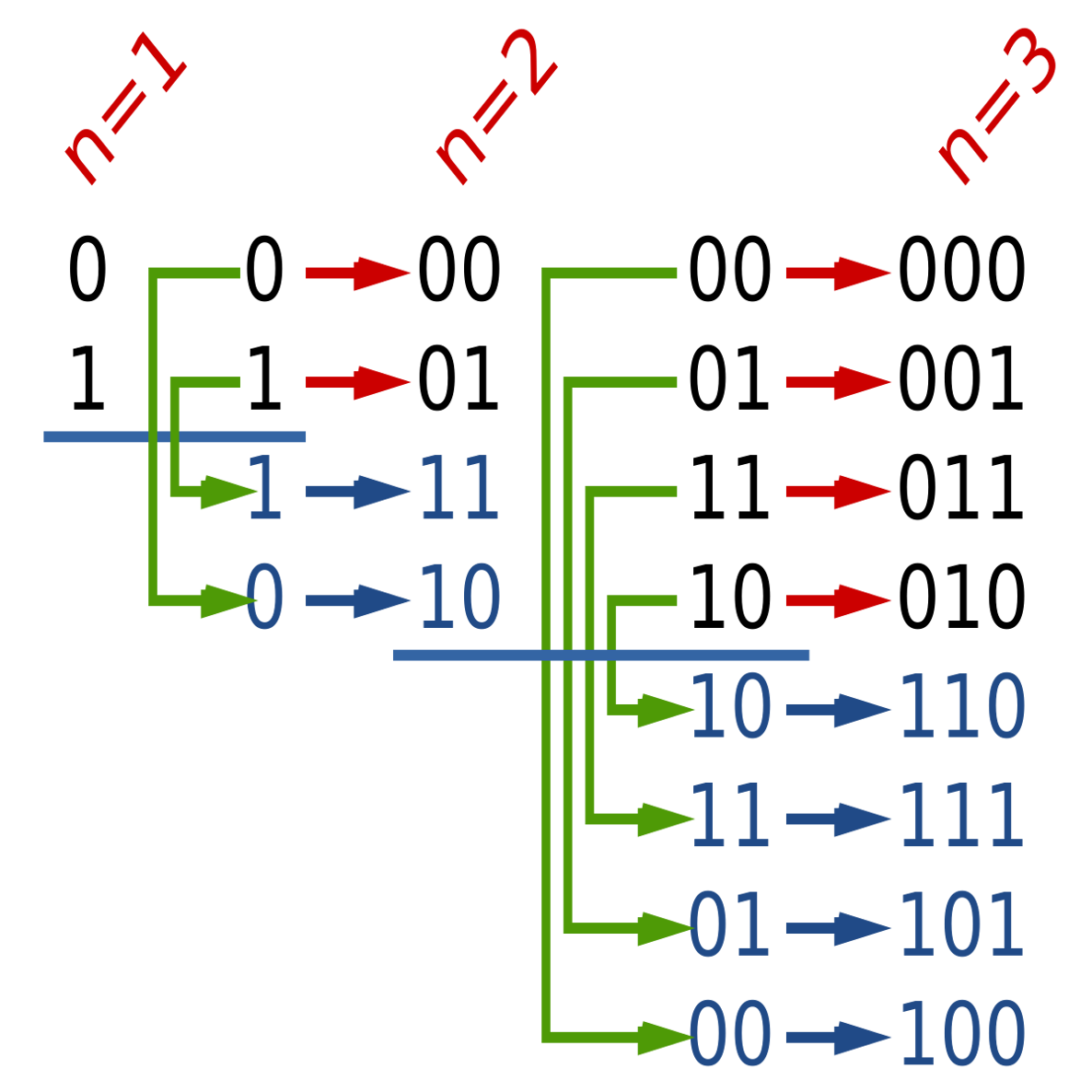
[0,2,3,1] is also a valid gray code sequence according to the above definition.

#### Challenge

O(2n)O(2^n)O(2n) 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](https://zh.wikipedia.org/wiki/%E6%A0%BC%E9%9B%B7%E7%A0%81) 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 2n2^n2n 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(2n)O(2^n)O(2n), 使用了 msb 代表最高位的值便于后续相加，空间复杂度 O(1)O(1)O(1).

### Reference

* Soulmachine 的 leetcode 题解

## Find the Missing Number

# Find the Missing Number

### Source

* lintcode: [(196) Find the Missing Number](http://www.lintcode.com/en/problem/find-the-missing-number/)
* [Find the Missing Number - GeeksforGeeks](http://www.geeksforgeeks.org/find-the-missing-number/)

### Problem

Given an array contains *N* numbers of 0 .. *N*, find which number doesn't exist in the array.

#### Example

Given *N* = 3 and the array [0, 1, 3] , return 2 .

#### Challenge

Do it in-place with O(1)O(1)O(1) extra memory and O(n)O(n)O(n) time.

### 题解1 - 位运算

和找单数的题类似，这里我们不妨试试位运算中异或的思路。最开始自己想到的是利用相邻项异或结果看是否会有惊喜，然而发现 a^(a+1) != a^a + a^1 之后眼泪掉下来... 如果按照找单数的做法，首先对数组所有元素异或，得到数 x1 , 现在的问题是如何利用 x1 得到缺失的数，由于找单数中其他数都是成对出现的，故最后的结果即是单数，这里每个数都是单数，怎么办呢？我们现在再来分析下如果没有缺失数的话会是怎样呢？假设所有元素异或得到数 x2 , 数 x1 和 x2 有什么差异呢？假设缺失的数是 x0 ，那么容易知道 x2 = x1 ^ x0 , 相当于现在已知 x1 和 x2 ，要求 x0 . 根据 [Bit Manipulation](http://algorithm.yuanbin.me/zh-cn/basics_misc/bit_manipulation.html) 中总结的交换律， x0 = x1 ^ x2 .

位运算的题往往比较灵活，需要好好利用常用等式变换。

### Java

public class Solution {  
 /\*\*  
 \* @param nums: an array of integers  
 \* @return: an integer  
 \*/  
 public int findMissing(int[] nums) {  
 if (nums == null || nums.length == 0) return -1;  
  
 // get xor from 0 to N excluding missing number  
 int x1 = 0;  
 for (int i : nums) {  
 x1 ^= i;  
 }  
  
 // get xor from 0 to N  
 int x2 = 0;  
 for (int i = 0; i <= nums.length; i++) {  
 x2 ^= i;  
 }  
  
 // missing = x1 ^ x2;  
 return x1 ^ x2;  
 }  
}

### 源码分析

略

### 复杂度分析

遍历原数组和 N+1大小的数组，时间复杂度 O(n)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

### 题解2 - 桶排序

非常简单直观的想法——排序后检查缺失元素，但是此题中要求时间复杂度为 O(n)O(n)O(n), 因此如果一定要用排序来做，那一定是使用非比较排序如桶排序或者计数排序。题中另一提示则是要求只使用 O(1)O(1)O(1) 的额外空间，那么这就是在提示我们应该使用原地交换。根据题意，元素应无重复，可考虑使用桶排，索引和值一一对应即可。第一重 for 循环遍历原数组，内循环使用 while, 调整索引处对应的值，直至相等或者索引越界为止，for 循环结束时桶排结束。最后再遍历一次数组找出缺失元素。

初次接触这种题还是比较难想到使用桶排这种思想的，尤其是利用索引和值一一对应这一特性找出缺失元素，另外此题在实际实现时不容易做到 bug-free, while 循环处容易出现死循环。

### Java

public class Solution {  
 /\*\*  
 \* @param nums: an array of integers  
 \* @return: an integer  
 \*/  
 public int findMissing(int[] nums) {  
 if (nums == null || nums.length == 0) return -1;  
  
 bucketSort(nums);  
 // find missing number  
 for (int i = 0; i < nums.length; i++) {  
 if (nums[i] != i) {  
 return i;  
 }  
 }  
  
 return nums.length;  
 }  
  
 private void bucketSort(int[] nums) {  
 for (int i = 0; i < nums.length; i++) {  
 while (nums[i] != i) {  
 // ignore nums[i] == nums.length  
 if (nums[i] == nums.length) {  
 break;  
 }  
 int nextNum = nums[nums[i]];  
 nums[nums[i]] = nums[i];  
 nums[i] = nextNum;  
 }  
 }  
 }  
}

### 源码分析

难点一在于正确实现桶排，难点二在于数组元素中最大值 N 如何处理。N 有三种可能：

1. N 不在原数组中，故最后应该返回 N
2. N 在原数组中，但不在数组中的最后一个元素
3. N 在原数组中且在数组最后一个元素

其中情况1在遍历桶排后的数组时无返回，最后返回 N.

其中2和3在 while 循环处均会遇到 break 跳出，即当前这个索引所对应的值要么最后还是 N，要么就是和索引相同的值。如果最后还是 N, 也就意味着原数组中缺失的是其他值，如果最后被覆盖掉，那么桶排后的数组不会出现 N, 且缺失的一定是 N 之前的数。

综上，这里的实现无论 N 出现在哪个索引都能正确返回缺失值。实现上还是比较巧妙的，所以说在没做过这类题时要在短时间内 bug-free 比较难，当然也可能是我比较菜...

另外一个难点在于如何保证或者证明 while 一定不会出现死循环，可以这么理解，如果 while 条件不成立且未出现 nums.length 这个元素，那么就一定会使得一个元素正确入桶，又因为没有重复元素出现，故一定不会出现死循环。

### 复杂度分析

桶排时间复杂度 O(n)O(n)O(n), 空间复杂度 O(1)O(1)O(1). 遍历原数组找缺失数时间复杂度 O(n)O(n)O(n). 故总的时间复杂度为 O(n)O(n)O(n), 空间复杂度 O(1)O(1)O(1).

## Minimum Window Substring

# Minimum Window Substring

### Source

* leetcode: [Minimum Window Substring | LeetCode OJ](https://leetcode.com/problems/minimum-window-substring/)
* lintcode: [(32) Minimum Window Substring](http://www.lintcode.com/en/problem/minimum-window-substring/)

### Problem

Given a string source and a string target, find the minimum window in sourcewhich will contain all the characters in target.

#### Example

source = "**ADOBECODEBANC**" target = "**ABC**" Minimum window is "**BANC**".

#### Note

If there is no such window in source that covers all characters in target,return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will alwaysbe only one unique minimum window in source.

#### Challenge

Can you do it in time complexity O(n) ?

#### Clarification

Should the characters in minimum window has the same order in target?

* Not necessary.

### 题解

计算目标字符串的字符在给定字符串中出现的最小窗口。由于并不需要在给定字符串中有序出现，故只需要统计出现次数。这是典型的需要借助『哈希表』实现的题。题中字符串中的字符可以假定为 ascii 码，那么我们使用256个 ascii 码处理起来较为方便。那么接下来有两个难点，一就是在于怎么知道给定字符串中某一窗口长度已包含目标字符串中的全部字符（可能重复），二是在包含目标字符串中全部字符后如果再出现目标字符串中的其他字符串时如何处理？

其中第一个难点我们通过巧用目标字符串的长度来处理，遍历给定字符串，如果给定字符串中出现的字符次数小于目标字符串，我们就更新总的字符出现次数。第二个难题通过维护窗口起止索引（两根指针）来处理，在给定字符串中出现目标字符串中的全部字符时向前移动窗口起始处，若窗口长度小于之前的窗口长度则更新最终答案要求的窗口起始索引。

### Java

public class Solution {  
 /\*\*  
 \* @param source: A string  
 \* @param target: A string  
 \* @return: A string denote the minimum window  
 \* Return "" if there is no such a string  
 \*/  
 public String minWindow(String source, String target) {  
 if (source == null || target == null) return "";  
 if (source.length() < target.length()) return "";  
  
 final int ASCII\_COUNT = 256;  
 int[] targetCount = new int[ASCII\_COUNT];  
 int[] sourceCount = new int[ASCII\_COUNT];  
 for (int i = 0; i < target.length(); i++) {  
 int ch2i = (int)target.charAt(i);  
 targetCount[ch2i]++;  
 }  
 // target string character appeared in source string  
 int winStart = 0, winMinStart = 0, winMin = Integer.MAX\_VALUE;  
 int occurence = 0;  
 for (int winEnd = 0; winEnd < source.length(); winEnd++) {  
 // convert character to integer  
 int ch2i = (int)source.charAt(winEnd);  
 sourceCount[ch2i]++;  
 // character occur in both source and target  
 if (targetCount[ch2i] > 0 && targetCount[ch2i] >= sourceCount[ch2i]) {  
 occurence++;  
 }  
 // adjust window size if all the target char occur in source  
 if (occurence == target.length()) {  
 // convert character to integer  
 int ch2i2 = (int)source.charAt(winStart);  
 while (sourceCount[ch2i2] > targetCount[ch2i2]) {  
 sourceCount[ch2i2]--;  
 winStart++;  
 ch2i2 = (int)source.charAt(winStart);  
 }  
 // update winMinStart  
 if (winMin > winEnd - winStart + 1) {  
 winMin = winEnd - winStart + 1;  
 winMinStart = winStart;  
 }  
 }  
 }  
  
 if (winMin == Integer.MAX\_VALUE) {  
 return "";  
 } else {  
 return source.substring(winMinStart, winMinStart + winMin);  
 }  
 }  
}

### 源码分析

整个程序最为核心的即为题解中所提出的两大难点，窗口移动的方法使用贪心实现，在窗口长度变小时需要记录起始索引。

### 复杂度分析

遍历给定字符串一次，外加更新窗口时可能需要遍历给定字符串一次，时间复杂度为 O(n)O(n)O(n), 使用了几个额外变量，空间复杂度 O(1)O(1)O(1).

### Reference

* [水中的鱼: [LeetCode] Minimum Window Substring 解题报告](http://fisherlei.blogspot.com/2012/12/leetcode-minimum-window-substring.html)

## Continuous Subarray Sum

# Continuous Subarray Sum

### Source

* lintcode: [(402) Continuous Subarray Sum](http://www.lintcode.com/en/problem/continuous-subarray-sum/)

### Problem

Given an integer array, find a continuous subarray where the sum of numbers isthe biggest. Your code should return the index of the first number and theindex of the last number. (If their are duplicate answer, return anyone)

#### Example

Give [-3, 1, 3, -3, 4] , return [1,4] .

### 题解

和题 [Maximum Subarray](http://algorithm.yuanbin.me/zh-cn/dynamic_programming/maximum_subarray.html) 几乎一模一样，只是返回值要求不一样。由于需要返回区间索引值，那么显然需要额外变量记录区间起止处。若使用题解2中提到的 sum - minSum 的区间和更新方式，索引终止处容易得知是 sum - minSum > maxSub 时的 i , 问题是索引起始处如何确定。容易得知索引起始处如果更新，必然在 minSum > sum 时，但问题在于满足此条件的可能不止一处，所以我们需要先记录这个索引值并在 sum - minSum > maxSub 时判定索引终止值是否大于索引起始值，不小于则更新。

此题难以一次 bug-free, 需要小心更新索引值。

### Java

public class Solution {  
 /\*\*  
 \* @param A an integer array  
 \* @return A list of integers includes the index of the first number and the index of the last number  
 \*/  
 public ArrayList<Integer> continuousSubarraySum(int[] A) {  
 ArrayList<Integer> result = new ArrayList<Integer>();  
 if (A == null || A.length == 0) return result;  
  
 int sum = 0, minSum = 0, maxSub = Integer.MIN\_VALUE;  
 int first = 0, last = 0;  
 int first2 = 0; // candidate for first  
 for (int i = 0; i < A.length; i++) {  
 if (minSum > sum) {  
 minSum = sum;  
 first2 = i;  
 }  
 sum += A[i];  
 if (sum - minSum > maxSub) {  
 maxSub = sum - minSum;  
 last = i;  
 // update first if valid  
 if (first2 <= last) first = first2;  
 }  
 }  
  
 result.add(first);  
 result.add(last);  
 return result;  
 }  
}

### 源码分析

除了最后要求的 first 和 last , 我们还需要引入 first2 作为 first 可能的候选变量值。

### 复杂度分析

略

## Continuous Subarray Sum II

# Continuous Subarray Sum II

### Source

* lintcode: [(403) Continuous Subarray Sum II](http://www.lintcode.com/en/problem/continuous-subarray-sum-ii/)
* [Maximum circular subarray sum - GeeksforGeeks](http://www.geeksforgeeks.org/maximum-contiguous-circular-sum/)

### Problem

Given an integer array, find a continuous rotate subarray where the sum ofnumbers is the biggest. Your code should return the index of the first numberand the index of the last number. (If their are duplicate answer, returnanyone. The answer can be rorate array or non- rorate array)

#### Example

Give [3, 1, -100, -3, 4] , return [4,1] .

### 题解

题 [Continuous Subarray Sum](http://algorithm.yuanbin.me/zh-cn/problem_misc/continuous_subarray_sum.html) 的 follow up, 这道题 AC 率极低，真是磨人的小妖精。在上题的基础上容易想到可以将 first 和 last 分四种情况讨论，然后再逆向求大于0的最大和即可，但是这种想法忽略了一种情况——旋转后的最大值可能由两段子数组和构成，而这种情况如果用上题的解法则会被忽略。

所以这道题的正确解法不是分 first 和 last 四种情况讨论，而是利用旋转数组的特性。第一种情况，无论怎么拼接原数组中的数组和都无法大于最大的单一数组和；第二种情况则相反。所以现在问题的关键则转化为怎么求第二种情况。首先可以明确一点，最终得到的数组和索引必须连续（含首尾相接）。也就是说情况二一旦出现，则我们可以将原数组中挖空一小段，现在问题来了：到底要挖掉多少元素？

**我们的目标是使得挖掉后的元素值最大。**由于分段求解不容易（被隔开），但是被挖掉的元素索引是挨着的！正难则反！由于数组的总和是一定的，那么我们只要求得被挖掉部分元素的最小值即可得两边子数组的最大值！最后判断两个最大值孰大孰小就可以了。

### Java

public class Solution {  
 /\*\*  
 \* @param A an integer array  
 \* @return A list of integers includes the index of the first number and the index of the last number  
 \*/  
 public ArrayList<Integer> continuousSubarraySumII(int[] A) {  
 ArrayList<Integer> result = new ArrayList<Integer>();  
 if (A == null || A.length == 0) return result;  
 // maximal subarray sum  
 ArrayList<Integer> sub1 = subSum(A, 1);  
 // minimal subarray sum  
 ArrayList<Integer> sub2 = subSum(A, -1);  
 int first = 0, last = 0;  
 if (sub1.get(3) - sub2.get(2) > sub1.get(2)) {  
 last = sub2.get(0) - 1;  
 first = sub2.get(1) + 1;  
 } else {  
 first = sub1.get(0);  
 last = sub1.get(1);  
 }  
 // corner case(all elements are negtive)  
 if (last == -1 && first == A.length) {  
 first = sub1.get(0);  
 last = sub1.get(1);  
 }  
  
 result.add(first);  
 result.add(last);  
 return result;  
 }  
  
 private ArrayList<Integer> subSum(int[] A, int sign) {  
 ArrayList<Integer> result = new ArrayList<Integer>();  
 // find the max/min subarray sum from [0...A.length]  
 int sum = 0, minSum = 0, maxSub = Integer.MIN\_VALUE;  
 if (sign == -1) maxSub = Integer.MAX\_VALUE;  
 int first = 0, last = 0;  
 int first2 = 0; // candidate for first  
 for (int i = 0; i < A.length; i++) {  
 if (sign \* minSum > sign \* sum) {  
 minSum = sum;  
 first2 = i;  
 }  
 sum += A[i];  
 if (sign \* (sum - minSum) > sign \* maxSub) {  
 maxSub = sum - minSum;  
 last = i;  
 // update first if valid  
 if (first2 <= last) first = first2;  
 }  
 }  
 result.add(first);  
 result.add(last);  
 result.add(maxSub);  
 result.add(sum);  
 return result;  
 }  
}

### 源码分析

由于既需要求最大子数组和，也需要求最小子数组和，我们将这一部分写成一私有方法，并加入 sign 控制符号。如果两段子数组和大于一段子数组和时，新的 first 和 last 正好相反。且在数组全为负时需要排除，直接使用单一子数组和最大的情况。

### 复杂度分析

遍历两次数组，时间复杂度 O(n)O(n)O(n), 使用了部分额外 List, 空间复杂度 O(1)O(1)O(1).

### Reference

* [CC150+Leetcode Continuous Subarray Sum II](http://meetqun.com/thread-9856-1-1.html)

## Longest Consecutive Sequence

# Longest Consecutive Sequence

### Source

* leetcode: [Longest Consecutive Sequence | LeetCode OJ](https://leetcode.com/problems/longest-consecutive-sequence/)
* lintcode: [(124) Longest Consecutive Sequence](http://www.lintcode.com/en/problem/longest-consecutive-sequence/)

### Problem

Given an unsorted array of integers, find the length of the longestconsecutive elements sequence.

#### Example

Given [100, 4, 200, 1, 3, 2] ,The longest consecutive elements sequence is [1, 2, 3, 4] . Return itslength: 4 .

#### Clarification

Your algorithm should run in O(*n*) complexity.

### 题解

首先看题要求，时间复杂度为 O(n)O(n)O(n), 如果排序，基于比较的实现为 nlognn \log nnlogn, 基数排序需要数据有特征。故排序无法达到复杂度要求。接下来可以联想空间换时间的做法，其中以哈希表为代表。这个题要求返回最长连续序列，不要求有序，非常符合哈希表的用法。**由于给定一个数其连续的数要么比它小1，要么大1，那么我们只需往左往右搜索知道在数组中找不到数为止。**结合哈希表查找为 O(1)O(1)O(1) 的特性即可满足要求。

### Java

public class Solution {  
 /\*\*  
 \* @param nums: A list of integers  
 \* @return an integer  
 \*/  
 public int longestConsecutive(int[] num) {  
 if (num == null || num.length == 0) return 0;  
  
 // add number to hashset  
 Set<Integer> hashset = new HashSet<Integer>();  
 for (int n : num) {  
 hashset.add(n);  
 }  
  
 int lcs = 0;  
 for (int n : num) {  
 int i = n, count = 1;  
 hashset.remove(n);  
 // i--  
 while (hashset.contains(--i)) {  
 count++;  
 hashset.remove(i);  
 }  
 // i++  
 i = n;  
 while (hashset.contains(++i)) {  
 count++;  
 hashset.remove(i);  
 }  
 // update lcs  
 lcs = Math.max(lcs, count);  
 }  
  
 return lcs;  
 }  
}

### 源码分析

首先使用 HashSet 建哈希表，然后遍历数组，依次往左往右搜相邻数，搜到了就从 Set 中删除。末尾更新最大值。

### 复杂度分析

时间复杂度和空间复杂度均为 O(n)O(n)O(n).

# Part III - Contest

# Part III - Contest

本节主要总结一些如 Google APAC, Microsoft 校招等在线测试的题。

# Google APAC

# Google APAC

本章总结 Google APAC 的一些题。

## APAC 2015 Round B

# APAC 2015 Round B

* [Dashboard - Round B APAC Test - Google Code Jam](https://code.google.com/codejam/contest/4214486/dashboard)

### Problem A. Password Attacker

# Problem A. Password Attacker

### Source

* [Dashboard - Round B APAC Test - Problem A. Password Attacker](https://code.google.com/codejam/contest/4214486/dashboard#s=p0)

### 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

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)O(mn)O(mn), 空间复杂度 O(mn)O(mn)O(mn).

### Reference

* [Google-APAC2015-"Password Attacker" - dmsehuang的专栏](http://blog.csdn.net/dmsehuang/article/details/40807799)

# Microsoft

# Microsoft

本章总结 Microsoft 校招的一些题。

## Microsoft 2015 April

# Microsoft 2015 April

本小节总结 Microsoft 2015年四月第一次招实习生的题，题目列表见 [hihoCoder](http://hihocoder.com/contest/mstest2015april/problems).

### Problem A. Magic Box

# Problem A. Magic Box

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015april/problem/1)

### Problem

鏃堕棿闄愬埗:10000ms

鍗曠偣鏃堕檺:1000ms

鍐呭瓨闄愬埗:256MB

### 鎻忚堪

The circus clown Sunny has a magic box. When the circus is performing, Sunnyputs some balls into the box one by one. The balls are in three colors:red(R), yellow(Y) and blue(B). Let Cr, Cy, Cb denote the numbers of red,yellow, blue balls in the box. Whenever the differences among Cr, Cy, Cbhappen to be x, y, z, all balls in the box vanish. Given x, y, z and thesequence in which Sunny put the balls, you are to find what is the maximumnumber of balls in the box **ever**.

For example, let's assume x=1, y=2, z=3 and the sequence is RRYBRBRYBRY. AfterSunny puts the first 7 balls, RRYBRBR, into the box, Cr, Cy, Cb are 4, 1, 2respectively. The differences are exactly 1, 2, 3. (|Cr-Cy|=3, |Cy-Cb|=1, |Cb-Cr|=2) Then all the 7 balls vanish. Finally there are 4 balls in the box,after Sunny puts the remaining balls. So the box contains 7 balls at most,after Sunny puts the first 7 balls and before they vanish.

#### 杈撳叆

Line 1: x y z

Line 2: the sequence consisting of only three characters 'R', 'Y' and 'B'.

For 30% data, the length of the sequence is no more than 200.

For 100% data, the length of the sequence is no more than 20,000, 0 <= x,y, z <= 20.

### 杈撳嚭

The maximum number of balls in the box **ever**.

### 鎻愮ず

Another Sample

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| (0 0 0)RBYRRBY | 4 |

鏍蜂緥杈撳叆

1 2 3  
RRYBRBRYBRY

鏍蜂緥杈撳嚭

7

### Problem B. Professor Q's Software

# Problem B. Professor Q's Software

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015april/problem/2)

### Problem

鏃堕棿闄愬埗:10000ms

鍗曠偣鏃堕檺:1000ms

鍐呭瓨闄愬埗:256MB

### 鎻忚堪

Professor Q develops a new software. The software consists of N modules whichare numbered from 1 to N. The i-th module will be started up by signal Si. Ifsignal Si is generated multiple times, the i-th module will also be startedmultiple times. Two different modules may be started up by the same signal.During its lifecircle, the i-th module will generate Ki signals: E1, E2, ...,EKi. These signals may start up other modules and so on. Fortunately thesoftware is so carefully designed that **there is no loop in the startingchain of modules**, which means eventually all the modules will be stoped.Professor Q generates some initial signals and want to know how many timeseach module is started.

### 杈撳叆

The first line contains an integer T, the number of test cases. T test casesfollows.

For each test case, the first line contains contains two numbers N and M,indicating the number of modules and number of signals that Professor Qgenerates initially.

The second line contains M integers, indicating the signals that Professor Qgenerates initially.

Line 3~N + 2, each line describes an module, following the format S, K, E1,E2, ... , EK. S represents the signal that start up this module. K representsthe total amount of signals that are generated during the lifecircle of thismodule. And E1 ... EK are these signals.

For 20% data, all N, M <= 10For 40% data, all N, M <= 103For 100% data, all 1 <= T <= 5, N, M <= 105, 0 <= K <= 3, 0<= S, E <= 105.

**Hint: HUGE input in this problem. Fast IO such as scanf and BufferedReader are recommended.**

### 杈撳嚭

For each test case, output a line with N numbers Ans1, Ans2, ... , AnsN. Ansiis the number of times that the i-th module is started. In case the answersmay be too large, output the answers modulo 142857 (the remainder of divisionby 142857).

鏍蜂緥杈撳叆

3  
3 2  
123 256  
123 2 456 256  
456 3 666 111 256  
256 1 90  
3 1  
100  
100 2 200 200  
200 1 300  
200 0  
5 1  
1  
1 2 2 3  
2 2 3 4  
3 2 4 5  
4 2 5 6  
5 2 6 7

鏍蜂緥杈撳嚭

1 1 3  
1 2 2  
1 1 2 3 5

### Problem C. Islands Travel

# Problem C. Islands Travel

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015april/problem/3)

### Problem

时间限制:10000ms

单点时限:1000ms

内存限制:256MB

### 描述

There are N islands on a planet whose coordinates are (X1, Y1), (X2, Y2), (X3,Y3) ..., (XN, YN). You starts at the 1st island (X1, Y1) and your destinationis the n-th island (XN, YN). Travelling between i-th and j-th islands willcost you min{|Xi-Xj|, |Yi-Yj|} (|a| denotes the absolute value of a. min{a, b}denotes the smaller value between a and b) gold coins. You want to know whatis the minimum cost to travel from the 1st island to the n-th island.

### 输入

Line 1: an integer N.

Line 2~N+1: each line contains two integers Xi and Yi.

For 40% data, N<=1000，0<=Xi,Yi<=100000.

For 100% data, N<=100000，0<=Xi,Yi<=1000000000.

### 输出

Output the minimum cost.

样例输入

3  
2 2  
1 7  
7 6

样例输出

2

### Problem D. Recruitment

# Problem D. Recruitment

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015april/problem/4)

### Problem

鏃堕棿闄愬埗:10000ms

鍗曠偣鏃堕檺:1000ms

鍐呭瓨闄愬埗:256MB

### 鎻忚堪

A company plans to recruit some new employees. There are N candidates (indexedfrom 1 to N) have taken the recruitment examination. After the examination,the well-estimated ability value as well as the expected salary per year ofeach candidate is collected by the Human Resource Department.

Now the company need to choose their new employees according to these data. Tomaximize the company's benefits, some principles should be followed:

1. There should be exactly X males and Y females.
2. The sum of salaries per year of the chosen candidates should not exceedthe given budget B.
3. The sum of ability values of the chosen candidates should be maximum,without breaking the previous principles. Based on this, the sum of the salaryper year should be minimum.
4. If there are multiple answers, choose the lexicographically smallest one.In other words, you should minimize the smallest index of the chosencandidates; If there are still multiple answers, then minimize the secondsmallest index; If still multiple answers, then minimize the third smallestone; ...

Your task is to help the company choose the new employees from thosecandidates.

### 杈撳叆

The first line contains four integers N, X, Y, and B, separated by a singlespace. The meanings of all these variables are showed in the descriptionabove. 1 <= N <= 100, 0 <= X <= N, 0 <= Y <= N, 1 <= X +Y <= N, 1 <= B <= 1000.

Then follows N lines. The i-th line contains the data of the i-th candidate: acharacter G, and two integers V and S, separated by a single space. Gindicates the gender (either "M" for male, or "F" for female), V is the well-estimated ability value and S is the expected salary per year of thiscandidate. 1 <= V <= 10000, 0 <= S <= 10.

We assure that there is always at least one possible answer.

### 杈撳嚭

On the first line, output the sum of ability values and the sum of salariesper year of the chosen candidates, separated by a single space.

On the second line, output the indexes of the chosen candidates in ascendingorder, separated by a single space.

鏍蜂緥杈撳叆

4 1 1 10  
F 2 3  
M 7 6  
M 3 2  
F 9 9

鏍蜂緥杈撳嚭

9 9  
1 2

## Microsoft 2015 April 2

# Microsoft 2015 April 2

本小节总结 Microsoft 2015年四月第二次招实习生的题，题目列表见 [hihoCoder](http://hihocoder.com/contest/mstest2015april2/problems).

### Problem A. Lucky Substrings

# Problem A. Lucky Substrings

### Source

* [hihoCoder](http://hihocoder.com/problemset/problem/1152)

### Problem

时间限制:10000ms

单点时限:1000ms

内存限制:256MB

### 描述

A string s is **LUCKY** if and only if the number of different characters in sis a [fibonacci number](http://en.wikipedia.org/wiki/Fibonacci_number). Givena string consisting of only lower case letters, output all its lucky non-emptysubstrings in lexicographical order. Same substrings should be printed once.

### 输入

A string consisting no more than 100 lower case letters.

### 输出

Output the lucky substrings in lexicographical order, one per line. Samesubstrings should be printed once.

样例输入

aabcd

样例输出

a  
aa  
aab  
aabc  
ab  
abc  
b  
bc  
bcd  
c  
cd  
d

### 题解

简单实现题，即判断 substring 中不同字符串的个数是否为 fibonacci 数，最后以字典序方式输出，且输出的字符串中相同的只输出一次。分析下来需要做如下几件事：

1. 两重 for 循环取输入字符串的所有可能子串。
2. 判断子串中不同字符的数目，这里使用可以去重的数据结构 Set 比较合适，最后输出 Set 的大小即为不同字符的数目。
3. 判断不同字符数是否为 fibonacci 数，由于子串数目较多，故 fibonacci 应该首先生成，由于字符串输入最大长度为100，故使用哈希表这种查询时间复杂度为 O(1)O(1)O(1) 的数据结构。
4. 将符合条件的子串加入到最终结果，由于结果需要去重，故选用 Set 数据结构。

### Java

import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner in = new Scanner(System.in);  
 String input = in.nextLine();  
 Set<String> result = solve(input);  
 for (String s : result) {  
 System.out.println(s);  
 }  
 }  
  
 public static Set<String> solve(String input) {  
 Set<Long> fibonacci = fibonacci\_number(input.length());  
 Set<String> res = new TreeSet<String>();  
 for (int i = 0; i < input.length(); i++) {  
 for (int j = i + 1; j <= input.length(); j++) {  
 String substr = input.substring(i, j);  
 if (isFibonacci(substr, fibonacci)) {  
 res.add(substr);  
 }  
 }  
 }  
  
 return res;  
 }  
  
 public static boolean isFibonacci(String s, Set<Long> fibo) {  
 Set<Character> charSet = new HashSet<Character>();  
 for (Character c : s.toCharArray()) {  
 charSet.add(c);  
 }  
 // convert charSet.size() to long  
 if (fibo.contains((long)charSet.size())) {  
 return true;  
 } else {  
 return false;  
 }  
 }  
  
 public static Set<Long> fibonacci\_number(int n) {  
 // generate fibonacci number till n  
 Set<Long> fibonacci = new HashSet<Long>();  
 long fn2 = 1, fn1 = 1, fn = 1;  
 fibonacci.add(fn);  
 for (int i = 3; i <= n; i++) {  
 fn = fn1 + fn2;  
 fibonacci.add(fn);  
 fn2 = fn1;  
 fn1 = fn;  
 }  
 return fibonacci;  
 }  
}

### 源码分析

fibonacci 数组的生成使用迭代的方式，由于保存的是 Long 类型，故在判断子串 size 时需要将 size 转换为 long . Java 中常用的 Set 有两种，无序的 HashSet 和有序的 TreeSet .

### 复杂度分析

遍历所有可能子串，时间复杂度 O(n2)O(n^2)O(n2), fibonacci 数组和临时子串，空间复杂度 O(n)O(n)O(n).

### Problem B. Numeric Keypad

# Problem B. Numeric Keypad

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015april2/problem/2)

### Problem

鏃堕棿闄愬埗:10000ms

鍗曠偣鏃堕檺:1000ms

鍐呭瓨闄愬埗:256MB

### 鎻忚堪

The numberic keypad on your mobile phone looks like below:

1 2 3  
4 5 6  
7 8 9  
 0

Suppose you are holding your mobile phone with single hand. Your thumb pointsat digit 1. Each time you can 1) press the digit your thumb pointing at, 2)move your thumb right, 3) move your thumb down. Moving your thumb left or upis not allowed.

By using the numeric keypad under above constrains, you can produce somenumbers like 177 or 480 while producing other numbers like 590 or 52 isimpossible.

Given a number K, find out the maximum number less than or equal to K that canbe produced.

### 杈撳叆

The first line contains an integer T, the number of testcases.

Each testcase occupies a single line with an integer K.

For 50% of the data, 1 <= K <= 999.

For 100% of the data, 1 <= K <= 10500, t <= 20.

### 杈撳嚭

For each testcase output one line, the maximum number less than or equal tothe corresponding K that can be produced.

鏍蜂緥杈撳叆

3  
25  
83  
131

鏍蜂緥杈撳嚭

25  
80  
129

### Problem C. Spring Outing

# Problem C. Spring Outing

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015april2/problem/3)

### Problem

鏃堕棿闄愬埗:20000ms

鍗曠偣鏃堕檺:1000ms

鍐呭瓨闄愬埗:256MB

### 鎻忚堪

You class are planning for a spring outing. N people are voting for adestination out of K candidate places.

The voting progress is below:

First the class vote for the first candidate place. If more than half of theclass agreed on the place, the place is selected. The voting ends.

Otherwise they vote for the second candidate place. If more than half of theclass agreed on the place, the place is selected. The voting ends.

Otherwise they vote for the third candidate place in the same way and go on.

If no place is selected at last there will be no spring outing and everybodystays at home.

Before the voting, the Chief Entertainment Officer did a survey, found outevery one's preference which can be represented as a permutation of 0, 1, ...K. (0 is for staying at home.) For example, when K=3, preference "1, 0, 2, 3"means that the first place is his first choice, staying at home is the secondchoice, the second place is the third choice and the third place is the lastchoice.

The Chief Entertainment Officer sends the survey results to the class. Soeverybody knows the others' preferences. Everybody wants his more preferedplace to be selected. And they are very smart, they always choose the optimalstrategy in the voting progress to achieve his goal.

Can you predict which place will be selected?

### 杈撳叆

The first line contains two integers, N and K, the number of people in yourclass and the number of candidate places.

The next N lines each contain a permutation of 0~K, representing someone'spreference.

For 40% of the data, 1 <= N, K <= 10

For 100% of the data, 1 <= N, K <= 1000

### 杈撳嚭

Output the selected place. Or "otaku" without quotes if no place is selected.

### 鏍蜂緥鎻愮ず

In the sample case, if the second peoson vote against the first place, noplace would be selected finally because the first person must vote against thesecond place for his own interest. Considering staying at home is a worsechoice than the first place, the second person's optimal strategy is votingfor the first place. So the first place will be selected.

鏍蜂緥杈撳叆

2 2  
1 0 2  
2 1 0

鏍蜂緥杈撳嚭

1

## Microsoft 2015 September 2

# Microsoft 2015 September 2

本小节总结 Microsoft 2015年九月第一次大规模校招的题，题目列表见 [hihoCoder](http://hihocoder.com/contest/mstest2015sept2/problems). 这一场的题目感觉偏难。

### Problem A. Farthest Point

# Problem A. Farthest Point（圆周上最远整点）

### Source

* [hihoCoder](http://hihocoder.com/contest/mstest2015sept2/problem/1)

### Problem

时间限制:5000ms

单点时限:1000ms

内存限制:256MB

### 描述

Given a circle on a two-dimentional plane.

Output the **integral** point in or on the boundary of the circle which hasthe largest distance from the center.

### 输入

One line with three floats which are all accurate to three decimal places,indicating the coordinates of the center x, y and the radius r.

For 80% of the data: |x|,|y|<=1000, 1<=r<=1000

For 100% of the data: |x|,|y|<=100000, 1<=r<=100000

### 输出

One line with two integers separated by one space, indicating the answer.

If there are multiple answers, print the one with the largest x-coordinate.

If there are still multiple answers, print the one with the largesty-coordinate.

#### 样例输入

1.000 1.000 5.000

#### 样例输出

6 1

### 题解1 - 圆周枚举

其实自己最开始做这道题时用的就是枚举，但是似乎忘记加圆心坐标了，一直 WA... 题目要求是返回最大的 x, 所以我们首先根据半径范围将 x 的整数解范围求出来。然后求出可能的 y, 由于题中给出的解有3位小数，如果要精确求解的话，可以将圆方程两边同乘1000，然后判断是否为整数。

### Java

import java.io.\*;  
import java.util.\*;  
import java.util.Queue;  
  
class Point {  
 long x;  
 long y;  
 Point(long x, long y) {  
 this.x = x;  
 this.y = y;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner in = new Scanner(System.in);  
 double xd = in.nextDouble(), yd = in.nextDouble(), rd = in.nextDouble();  
 Point result = solve(xd, yd, rd);  
 System.out.println(result.x + " " + result.y);  
 }  
  
 private static Point solve(double x0, double y0, double r) {  
 // convert double to long(accurate)  
 long xl0 = (long)(x0 \* 1000), yl0 = (long)(y0 \* 1000), rl0 = (long)(r \* 1000);  
 Point res = new Point(Long.MIN\_VALUE, Long.MIN\_VALUE);  
 int lower\_x = (int)Math.ceil(x0 - r), upper\_x = (int)Math.floor(x0 + r);  
 for (int i = upper\_x; i >= lower\_x; i--) {  
 // circle above  
 long y1l = yl0 + (long)(Math.sqrt(rl0\*rl0 - (i\*1000 - xl0)\*(i\*1000 - xl0)) + 0.5);  
 if ((i\*1000 - xl0)\*(i\*1000 - xl0) + (y1l - yl0)\*(y1l - yl0) == rl0\*rl0) {  
 // ensure y1 is integer  
 if (y1l % 1000 == 0) {  
 res.x = i;  
 res.y = y1l / 1000;  
 return res;  
 }  
 }  
 // circle below  
 y1l = yl0 - (long)(Math.sqrt(rl0\*rl0 - (i\*1000 - xl0)\*(i\*1000 - xl0)) + 0.5);  
 if ((i\*1000 - xl0)\*(i\*1000 - xl0) + (y1l - yl0)\*(y1l - yl0) == rl0\*rl0) {  
 // ensure y1 is integer  
 if (y1l % 1000 == 0) {  
 res.x = i;  
 res.y = y1l / 1000;  
 return res;  
 }  
 }  
 }  
  
 return res;  
 }  
}

### 源码分析

自右向左枚举，先枚举圆的上半部分，再枚举圆的下半部分。注意1000的转换。

### 复杂度分析

最坏情况下 O(R)O(R)O(R).

### 题解2 - 整数分解

看似容易实则比较难的一道题，现场通过率非常低。我们仔细审下题，求圆周上的整点，有多个整点时输出最大的 x 和最大的 y. 容易想到的方案是枚举所有可能的 x 和 y, 然后代入等式测试是否相等，这个过不了大的 x 和 y. 如果用开方的方法必然有误差，我用这种方法不知道贡献了多少 WA, 泪流满面... 作为在线测试，**更为合理的方案应为先暴力搜索拿到百分之八十的分数。**

从 Microsoft 和 Google APAC 在线测试的风格来看是偏向于程序设计竞赛的，那么题目的考点自然就在竞赛范围之内，这道题看似是浮点型的数据，实际上考的却是整数中数论的基础。**注意题中的 accurate to three decimal places, 那么也就意味着我们对给定的数据同乘 10310^3103 后一定是整数！！**！这个关键的信息我在测试过程中也没注意到，直到第二天早上醒来后突然就想到了！兴奋地六点多就爬起来了。

首先肯定是要写出圆方程的，设圆心坐标为 (x0,y0)(x\_0, y\_0)(x0,y0), 半径为 rrr, 那么我们有：(x−x0)2+(y−y0)2=r2(x - x\_0)^2 + (y - y\_0)^2 = r^2(x−x0)2+(y−y0)2=r2

设 m=103(x−x0)m = 10^3(x - x\_0)m=103(x−x0), n=103(y−y0)n = 10^3(y - y\_0)n=103(y−y0), R=103rR = 10^3rR=103r, 那么我们有新的圆方程：m2+n2=R2m^2 + n^2 = R^2m2+n2=R2其中 m, n, R 均为整数。接下来我们看看给出的数据范围，x, y, r 均是 10610^6106 以内，那么圆方程两边同乘 10610^6106 （括号内的数即乘上 10310^3103）后数据在 101810^{18}1018 以内。我们来估算下整数的范围，210≈1032^{10} \approx 10^3210≈103, Java 中 int 型为4个字节，最大为 231−1≈2⋅1092^{31} - 1 \approx 2 \cdot 10^9231−1≈2⋅109, long 型为8个字节，最大为 263−1≈23⋅10182^{63} - 1 \approx 2^3 \cdot 10^{18}263−1≈23⋅1018, 估算下来应该选用 long 保存 m, n, R.

接下来就是数论部分的推导了，先来一个简单的推导，勾股数部分的推导不直观。首先从已知部分出发，已知的只有勾股数方程和 m, n 均是整数，那么接下来肯定是要利用整数的理论无疑了。我们首先对以上圆方程移项开方，考虑到圆的对称性，我们其实只需要考虑圆的八分之一即可。这里考虑 0 < m < r 部分， m == 0 表示在点在轴上，最后单独加上。

m=R2−n2=(R+n)(R−n)m = \sqrt{R^2 - n^2} = \sqrt{(R + n)(R - n)}m=√R2−n2=√(R+n)(R−n)由于 m 一定是整数，故根号内一定为完全平方数，由于排除了轴上的点，那么 -R < n < R , 设 G = gcd(R + n, R - n) , p2=(R+n)/Gp^2 = (R + n) / Gp2=(R+n)/G, q2=(R−n)/Gq^2 = (R - n) / Gq2=(R−n)/G, 于是我们有 m = Gpq , p > q , 由于 G 是 R + n 和 R - n 的最大公约数，故 p 和 q 一定互质，且有：p2+q2=2R/Gp^2 + q^2 = 2R / Gp2+q2=2R/G由于 p , q 都大于等于1，那么我们能推断出 G 一定是 2R 的约数！根据约数(素数)部分的基础理论，我们可以在 O(2R)O(\sqrt{2R})O(√2R) 时间内找出所有约数。然后对以上等式进行缩放得到 p 的范围，枚举求解，判断 p^2 和 q^2 是否互质(最大公约数是否为1)。

### Java

import java.io.\*;  
import java.util.\*;  
  
class Point {  
 long x;  
 long y;  
 Point(long x, long y) {  
 this.x = x;  
 this.y = y;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner in = new Scanner(System.in);  
 double xd = in.nextDouble(), yd = in.nextDouble(), rd = in.nextDouble();  
 // convert double to long(accurate)  
 long x0 = (long)(xd \* 1000), y0 = (long)(yd \* 1000), r0 = (long)(rd \* 1000);  
 Point result = solve(x0, y0, r0);  
 System.out.println(result.x + " " + result.y);  
 }  
  
 private static Point solve(long x0, long y0, long r) {  
 Point res = new Point(Long.MIN\_VALUE, Long.MIN\_VALUE);  
 long x\_max = Long.MIN\_VALUE, y\_max = Long.MIN\_VALUE;  
 // p^2 + q^2 = 2R/divisor, p > q >= 1  
 // 1 <= q^2 < R/G < p^2 < 2R/G ==> p >= 2  
 List<Long> divisors = getDivisor(r << 1);  
 for (long divisor : divisors) {  
 long lower = Math.max(2, (long)Math.sqrt(r \* 1.0/ divisor));  
 // long upper = (long)Math.sqrt(2.0 \* r / divisor);  
 for (long p = lower; p \* p <= 2 \* r / divisor; p++) {  
 long q = (long)Math.sqrt(2.0 \* r / divisor - p \* p);  
 // check if q is integer  
 if (p \* p + q \* q != 2 \* r / divisor) continue;  
 // ensure p^2 and q^2 have no common divisor  
 if (gcd(p \* p, q \* q) == 1) {  
 long m = divisor \* p \* q;  
 long n = r - p \* p \* divisor;  
 List<Point> points = new ArrayList<Point>();  
 points.add(new Point(m + x0, n + y0));  
 points.add(new Point(m + x0, -1 \* n + y0));  
 points.add(new Point(-1 \* m + x0, n + y0));  
 points.add(new Point(-1 \* m + x0, -1 \* n + y0));  
 for (Point point : points) {  
 updateAns(point, res);  
 }  
 }  
 }  
 }  
  
 // axis point check  
 List<Point> axis = new ArrayList<Point>();  
 axis.add(new Point(x0 + r, y0));  
 axis.add(new Point(x0 - r, y0));  
 axis.add(new Point(x0, y0 + r));  
 axis.add(new Point(x0, y0 - r));  
 for (Point point : axis) {  
 updateAns(point, res);  
 }  
 // divide by 1000  
 res.x /= 1000;  
 res.y /= 1000;  
  
 return res;  
 }  
  
 public static void updateAns(Point p, Point res) {  
 // point(x, y) in integer  
 if ((p.x % 1000 == 0) && (p.y % 1000 == 0)) {  
 if (p.x > res.x) {  
 res.x = p.x;  
 res.y = p.y;  
 } else if (p.x == res.x && p.y > res.y) {  
 res.y = p.y;  
 }  
 }  
 }  
  
 // enumerate all the divisor for n  
 public static List<Long> getDivisor(long n) {  
 List<Long> result = new ArrayList<Long>();  
 for (long i = 1; i \* i <= n; i++) {  
 if (n % i == 0) {  
 result.add(i);  
 // i \* i <= n ==> i <= n / i  
 if (i != n / i) result.add(n / i);  
 }  
 }  
 Collections.sort(result);  
 return result;  
 }  
  
 public static long gcd(long a, long b) {  
 return (b == 0L) ? a : gcd(b, a % b);  
 }  
}

### 源码分析

由于更新结果的操作非常频繁，单独写一个方法较好。

### 复杂度分析

求所有素数时间复杂度 O(n)O(\sqrt{n})O(√n), 判断是否互质时间复杂度 O(logn)O(\log n)O(logn). 枚举最大公约数时间复杂度约 (n)(\sqrt{n})(√n)，总的时间复杂度估算应该比 O(n)O(n)O(n) 小一些，但是小的不明显。**所以说，这种方法费了老大劲，但是吃力不讨好！笔试中这种方法极不可取！**

### 题解3 - 勾股数

除了以上使用数论部分整数分解的方法外，还可以巧用勾股数的特性，这种方法需要熟知勾股数的特性。设正整数 m,n,rm, n, rm,n,r 满足：m2+n2=r2m^2 + n^2 = r^2m2+n2=r2我们对上式两边进行平方可得：(m2−n2)2+(2mn)2=(m2+n2)2=(r2)2(m^2 - n^2)^2 + (2mn)^2 = (m^2 + n^2)^2 = (r^2)^2(m2−n2)2+(2mn)2=(m2+n2)2=(r2)2令 a=m2−n2a = m^2 - n^2a=m2−n2, b=2mnb = 2mnb=2mn, c=m2+n2c = m^2 + n^2c=m2+n2. 容易得到：a2+b2=c2a^2 + b^2 = c^2a2+b2=c2注意到上述推导可逆，那么也就是说只要我们找到正整数满足 m > n 就能找到所有可能的勾股数。且根据素勾股数的特性， m , n 为一奇一偶，不妨设其为 2k-1 , 2k . 代入 c 中可知 c 为 4K + 1 . 即 c % 4 = 1 . 根据 [Tree of primitive Pythagorean triples](https://en.wikipedia.org/wiki/Tree_of_primitive_Pythagorean_triples) 中提到的方法，我们只需找出小于给定的 r 的素勾股数即可，然后判断是否能整除 r .

### Java

import java.io.\*;  
import java.util.\*;  
import java.util.Queue;  
  
class Point {  
 long x;  
 long y;  
 Point(long x, long y) {  
 this.x = x;  
 this.y = y;  
 }  
}  
  
class Pythagorean {  
 long x;  
 long y;  
 long z;  
 Pythagorean(long x, long y, long z) {  
 this.x = x;  
 this.y = y;  
 this.z = z;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner in = new Scanner(System.in);  
 double xd = in.nextDouble(), yd = in.nextDouble(), rd = in.nextDouble();  
 // convert double to long(accurate)  
 long x0 = (long)(xd \* 1000), y0 = (long)(yd \* 1000), r0 = (long)(rd \* 1000);  
 Point result = solve(x0, y0, r0);  
 System.out.println(result.x + " " + result.y);  
 }  
  
 private static Point solve(long x0, long y0, long r) {  
 Point res = new Point(Long.MIN\_VALUE, Long.MIN\_VALUE);  
 long x\_max = Long.MIN\_VALUE, y\_max = Long.MIN\_VALUE;  
 // init  
 Pythagorean pyth0 = new Pythagorean(3, 4, 5);  
 Queue<Pythagorean> q = new LinkedList<Pythagorean>();  
 q.offer(pyth0);  
 boolean update = true;  
 while (!q.isEmpty()) {  
 int qSize = q.size();  
 for (int i = 0; i < qSize; i++) {  
 Pythagorean pyth = q.poll();  
 if ((r % pyth.z) == 0) {  
 // System.out.println("x = " + pyth.x + ", y = " + pyth.y + ", r = " + pyth.z);  
 long k = r / pyth.z;  
 long kx = k \* pyth.x;  
 long ky = k \* pyth.y;  
 List<Point> points = new ArrayList<Point>();  
 points.add(new Point(x0 + kx, y0 + ky));  
 points.add(new Point(x0 - kx, y0 + ky));  
 points.add(new Point(x0 + kx, y0 - ky));  
 points.add(new Point(x0 - kx, y0 - ky));  
 if (kx != ky) {  
 points.add(new Point(y0 + ky, x0 + kx));  
 points.add(new Point(y0 - ky, x0 + kx));  
 points.add(new Point(y0 + ky, x0 - kx));  
 points.add(new Point(y0 - ky, x0 - kx));  
 }  
 for (Point point : points) {  
 updateAns(point, res);  
 }  
 }  
 // add next level Pythagorean  
 for (Pythagorean p : nextPyths(pyth)) {  
 if (p.z > r) continue;  
 q.offer(p);  
 }  
 }  
 }  
  
 // axis point check  
 List<Point> axis = new ArrayList<Point>();  
 axis.add(new Point(x0 + r, y0));  
 axis.add(new Point(x0 - r, y0));  
 axis.add(new Point(x0, y0 + r));  
 axis.add(new Point(x0, y0 - r));  
 for (Point point : axis) {  
 updateAns(point, res);  
 }  
 // divide by 1000  
 res.x /= 1000;  
 res.y /= 1000;  
  
 return res;  
 }  
  
 public static List<Pythagorean> nextPyths(Pythagorean pyth) {  
 List<Pythagorean> pyths = new ArrayList<Pythagorean>();  
 // method 1  
 Pythagorean pyth1 = new Pythagorean(0, 0, 0);  
 pyth1.x = pyth.x - 2 \* pyth.y + 2 \* pyth.z;  
 pyth1.y = 2 \* pyth.x - 1 \* pyth.y + 2 \* pyth.z;  
 pyth1.z = 2 \* pyth.x - 2 \* pyth.y + 3 \* pyth.z;  
 pyths.add(pyth1);  
 // method 2  
 Pythagorean pyth2 = new Pythagorean(0, 0, 0);  
 pyth2.x = pyth.x + 2 \* pyth.y + 2 \* pyth.z;  
 pyth2.y = 2 \* pyth.x + 1 \* pyth.y + 2 \* pyth.z;  
 pyth2.z = 2 \* pyth.x + 2 \* pyth.y + 3 \* pyth.z;  
 pyths.add(pyth2);  
 // method 3  
 Pythagorean pyth3 = new Pythagorean(0, 0, 0);  
 pyth3.x = -1 \* pyth.x + 2 \* pyth.y + 2 \* pyth.z;  
 pyth3.y = -2 \* pyth.x + pyth.y + 2 \* pyth.z;  
 pyth3.z = -2 \* pyth.x + 2 \* pyth.y + 3 \* pyth.z;  
 pyths.add(pyth3);  
  
 return pyths;  
 }  
  
 public static void updateAns(Point p, Point res) {  
 // point(x, y) in integer  
 if ((p.x % 1000 == 0) && (p.y % 1000 == 0)) {  
 if (p.x > res.x) {  
 res.x = p.x;  
 res.y = p.y;  
 } else if (p.x == res.x && p.y > res.y) {  
 res.y = p.y;  
 }  
 }  
 }  
}

### 源码分析

根据链接中提到的数据结构，使用队列按层次遍历较好，但是空间消耗较大，所以在入队时一定要剪枝。

### 复杂度分析

时间复杂度最坏情况下需要遍历所有可能素勾股数。空间复杂度消耗也比较客观...

### Reference

* [BZOJ 1041 [HAOI2008] 圆上的整点 题解与分析 - 初学者 - 博客频道 - CSDN.NET](http://blog.csdn.net/csyzcyj/article/details/10044629)
* [[BZOJ1041 [HAOI2008]圆上的整点]数论、勾股数相关定理 | edward\_mj](http://edward-mj.com/archives/166)
* [勾股数 - 维基百科，自由的百科全书](https://zh.wikipedia.org/wiki/%E5%8B%BE%E8%82%A1%E6%95%B0)
* [hihoCoder](http://hihocoder.com/discuss/question/2619)

# Appendix I Interview and Resume

# 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

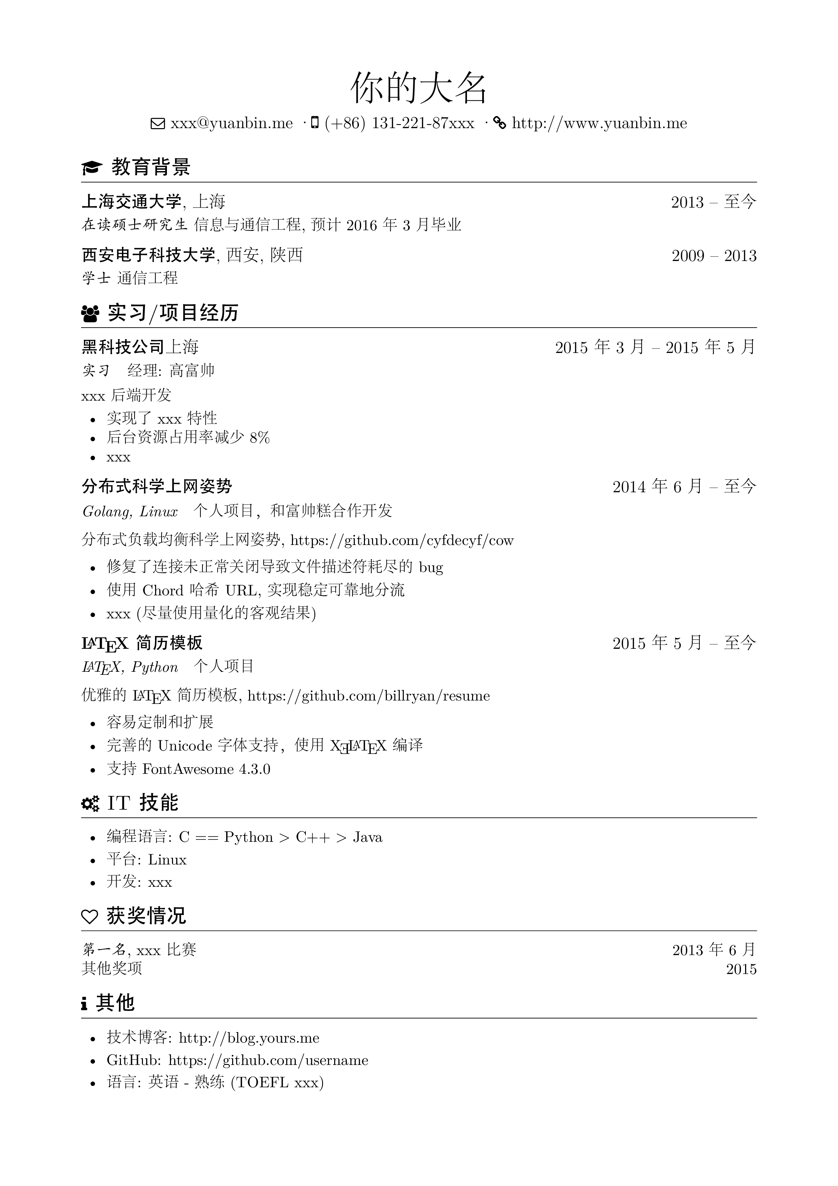
## Resume

# 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](https://www.sharelatex.com/templates/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](https://www.sharelatex.com/templates/cv-or-resume/professional-cv) 相对紧凑一些，[LaTeX Templates » Curricula Vitae/Résumés](http://www.latextemplates.com/cat/curricula-vitae) 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](https://github.com/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。



### 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](http://get.jobdeer.com/744.get) - 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](http://zh.lucida.me/blog/lean-technical-resume/) - 『Effective 简历』系列。
* [如何把简历写进一页 - V2EX](https://www.v2ex.com/t/175250) - 众人支招助萌妹纸优化简历。
* *Cracking the coding interview* 『写好简历』一节。