

Data Structure and Algorithm

yuanbin

Published
with GitBook



目錄

1. [leetcode/lintcode题解/算法学习笔记](#)
2. [Part I - Basics](#)
3. [Data Structure](#)
 - i. [Linked List](#)
 - ii. [Binary Tree](#)
 - iii. [Binary Search Tree](#)
 - iv. [Huffman Compression](#)
 - v. [Priority Queue](#)
4. [Basics Sorting](#)
 - i. [Bubble Sort](#)
 - ii. [Selection Sort](#)
 - iii. [Insertion Sort](#)
 - iv. [Merge Sort](#)
 - v. [Quick Sort](#)
 - vi. [Heap Sort](#)
 - vii. [Bucket Sort](#)
 - viii. [Counting Sort](#)
5. [Basics Misc](#)
 - i. [Bit Manipulation](#)
6. [Part II - Coding](#)
7. [String - 字符串](#)
 - i. [strStr](#)
 - ii. [Two Strings Are Anagrams](#)
 - iii. [Compare Strings](#)
 - iv. [Anagrams](#)
 - v. [Longest Common Substring](#)
8. [Integer Array - 整型数组](#)
 - i. [Remove Element](#)
 - ii. [Zero Sum Subarray](#)
 - iii. [Subarray Sum K](#)
 - iv. [Subarray Sum Closest](#)
 - v. [Product of Array Exclude Itself](#)
 - vi. [Partition Array](#)
 - vii. [First Missing Positive](#)
 - viii. [2 Sum](#)
 - ix. [3 Sum](#)
 - x. [3 Sum Closest](#)
 - xi. [Remove Duplicates from Sorted Array](#)
 - xii. [Remove Duplicates from Sorted Array II](#)
 - xiii. [Merge Sorted Array](#)
 - xiv. [Merge Sorted Array II](#)
9. [Binary Search - 二分搜索](#)

- i. [Binary Search](#)
 - ii. [Search Insert Position](#)
 - iii. [Search for a Range](#)
 - iv. [First Bad Version](#)
 - v. [Search a 2D Matrix](#)
 - vi. [Find Peak Element](#)
 - vii. [Search in Rotated Sorted Array](#)
 - viii. [Find Minimum in Rotated Sorted Array](#)
 - ix. [Search a 2D Matrix II](#)
 - x. [Median of two Sorted Arrays](#)
 - xi. [Sqrt x](#)
 - xii. [Wood Cut](#)
- 10. [Math and Bit Manipulation - 数学技巧与位运算](#)
 - i. [Single Number](#)
 - ii. [Single Number II](#)
 - iii. [Single Number III](#)
 - iv. [O1 Check Power of 2](#)
 - v. [Convert Integer A to Integer B](#)
 - vi. [Factorial Trailing Zeroes](#)
- 11. [Linked List - 链表](#)
 - i. [Remove Duplicates from Sorted List](#)
 - ii. [Remove Duplicates from Sorted List II](#)
 - iii. [Partition List](#)
 - iv. [Two Lists Sum](#)
 - v. [Two Lists Sum Advanced](#)
 - vi. [Remove Nth Node From End of List](#)
 - vii. [Linked List Cycle](#)
 - viii. [Linked List Cycle II](#)
 - ix. [Reverse Linked List](#)
 - x. [Reverse Linked List II](#)
 - xi. [Merge Two Sorted Lists](#)
 - xii. [Merge k Sorted Lists](#)
 - xiii. [Sort List](#)
 - xiv. [Reorder List](#)
- 12. [Reverse - 翻转法](#)
 - i. [Recover Rotated Sorted Array](#)
 - ii. [Rotate String](#)
 - iii. [Reverse Words in a String](#)
- 13. [Binary Tree - 二叉树](#)
 - i. [Binary Tree Preorder Traversal](#)
 - ii. [Binary Tree Inorder Traversal](#)
 - iii. [Binary Tree Postorder Traversal](#)
 - iv. [Binary Tree Level Order Traversal](#)
 - v. [Maximum Depth of Binary Tree](#)
 - vi. [Balanced Binary Tree](#)
 - vii. [Binary Tree Maximum Path Sum](#)

- viii. [Lowest Common Ancestor](#)
- 14. [Binary Search Tree - 二叉搜索树](#)
 - i. [Insert Node in a Binary Search Tree](#)
 - ii. [Validate Binary Search Tree](#)
 - iii. [Search Range in Binary Search Tree](#)
 - iv. [Convert Sorted Array to Binary Search Tree](#)
 - v. [Convert Sorted List to Binary Search Tree](#)
 - vi. [Binary Search Tree Iterator](#)
- 15. [Backtracking - 回溯法](#)
 - i. [Subsets](#)
 - ii. [Permutation](#)
- 16. [Dynamic Programming - 动态规划](#)
 - i. [Triangle](#)
 - ii. [Knapsack - 背包问题](#)
 - i. [Backpack](#)
 - iii. [Matrix](#)
 - i. [Minimum Path Sum](#)
 - ii. [Unique Paths](#)
 - iv. [Sequence](#)
 - i. [Climbing Stairs](#)
 - ii. [Jump Game](#)
- 17. [術語表](#)

leetcode/lintcode题解/算法学习笔记

build passing

一晃就研二下了，离毕业也只有短短一年，终于快逃出无线通信的魔爪了，想想就有点小激动啊，由于自己是非CS科班出身，一些CS方面的基础肯定是在找工作/实习之前夯实的啦，比如数据结构和算法、编程语言、操作系统、数据库等等啦，最最重要的自然就是算法和编程语言了咯。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。

About - 关于本文档

主要内容为学习算法和刷leetcode/lintcode过程中的笔记，很大程度上参考了[九章算法](#)的代码和讲稿，先行谢过！同时也参考了一些其他教材和优质博客，凡参考过的几乎都给明确链接，如果不小心忘记了，请不要吝惜你的评论和issue :)

- 本笔记的在线托管仓库为 <https://github.com/billryan/algorithm-exercise> 你可以在github中star本项目查看更新。
- 在线阅读网址为 <http://algorithm.yuanbin.me> 在线阅读的网页通过gitbook后端生成，推送到github后会触发 gitbook 和 travis-ci 的编译，相应的编译输出下载链接提供Gitbook官网和七牛两种下载方式，七牛的链接中文显示比较好。
 1. Read on the [website](#). 力荐
 2. EPUB. [Gitbook](#), [七牛](#) - Recommended for iPhone/iPad/MAC. 最适合离线查看，实测效果极好。
 3. 离线html. [七牛](#) - 解压后即为整个网站的内容，用于本地查看，目前内容更新频繁，不推荐下载。
 4. PDF. [Gitbook](#), [七牛-适合打印版](#) - Recommended for Desktop. 推荐下载七牛的版本。
 5. MOBI. [Gitbook](#), [七牛](#) - Recommended for Kindle. 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...
- 全文大体上分为两大部分，第一部分为算法基础，是自己参考书籍及一些网页的总结；第二部分为代码实战，是自己在leetcode/lintcode上刷题的总结。

License(许可证)



如无特殊说明，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。欢迎 **fork** 和传播本文档，但是请注意遵循以上许可协议。

Contribution - 如何贡献本文档

如果你发现本文档有任何可以改进之处，欢迎提交你的改进，具体形式有如下几种。

1. 成为本项目的contributor, 发邮件并把你的github账户名告诉我就可以了，我收到邮件后把你的github账号加到Collaborators中。
2. 提交Pull Request, fork本文档的github repo, 发PR给我就好了。
3. 在本文档的github repo处提交issue, 指出有问题的地方。

- 在 website <http://algorithm.yuanbin.me> 相应网页下的disqus评论框中添加评论，指出一些typo或者可以改进的地方。

既然涉及到文档合作，那么最好是能有个像样的文档规范之类的东西方便大家更好的合(jiao)作(ji)，目前想到的有如下几点。

全文组织架构

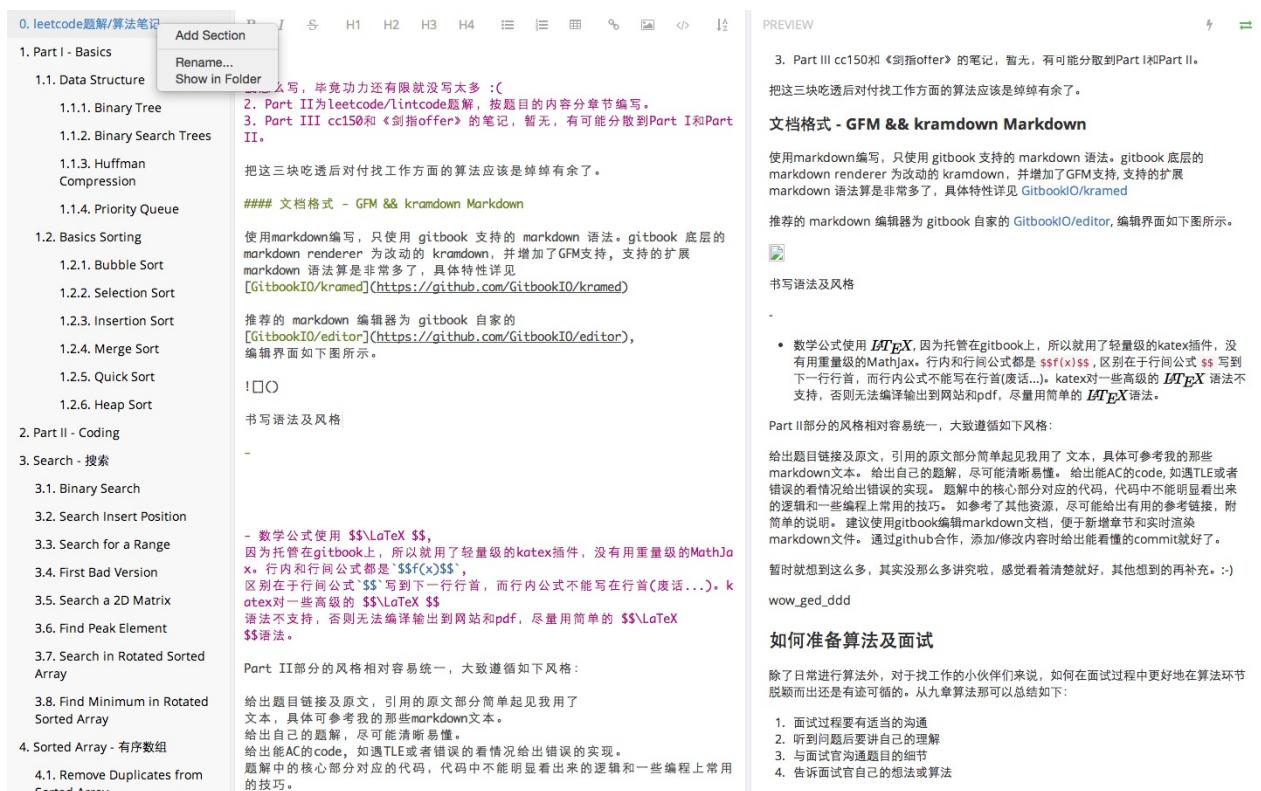
- Part I为基础知识复习，介绍一些基础的排序/链表/基础算法，这一块目前我没怎么写，毕竟功力有限就没写太多：(
- Part II为leetcode/lintcode题解，按题目的内容分章节编写。
- Part III cc150和《剑指offer》的笔记，暂无，有可能分散到Part I和Part II。

把这三块吃透后对付找工作方面的算法应该是绰绰有余了。

文档格式及编辑工具 - GFM & kramdown Markdown

使用markdown编写，只使用 gitbook 支持的 markdown 语法。gitbook 底层的 markdown renderer 为改动的 kramdown，并增加了GFM支持，支持的扩展 markdown 语法算是非常多了，具体特性详见 [GitbookIO/kramed](https://github.com/GitbookIO/kramed)

推荐的 markdown 编辑器为 gitbook 自家的 [GitbookIO/editor](https://github.com/GitbookIO/editor)，支持 Windows/Linux/MAC 三大平台，业界良心！但是实测在Arch Linux下可能会出现占用内存过高的情况... OS X 下目前表现还算良好，编辑界面如下图所示，最左边为章节预览，中间为 markdown 编辑框，右边为实时渲染页面，可选择使用全屏模式。



使用其他如 Mou/Vim/Emacs/Sublime Text也不错，但是在新增Chapter/Section时就比较闹心了，嗯，你也可以新建 Section 后再使用其他编辑器编辑。

对 Gitbook 不熟的建议看看 [Gitbook Documentation](https://github.com/GitbookIO/documentation)，有助于了解 <http://algorithm.yuanbin.me> 网页上的文

字及各章节等是如何编辑及渲染的。

章节名及编号

章节等文件名全部采用英文，子章节最多到三级，章节编号无需操心，这种琐事交给 Gitbook 去做就好，如果一定要手动调整，修改 SUMMARY.md 文件，注意其中的缩进关系，Gitbook就指望这个自动给章节编号了。

举个例子，我现在想新增「动态规划」及其子章节。首先在 Gitbook 顶部菜单栏「Book」中找到「Add Chapter」，填入「Dynamic Programming」。好了，在Gitbook左侧章节栏中就能看到新生成的「10. Dynamic Programming」了，左键击之，Gitbook 就会生成「dynamic_programming」目录及本章的说明文件「dynamic_programming/README.md」。如果想在「10. Dynamic Programming」下新增子章节，右键击之，「Add Section」即可，同上，子章节文件名仍然使用英文名，网页显示的标题可以通过 rename 更改再加入中文。

嗯，以上步骤均可直接新建文件夹及操作 SUMMARY.md 文件完成。

数学公式

其实代码里是用不着写数学公式的，但是偶尔分析算法可能会用着，用过 LaTeX 的都知道她生成的数学公式有多优雅，以至于不用她来写数学公式都有点不舒服...

这个文档里对于较复杂的数学公式建议使用 LaTeX, 因为托管在gitbook上，所以就用了轻量级的katex插件，没有用重量级的 MathJax。行内和行间公式都是两个\$, 区别在于行间公式写到下一行行首，而行内公式不能写在行首(废话...)。katex非常脆弱，对一些高级的 LaTeX 语法不支持，否则无法编译输出到网站和 pdf，尽量用简单的 LaTeX 语法或者不用。

正文书写风格

1. 中英文混排贯穿全文，优雅美观起见，尽可能在英文单词前后加空格，这个使能输入法的中英文间加入空格功能就好了。
2. 代码的函数名或短代码建议使用`code`
3. 使用空行进行分段，嗯，markdown通用

Part II为leetcode/lintcode题解，这部分的风格相对容易统一，大致遵循如下风格：

1. 给出题目链接及原文，引用的原文部分简单起见我对题目使用了blockquote，具体可参考我的那些markdown文本。
2. 给出自己的题解，尽可能清晰易懂。
3. 给出能AC的代码，如遇TLE或者错误的看情况给出错误的实现。使用blockquote, 给出语言类别以便高亮。具体可参看原markdown文件。
4. 题解中的核心部分对应的代码，代码中不能明显看出来的逻辑和一些编程上常用的技巧。
5. 如参考了其他资源，尽可能给出有用的参考链接，附简单的说明。

感觉还不错的风格 - [raw convert_sorted_list_to_binary_search_tree.md](#)

通过github合作时，添加/修改内容时给出能看懂的commit就好了。暂时就想到这么多，其实没那么多讲究啦，感觉看着清楚就好，其他想到的再补充。:-)

附件及图片引用

图片统一存放在 `images` 目录下，其他附件存放在 `docs` 目录下。引用图片链接一般可以通过 `![Caption]` (`../images/xxx.png`) 声明。

图片体积太大不利于页面加载，建议先压缩后再放入，如果是png图片可考虑使用 [TinyPNG – Compress PNG images while preserving transparency](#)

To-Do

- ☐ 探索适用于后期批处理的书写及排版格式。
- ☐ 添加多国语言支持(English, 繁體中文, 简体中文)。
- ☐ 完善 leetcode/lintcode 部分 C++, Java, Python 三大语言的实现。
- ☐ 加入时间/空间复杂度分析。
- ☐ Part I 部分基础知识的总结。
- ☐ CC150 书中题目及基础知识的引入。
- ☐ 添加一些较好的最近面试真题。
- ☒ 完善在线版本 <http://algorithm.yuanbin.me> 的 css, 使用 yahei 插件初步达到目标。
- ☒ 完善离线版本如 PDF(适合打印的字型) 的中文支持。
- ☐ 完善离线版本如 PDF(适合在电子屏上浏览的字型) 的中文支持。

如何准备算法及面试

除了日常进行算法练习外，对于找工作的小伙伴们来说，如何在面试过程中更好地在算法环节脱颖而出还是有迹可循的。从九章算法那可以总结如下：

1. 面试过程要有适当的沟通
2. 听到问题后要讲自己的理解
3. 与面试官沟通题目的细节
4. 告诉面试官自己的想法或算法

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

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

而对于实战算法的过程中，我们可以采取如下策略：

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

以下整理了一些最近练习算法的网站资源，和大家共享之。

在线OJ及部分题解

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

面试相关

本小节部分摘自九章微信的分享。

- [www.geeksforgeeks.org](#) - 非常著名的漏题网站之一。上面会时不时的有各种公司的面试真题漏出。有一些题也会有解法分析。
- [Programming Interview Questions | CareerCup](#) - CC150作者搞的网站，也是著名的漏题网站之一。大家会在上面讨论各个公司的面试题。
- [Glassdoor – Get Hired. Love Your Job.](#) - 一个给公司打分的网站，类似yelp的公司版。会有一些人在上面讨论面试题，适合你在面某个公司的时候专门去看一下。
- [面经网 | 汇集热气腾腾的求职咨询](#) - 面经网。应该是个人经营的一个积累面经的网站。面经来源主要是一亩三分地，mitbbs之类的地方。
- [一亩三分地论坛-美国加拿大留学申请|工作就业|英语考试|学习生活信噪比最高的网站](#) - 人气非常高的论坛。
- [待字闺中\(JobHunting\)版 | 未名空间\(mitbbs.com\) jobhunting版](#)，美华人找工作必上。
- [程序员面试：电话面试问答Top 50 - 博客 - 伯乐在线](#) - 其实不仅仅是 Top 50，扩展连接还给出了其他参考。
- [如何写好技术简历 —— 实例、模板及工具 | @Get社区](#) - 挺不错的技术简历实战。
- [想加入硅谷顶级科技公司，你该知道这些-LinkedIn中国](#) - 数据工程师董飞的求职分享。
- [求职在美国，面试攻略我知道 on Vimeo](#) - Coursera 数据工程师董飞的视频分享。

其他资源

- [九章算法](#) | 帮助更多的中国人找到好工作, 美国硅谷一线工程师实时在线授课 - 代码质量不错, 整理得也很好。
- [七月算法](#) - [julyedu.com](#) - july大神主导的在线算法辅导。
- [刷题 | 一亩三分地论坛](#) - 时不时就会有惊喜放出。
- [VisuAlgo](#) - [visualising data structures and algorithms through animation](#) - 相当碉堡的数据结构和算法可视化。
- [Data Structure Visualization](#) - 同上, 非常好的动画演示!! 涵盖了常用的各种数据结构/排序/算法。
- [结构之法 算法之道](#) - 不得不服!
- [julycoding/The-Art-Of-Programming-By-July](#) - 程序员面试艺术的电子版
- [程序员面试、算法研究、编程艺术、红黑树、数据挖掘5大系列集锦](#)
- 专栏: [算法笔记——《算法设计与分析》](#) - CSDN上对《算法设计与分析》一书的学习笔记。

书籍推荐

本节后三项参考自九章微信分享, 谢过。

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

Part I - Basics

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

本节主要由以下章节构成。

Reference

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

Data Structure - 数据结构

本章主要介绍一些基本的数据结构和算法。

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。

链表的基本操作

反转链表

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

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

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

删除链表中的某个节点

删除链表中的某个节点一定需要知道这个点的前继节点，所以需要一直有指针指向前继节点。

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

链表指针的鲁棒性

综合上面讨论的两种基本操作，链表操作时的鲁棒性问题主要包含两个情况：

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

Dummy Node

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

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

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

快慢指针

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

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

Binary Tree - 二叉树

二叉树是每个节点最多有两个子树的树结构，子树有左右之分，二叉树常被用于实现二叉查找树和二叉堆。

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

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

树的遍历

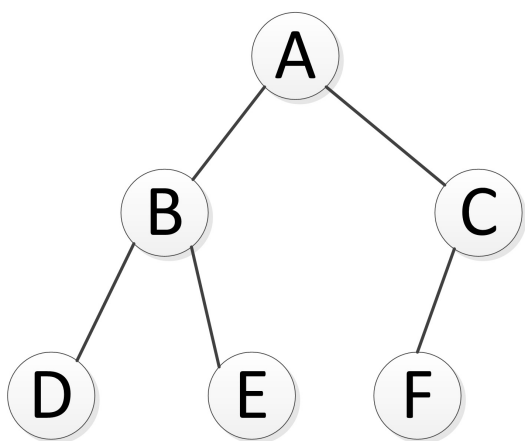
从二叉树的根节点出发，节点的遍历分为三个主要步骤：对当前节点进行操作（称为“访问”节点，或者根节点）、遍历左边子节点、遍历右边子节点。访问节点顺序的不同也就形成了不同的遍历方式。需要注意的是树的遍历通常使用递归的方法进行理解和实现，在访问元素时也需要使用递归的思想去理解。

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。



pre-order: **A** BDE CF
 in-order: D B E **A** F C
 post-order: D E B F C **A**
 level-order: **A** BC DEF

相关算法——递归法遍历

相关算法——分治法（Divide & Conquer）

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

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

定义：

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。

二叉树中每个节点只能有一个父节点(根节点无父节点)，只有左右两个链接，分别为左子节点和右子节点。

基本实现

节点包含

- 一个键
- 一个值
- 一条左链接
- 一条右链接

Huffman Compression - 霍夫曼压缩

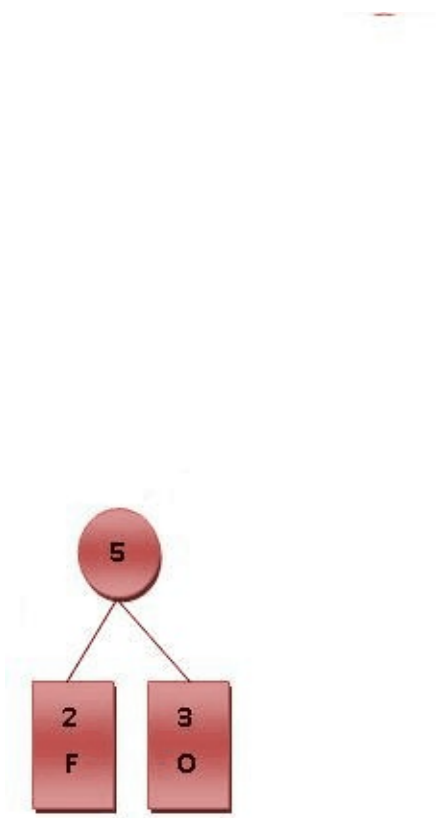
主要思想：放弃文本文件的普通保存方式：不再使用7位或8位二进制数表示每一个字符，而是用较少的比特表示出现频率最高的字符，用较多的比特表示出现频率低的字符。

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，其中最优化前缀码即为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
Frequency	2	3	4	4	5	7
Code	000	001	100	101	01	10

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



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

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，往往用堆来实现。

Reference

- [优先佇列 - 维基百科，自由的百科全书](#)

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. Quick 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

- [Sorting algorithm - Wikipedia, the free encyclopedia](#) - 各类排序算法的「平均、最好、最坏时间复杂度」总结。
- [经典排序算法总结与实现 | Jark's Blog](#) - 基于 Python 的较为清晰的总结。
- [【面经】硅谷前沿Startup面试经验-排序算法总结及快速排序算法代码_九章算法](#) - 总结了一些常用常

问的排序算法。

Bubble Sort - 冒泡排序

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

6 5 3 1 8 7 2 4

以上排序过程使用 Python 实现如下所示：

```
#!/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))
```

Reference

- [冒泡排序 - 维基百科，自由的百科全书](#)

Selection Sort - 选择排序

核心：不断地选择剩余元素中的最小者。

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

下图来源为 [File:Selection-Sort-Animation.gif - IB Computer Science](#)

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

以上排序过程使用 Python 实现如下所示：

```
#!/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))
```

性质：

- 比较次数 $= (N-1) + (N-2) + (N-3) + \dots + 2 + 1 \sim N^2/2$

- 交换次数=N
- 运行时间与输入无关
- 数据移动最少

Reference

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

Insertion Sort - 插入排序

核心：通过构建有序序列，对于未排序序列，从后向前扫描，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

6 5 3 1 8 7 2 4

以上排序过程使用 Python 实现如下所示：

```
#!/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))
```

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;
    }
}

```

性质：

- 交换操作和数组中导致的数量相同
- 比较次数 \geq 倒置数量， \leq 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 $\sim N^2/2$ 次比较和 $\sim N^2/2$ 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 $\sim N^2/4$ 次比较和 $\sim N^2/4$ 次交换

希尔排序

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

实现(C++):

```

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

```

Reference

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

Merge Sort - 归并排序

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

先来看看动图，归并排序是一种典型的分治应用。

6 5 3 1 8 7 2 4

```
#!/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.mergeSortedList(left, right)

    #@param A and B: sorted integer array A and B.
    #@return: A new sorted integer array
    def mergeSortedList(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

辅助函数：用于将已排序好的两个数组归并。

```
merge(Comparable[] a, int lo, int mid, int hi)
{    //将a[lo..mid] 和 a[mid+1..hi] 归并
    int i = lo, j = mid + 1;

    // 拷贝a[lo..hi]原数组至aux中
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    for (int k = lo; k <= hi; k++) {
        if (i > mid) { //左半边用尽，取右半边元素
            a[k] = aux[j++];
        } else if (j > hi) { //右半边用尽，取左半边元素
            a[k] = aux[i++];
        } else if (less(aux[j], aux[i])) { //右半边的当前元素小于左半边的当前元素，取右半边的元素
            a[k] = aux[j++];
        } else { //右半边的当前元素大于等于左半边的当前元素，取左半边的元素
            a[k] = aux[i++];
        }
    }
}
```

时间复杂度为 $N\log N$ ，但是空间复杂度为 N 。

Reference

- [Mergesort](#) - Robert Sedgewick 的大作，非常清晰。

Quick Sort - 快速排序

核心：快排是一种采用分治思想的排序算法，大致分为三个步骤。

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

先来一张动图看看快速排序的过程。

6 5 3 1 8 7 2 4

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

与归并排序的区别：

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

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

Reference

- [快速排序 - 维基百科，自由的百科全书](#)
- [Quicksort | Robert Sedgewick](#)

Heap Sort - 堆排序

特点：唯一能够同时最优地利用空间和时间的方法——最坏情况下也能保证使用 $2N\log N$ 次比较和恒定的额外空间。

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

Reference

- [堆排序 - 维基百科，自由的百科全书](#)
- [Priority Queues](#) - Robert Sedgewick 的大作，详解了关于堆的操作。
- [经典排序算法总结与实现 | Jark's Blog](#) - 堆排序讲的很好。

Bucket Sort

桶排序和归并排序有那么点点类似，也使用了归并的思想。大致步骤如下：

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

Reference

- [Bucket Sort Visualization](#) - 动态演示。
- [桶排序](#) - 维基百科，自由的百科全书

Counting Sort

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

维基上总结的四个步骤如下：

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

其中反向填充主要是为了避免重复元素落入新数组的同一索引处。

Reference

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

Basics Miscellaneous

杂项部分，涉及「位操作」等。

Bit Manipulation

位操作有按位与、或、非、左移n位和右移n位等操作。

XOR - 异或

异或：相同为0，不同为1。也可用「不进位加法」来理解。

异或操作的一些特点：

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

移位操作

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

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

Reference

- [位运算简介及实用技巧（一）：基础篇 | Matrix67: The Aha Moments](#)
- [cc150 chapter 8.5 and chapter 9.5](#)

Part II - Coding

本节主要总结一些leetcode等题目的实战经验。

主要有以下章节构成。

String - 字符串

strStr

Source

- leetcode: [Implement strStr\(\) | LeetCode OJ](#)
- lintcode: [lintcode - \(13\) strstr](#)

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

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

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

Example

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

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

Challenge

O(n) time.

Clarification

Do I need to implement KMP Algorithm in an interview?

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

题解

对于字符串查找问题，可使用双重for循环解决，效率更高的则为KMP算法。

Java

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

```

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

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

```

源码分析

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

Another Similar Question

```

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

```


Two Strings Are Anagrams

Source

- CC150: [\(158\) Two Strings Are Anagrams](#)

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

Example

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

Challenge

$O(n)$ time, $O(1)$ extra space

题解1 - hashmap 统计字频

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

C++

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

        int letterCount[256] = {0};

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



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

源码分析

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

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

复杂度分析

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

题解2 - 排序字符串

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

C++

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

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

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

源码分析

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

复杂度分析

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

Reference

- *CC150 Chapter 9.1* 中文版 p109

Compare Strings

Source

- lintcode: [\(55\) Compare Strings](#)

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

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

Example

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

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

题解

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

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

C++

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

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

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

源码解析

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

复杂度分析

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

Anagrams

Source

- leetcode: [Anagrams | LeetCode OJ](#)
- lintcode: [\(171\) Anagrams](#)

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

Example

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

Given ["ab", "ba", "cd", "dc", "e"], return ["ab", "ba", "cd", "dc"].

Note

All inputs will be in lower-case

题解1 - 双重 for 循环(TLE)

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

C++

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

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

```

        return result;
    }

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

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

        return true;
    }
};

```

源码分析

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

复杂度分析

私有方法 `isAnagrams` 最坏的时间复杂度为 $O(2L)$, 其中 L 为字符串长度。双重 `for` 循环时间复杂度近似为 $\frac{1}{2}O(n^2)$, n 为给定字符串数组数目。总的时间复杂度近似为 $O(n^2L)$ 。

题解2 - 排序 + hashmap

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

C++

```

class Solution {
public:
    /**
     * @param strs: A list of strings

```

```

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

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

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

        return result;
    }
};

```

源码分析

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

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

复杂度分析

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

Reference

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

Longest Common Substring

Source

- lintcode: [\(79\) Longest Common Substring](#)

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

Example

Given A="ABCD", B="CBCE", return 2.

Note

The characters in substring should occur continuously in original string.

This is different with subsequence.

题解

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

C++

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

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

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



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

源码分析

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

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

复杂度分析

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

Reference

- [Longest Common Substring | 九章算法](#)

Integer Array - 整型数组

本章主要总结与整型数组相关的题。

Remove Element

Source

- leetcode: [Remove Element | LeetCode OJ](#)
- lintcode: [\(172\) Remove Element](#)

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

没啥好分析的，遍历一次数组 $O(n)$ 。

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

同题解1， $O(n)$ 。

Reference

- [Remove Element | 九章算法](#)

Zero Sum Subarray

Source

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

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

Example

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

Note

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

题解1 - 两重 for 循环

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

题解2 - 比较子串和(TLE)

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

C++

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

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

```

        curr_sum += nums[i];

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

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

        sum_i.push_back(curr_sum);
    }

    return result;
}
};

```

源码分析

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

复杂度分析

最坏情况下 $O(n^2)$ ，实测和题解1中的方法运行时间几乎一致。

题解3 - 哈希表

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

C++

```

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

```

```

        hash[0] = 0;

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

        return result;
    }
};

```

源码分析

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

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

复杂度分析

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

题解4 - 排序

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

C++

```

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

```

```

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

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

return result;
}
};

```

源码分析

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

复杂度分析

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

扩展

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

另一道扩展题来自 Google 的面试题 - [Find subarray with given sum - GeeksforGeeks](#)。

Reference

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

Subarray Sum K

Source

- GeeksforGeeks: [Find subarray with given sum - GeeksforGeeks](#)

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

Example

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

题解1 - 哈希表

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

C++

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

using namespace std;

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

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

```

        hash[curr_sum] = i + 1;
    }
}

return result;
}
};

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

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

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

    return 0;
}

```

源码分析

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

复杂度分析

略，见 [Zero Sum Subarray | Data Structure and Algorithm](#)

题解2 - 利用单调函数特性

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

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

$f(i)$	1	5	25	28	38

<i>i</i>	0	1	2	3	4
----------	---	---	---	---	---

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

C++

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

using namespace std;

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

        int left_index = 0;
        int curr_sum = 0;
        for (int i = 0; i != nums.size(); ++i) {
            curr_sum += nums[i];
            if (curr_sum == k) {
                result.push_back(left_index);
                result.push_back(i);
                return result;
            }

            while (curr_sum > k) {
                curr_sum -= nums[left_index];
                ++left_index;
            }

            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 循环累加 curr_sum , 在 curr_sum > k 时再使用 while 递减 curr_sum , 同时递增左边索引 left_index .

复杂度分析

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

Reference

- [Find subarray with given sum - GeeksforGeeks](#)

Subarray Sum Closest

Source

- lintcode: [\(139\) Subarray Sum Closest](#)

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

Example

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

Challenge

$O(n \log n)$ time

题解

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

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

C++

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

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

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

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

```

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

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

```

源码分析

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

复杂度分析

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

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

扩展

- [algorithm - How to find the subarray that has sum closest to zero or a certain value t in \$O\(n \log n\)\$ - Stack Overflow](#)

Product of Array Exclude Itself

Source

- lintcode: [\(50\) Product of Array Exclude Itself](#)
- GeeksforGeeks: [A Product Array Puzzle - GeeksforGeeks](#)

Given an integers array A.

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

Example

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

题解1 - 左右分治

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

C++

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

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

        return result;
    }
};
```

源码分析

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

复杂度分析

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

题解2 - 原地求积

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

C++

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

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

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

        return result;
    }
};
```

源码分析

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

复杂度分析

时间复杂度同上，空间复杂度为 $O(1)$ 。

Partition Array

Source

- [\(31\) Partition Array](#)

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

All elements $< k$ are moved to the left

All elements $\geq k$ are moved to the right

Return the partitioning index, i.e the first index `i` `nums[i] $\geq k$` .

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 \geq 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 $i \geq right$ 不能是 $i > right$ ，否则需要对特殊情况如全小于 k 时的考虑，而且即使考虑了这一特殊情况也可能存在其他 bug。具体是什么 bug 呢？欢迎提出你的分析意见~

复杂度分析

遍历一次数组，时间复杂度最少为 $O(n)$ ，可能需要一定次数的交换。

题解2 - 两根指针

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

C++

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

        return left;
    }
};
```

源码分析

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

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

复杂度分析

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

Reference

- [Partition Array | 九章算法](#)

First Missing Positive

Source

- leetcode: [First Missing Positive | LeetCode OJ](#)
- lintcode: [\(189\) First Missing Positive](#)

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

Example

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

Challenge

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

题解

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

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

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

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

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

C++

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

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

```

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

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

    return size + 1;
}
};

```

源码分析

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

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

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

注意交换的写法，若写成

```

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

```

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

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

复杂度分析

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

Reference

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

2 Sum

Source

- leetcode: [Two Sum | LeetCode OJ](#)
- lintcode: [\(56\) 2 Sum](#)

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

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

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

Input: numbers={2, 7, 11, 15}, target=9

Output: index1=1, index2=2

题解1 - 哈希表

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

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

C++

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



```

    }

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

    return result;
}
};

```

源码分析

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

复杂度分析

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

Python

```

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

        return (-1, -1)

```

源码分析

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

题解2 - 排序后使用两根指针

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

C++

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

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

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

        return result;
    }
};
```

源码分析

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

5. 两根指针指头尾，逐步靠拢。

复杂度分析

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

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

3 Sum

Source

- leetcode: [3Sum | LeetCode OJ](#)
- lintcode: [\(57\) 3 Sum](#)

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

Example

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

```
(-1, 0, 1)
(-1, -1, 2)
```

Note

Elements in a triplet (a,b,c) must be in non-descending order. (ie, $a \leq b \leq c$)

The solution set must not contain duplicate triplets.

题解1 - 排序 + 哈希表 + 2 Sum

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

Python

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

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

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

    return triplets
```

源码分析

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

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

复杂度分析

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

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

Reference

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

3 Sum Closest

Source

- leetcode: [3Sum Closest | LeetCode OJ](#)
- lintcode: [\(59\) 3 Sum Closest](#)

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 Closest 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 Closest』问题。『2 Sum Closest』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 Closest』，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 Closest』进行优化，仍然利用有序数组的特点，若处于『一大一小』的临界值时就可以马上退出了，后面的元素与 `target` 之差的绝对值只会越来越大。

复杂度分析

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

Reference

- [3Sum Closest | 九章算法](#)

Remove Duplicates from Sorted Array

Source

- lintcode: [\(100\) Remove Duplicates from Sorted Array](#)

Given a sorted array, remove the duplicates in place such that each element appear only once.

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

使用双指针(下标)，一个指针(下标)遍历vector数组，另一个指针(下标)只取不重复的数置于原vector中。

```
class Solution {
public:
    /**
     * @param A: a list of integers
     * @return : return an integer
     */
    int removeDuplicates(vector<int> &nums) {
        if (nums.empty()) {
            return 0;
        }

        int size = 0;
        for (vector<int>::size_type i = 0; i != nums.size(); ++i) {
            if (nums[i] != nums[size]) {
                nums[++size] = nums[i];
            }
        }
        return ++size;
    }
};
```

源码分析

注意最后需要返回的是 `++size` 或者 `size + 1`

Remove Duplicates from Sorted Array II

Source

- lintcode: [\(101\) Remove Duplicates from Sorted Array II](#)

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

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

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

题解

在上题基础上加了限制条件元素最多可重复出现两次。因此可以在原题的基础上添加一变量跟踪元素重复出现的次数，小于指定值时执行赋值操作。但是需要注意的是重复出现次数 `occurrence` 的初始值(从1开始，而不是0)和reset的时机。

C++

```
class Solution {
public:
    /**
     * @param A: a list of integers
     * @return : return an integer
     */
    int removeDuplicates(vector<int> &nums) {
        if (nums.size() < 3) {
            return nums.size();
        }

        int size = 0;
        int occurrence = 1;
        for (vector<int>::size_type i = 1; i != nums.size(); ++i) {
            if (nums[size] != nums[i]) {
                nums[++size] = nums[i];
                occurrence = 1;
            } else if (nums[size] == nums[i]) {
                if (occurrence++ < 2) {
                    nums[++size] = nums[i];
                }
            }
        }

        return ++size;
    }
}
```

```
};
```

源码分析

1. 在数组元素小于3(即为2)时可直接返回vector数组大小。
2. 初始化 `occurence` 的值为1, 而不是0. 理解起来也方便些。
3. 初始化下标值 `i` 从1开始
 - `nums[size] != nums[i]` 时递增 `size` 并赋值, 同时重置 `occurence` 的值为1
 - `(nums[size] == nums[i])` 时, 首先判断 `occurence` 的值是否小于2, 小于2则先递增 `size`, 随后将 `nums[i]` 的值赋给 `nums[size]`。这里由于小标 `i` 从1开始, 免去了对 `i` 为0的特殊情况考虑。
4. 最后返回 `size + 1`, 即为 `++size`

Merge Sorted Array

Source

- leetcode: [Merge Sorted Array | LeetCode OJ](#)
- lintcode: [\(6\) Merge Sorted Array](#)

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

自尾部向首部逐个比较两个数组内的元素，取较大的置于新数组尾部元素中。

Python

```
class Solution:
    # @param {integer[]} nums1
    # @param {integer} m
    # @param {integer[]} nums2
    # @param {integer} n
    # @return {void} Do not return anything, modify nums1 in-place instead.
    def merge(self, nums1, m, nums2, n):
        # resize nums1 to required size
        nums1 += [0] * (n + m - len(nums1))
        index = m + n
        while m > 0 and n > 0:
            if nums1[m - 1] > nums2[n - 1]:
                index -= 1
                m -= 1
                nums1[index] = nums1[m]
            else:
                index -= 1
                n -= 1
                nums1[index] = nums2[n]

        while n > 0:
            index -= 1
            n -= 1
            nums1[index] = nums2[n]
```

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) {
        // write your code here
        int aLen = A.size();
        int bLen = B.size();
        ArrayList<Integer> res = new ArrayList<Integer>();

        int i = 0, j = 0;
        while (i < aLen || j < bLen) {
            if (i == aLen) {
                res.add(B.get(j++));
                continue;
            } else if (j == bLen) {
                res.add(A.get(i++));
                continue;
            }

            if (A.get(i) < B.get(j)) {
                res.add(A.get(i++));
            } else {
                res.add(B.get(j++));
            }
        }
        return res;
    }
}
```

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;
        }

        vector<int>::size_type sizeA = A.size();
        vector<int>::size_type sizeB = B.size();
        vector<int>::size_type sizeC = sizeA + sizeB;
```

```

vector<int> C(sizeC);

while (sizeA > 0 && sizeB > 0) {
    if (A[sizeA - 1] > B[sizeB - 1]) {
        C[--sizeC] = A[--sizeA];
    } else {
        C[--sizeC] = B[--sizeB];
    }
}
while (sizeA > 0) {
    C[--sizeC] = A[--sizeA];
}
while (sizeB > 0) {
    C[--sizeC] = B[--sizeB];
}

return C;
}
};

```

源码分析

分三种情况遍历比较。实际上在能确定最后返回的数组时，只需要分两次遍历即可。

复杂度分析

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

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。

Merge Sorted Array II

Source

- lintcode: [\(64\) Merge Sorted Array II](#)

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

Note

You may assume that A has enough space (size that is greater or equal to $m + n$) to hold all elements from both arrays.

Example

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

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

题解

在上题的基础上加入了in-place的限制。将上题的新数组视为length相对较大的数组即可，仍然从数组末尾进行归并，取出较大的元素。

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) {
        int index = m + n;

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

源码分析

1. 因为本题有了 in-place 的限制，则必须从数组末尾的两个元素开始比较；否则就会产生挪动，一旦挪动就会是 $O(n^2)$ 的。

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 = n + m;

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

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 $O(\lg n)$ 。
- 递归和迭代的使用场景：能用迭代就用迭代，特别复杂时采用递归。

Binary Search - 二分查找

Source

- lintcode: [lintcode - \(14\) Binary Search](#)

Binary search is a famous question in algorithm.

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

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

Example

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

Challenge

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

题解

对于已排序升序数组，使用二分查找可满足复杂度要求，注意数组中可能有重复值。

Java

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

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

```
        end = mid;
    }
}

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

return -1;
}
}
```

源码分析

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

Search Insert Position

Source

- lintcode: [\(60\) Search Insert Position](#)

Given a sorted array and a target value, return the index if the target is found. If not, You may assume no duplicates in the array.

Example

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

题解

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

Java

```
public class Solution {  
    /**  
     * param A : an integer sorted array  
     * param target : an integer to be inserted  
     * return : an integer  
     */  
    public int searchInsert(int[] A, int target) {  
        if (A == null) {  
            return -1;  
        }  
        if (A.length == 0) {  
            return 0;  
        }  
  
        int start = 0, end = A.length - 1;  
        int mid;  
  
        while (start + 1 < end) {  
            mid = start + (end - start) / 2;  
            if (A[mid] == target) {  
                return mid; // no duplicates, if not `end = target;`  
            } else if (A[mid] < target) {  
                start = mid;  
            } else {  
                end = mid;  
            }  
        }  
    }  
}
```

```
    }

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

源码分析

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

Search for a Range

Source

- lintcode: [\(61\) Search for a Range](#)

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

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

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

Example

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

题解

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

Java

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

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

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

```

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

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

    return result;
    // write your code here
}
}

```

源码分析

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

8. 两次二分查找中间不要忘记重置 `start, end` 的变量值。

First Bad Version

Source

- lintcode: [\(74\) First Bad Version](#)

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

Java:

```
public VersionControl {
    boolean isBadVersion(int version);
}
```

C++:

```
class VersionControl {
public:
    bool isBadVersion(int version);
};
```

Python:

```
class VersionControl:
    def isBadVersion(version)
```

Find the first bad version.

Note

You should call isBadVersion as few as possible.

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

Example

Given n=5

Call isBadVersion(3), get false

Call isBadVersion(5), get true

Call isBadVersion(4), get true

return 4 is the first bad version

Challenge

Do not call isBadVersion exceed $O(\log n)$ times.

题 Search for a Range 的变形，找出左边界即可。

Java

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



```

* you can use VersionControl.isBadVersion(k) to judge 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) {
        // write your code here
        if (n == 0) {
            return -1;
        }

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

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

```

C++

```

/**
 * class VersionControl {
 *     public:
 *     static bool isBadVersion(int k);
 * }
 * you can use VersionControl::isBadVersion(k) to judge 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) {
        if (n < 1) {
            return -1;
        }

        int start = 1;
        int end = n;

```

```

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

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

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

```

源码分析

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

Python

```

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

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

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

```

```
return -1
```

Search a 2D Matrix

Source

- lintcode: [\(28\) Search a 2D Matrix](#)

Write an efficient algorithm that searches for a value in an $m \times n$ matrix.

This matrix has the following properties:

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

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

Example

Consider the following matrix:

```
[
  [1, 3, 5, 7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given target = 3, return true.

Challenge

$O(\log(n) + \log(m))$ time

题解 - 一次二分搜索 V.S. 两次二分搜索

一次二分搜索

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

两次二分搜索

先按行再按列进行搜索，即两次二分搜索。时间复杂度相同。

以一次二分搜索的方法为例。

Java

```
/**
```

```

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

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

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

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

        return false;
    }
}

```

源码分析

仍然可以使用经典的二分搜索模板，注意下标的赋值即可。

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

第一次A掉这个题用的是分行分列两次搜索，好蠢...

Find Peak Element

Source

- leetcode: [Find Peak Element | LeetCode OJ](#)
- lintcode: [\(75\) Find Peak Element](#)

There is an integer array which has the following features:

* The numbers in adjacent positions are different.

* $A[0] < A[1]$ && $A[A.length - 2] > A[A.length - 1]$.

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

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

Note

The array may contains multiple peaks, find any of them.

Example

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

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

Challenge

Time complexity $O(\log N)$

题解1 - lintcode

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

备注：如果本题是找 first/last peak，就不能用二分法了。

Java

```
class Solution {
    /**
     * @param A: An integers array.
     * @return: return any of peak positions.
     */
    public int findPeak(int[] A) {
        // write your code here
        if (A == null) {
```

```

        return -1;
    }
    if (A.length == 0) {
        return 0;
    }

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

    mid = (A[start] > A[end]) ? start : end;
    return mid;
}
}

```

C++

```

class Solution {
public:
    /**
     * @param A: An integers array.
     * @return: return any of peak positions.
     */
    int findPeak(vector<int> A) {
        if (A.empty()) {
            return -1;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid = end / 2;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (A[mid] < A[mid - 1]) {
                end = mid;
            } else if (A[mid] < A[mid + 1]) {
                start = mid;
            } else {
                return mid;
            }
        }

        mid = (A[start] > A[end]) ? start : end;
        return mid;
    }
};

```

题解2 - leetcode

leetcode 上的题和 lintcode 上有细微的变化，题目如下：

A peak element is an element that is greater than its neighbors.

Given an input array where $\text{num}[i] \neq \text{num}[i+1]$,
find a peak element and return its index.

The array may contain multiple peaks,
in that case return the index to any one of the peaks is fine.

You may imagine that $\text{num}[-1] = \text{num}[n] = -\infty$.

For example, in array $[1, 2, 3, 1]$, 3 is a peak element and
your function should return the index number 2.

click to show spoilers.

Note:

Your solution should be in logarithmic complexity.

如果一开始做的是 leetcode 上的版本而不是 lintcode 上的话，这道题难度要大一些。有了以上的分析基础再来刷 leetcode 上的这道题就是小 case 了，注意题中的关键提示 $\text{num}[-1] = \text{num}[n] = -\infty$ ，虽然不像 lintcode 上那么直接，但是稍微变通下也能想到。即 $\text{num}[-1] < \text{num}[0]$ && $\text{num}[n-1] > \text{num}[n]$ ，那么问题来了，这样一来就不能确定峰值一定存在了，因为给定数组为单调序列的话就咻有峰值了，但是实际情况是——题中有负无穷的假设，也就是说在单调序列的情况下，峰值为数组首部或者尾部元素，谁大就是谁了。

Java1 - readily understood

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1, mid = end / 2;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (nums[mid] < nums[mid - 1]) {
                // 1 peak at least in the left side
                end = mid;
            } else if (nums[mid] < nums[mid + 1]) {
                // 1 peak at least in the right side
                start = mid;
            } else {
                return mid;
            }
        }

        mid = (nums[start] > nums[end]) ? start : end;
        return mid;
    }
}
```



```
    }
}
```

源码分析

典型的二分法模板应用，需要注意的是需要考虑单调序列的特殊情况。当然也可使用紧凑一点的实现如改写循环条件为 `start < end`，这样就不用考虑单调序列了，见实现2.

复杂度分析

二分法，时间复杂度 $O(\log n)$.

Java2 - compact implementation [leetcode_discussion](#)

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1, mid = end / 2;
        while (start < end) {
            if (nums[mid] < nums[mid + 1]) {
                // 1 peak at least in the right side
                start = mid + 1;
            } else {
                // 1 peak at least in the left side
                end = mid;
            }
            mid = start + (end - start) / 2;
        }

        return start;
    }
}
```

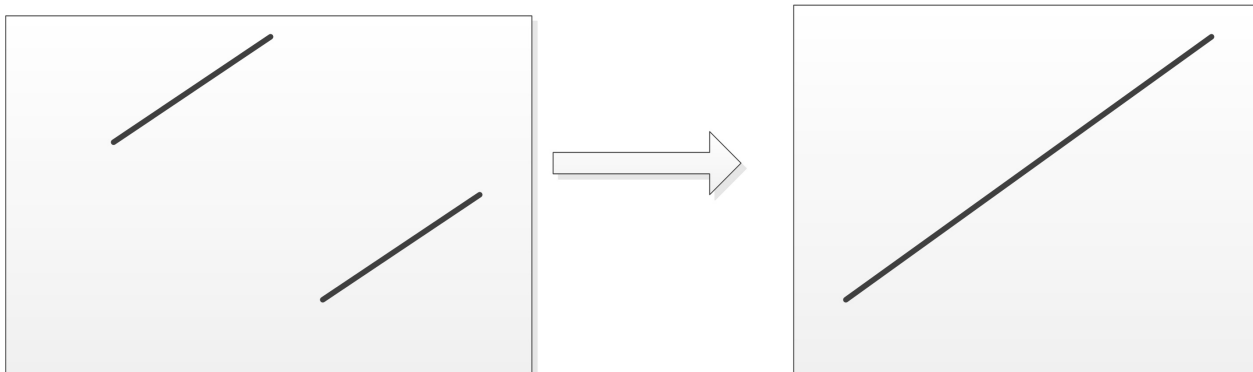
leetcode 和 lintcode 上给的方法名不一样，leetcode 上的为 `findPeakElement` 而 lintcode 上为 `findPeak`，弄混的话会编译错误。

Reference

- [leetcode_discussion](#). [Java - Binary-Search Solution - Leetcode Discuss](#) ↩

Search in Rotated Sorted Array

对于旋转数组的分析可使用画图的方法，如下图所示，升序数组经旋转后可能为如下两种形式。



Source

- lintcode: [\(62\) Search in Rotated Sorted Array](#)

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise

You may assume no duplicate exists in the array.

Example

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

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

题解

对于有序数组，使用二分搜索比较方便。分析题中的数组特点，旋转后初看是乱序数组，但仔细一看其实里面是存在两段有序数组的。因此该题可转化为如何找出旋转数组中的局部有序数组，并使用二分搜索解之。结合实际数组在纸上分析较为方便。

C++

```
/**
 * 本代码fork自
 * http://www.jiuzhang.com/solutions/search-in-rotated-sorted-array/
 */
class Solution {
/**
 * param A : an integer rotated sorted array
 * param target : an integer to be searched
 */
}
```

```

    * return : an integer
    */
public:
    int search(vector<int> &A, int target) {
        if (A.empty()) {
            return -1;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (target == A[mid]) {
                return mid;
            }
            if (A[start] < A[mid]) {
                // situation 1, numbers between start and mid are sorted
                if (A[start] <= target && target < A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else {
                // situation 2, numbers between mid and end are sorted
                if (A[mid] < target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }

        if (A[start] == target) {
            return start;
        }
        if (A[end] == target) {
            return end;
        }
        return -1;
    }
};

```

源码分析

1. 若 `target == A[mid]`，索引找到，直接返回
2. 寻找局部有序数组，分析 `A[mid]` 和两段有序的数组特点，由于旋转后前面有序数组最小值都比后面有序数组最大值大。故若 `A[start] < A[mid]` 成立，则 `start` 与 `mid` 间的元素必有序（要么是前一段有序数组，要么是后一段有序数组，还有可能是未旋转数组）。
3. 接着在有序数组 `A[start]~A[mid]` 间进行二分搜索，但能在 `A[start]~A[mid]` 间搜索的前提是 `A[start] <= target <= A[mid]`。
4. 接着在有序数组 `A[mid]~A[end]` 间进行二分搜索，注意前提条件。
5. 搜索完毕时索引若不是 `mid` 或者未满足 `while` 循环条件，则测试 `A[start]` 或者 `A[end]` 是否满足条件。
6. 最后若未找到满足条件的索引，则返回 -1。

Java

```
public class Solution {
    /**
     * @param A : an integer rotated sorted array
     * @param target : an integer to be searched
     * @return : an integer
     */
    public int search(int[] A, int target) {
        // write your code here
        if (A == null || A.length == 0) {
            return -1;
        }

        int start = 0, end = A.length - 1, mid = 0;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (A[mid] == target) {
                return mid;
            }
            if (A[start] < A[mid]) { // part 1
                if (A[start] <= target && target <= A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else { // part 2
                if (A[mid] <= target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }
        // end while

        if (A[start] == target) {
            return start;
        } else if (A[end] == target) {
            return end;
        } else {
            return -1; // not found
        }
    }
}
```

Search in Rotated Sorted Array II

Source

- lintcode: [\(63\) 搜索旋转排序数组 II](#)

跟进“搜索旋转排序数组”，假如有重复元素又将如何？

是否会影响运行时间复杂度？

如何影响？

为何会影响？

写出一个函数判断给定的目标值是否出现在数组中。

样例

给出[3,4,4,5,7,0,1,2]和target=4, 返回 true

题解

仔细分析此题和之前一题的不同之处，前一题我们利用 $A[start] < A[mid]$ 这一关键信息，而在此题中由于有重复元素的存在，在 $A[start] == A[mid]$ 时无法确定有序数组，此时只能依次递增start/递减end以缩小搜索范围，时间复杂度最差变为 $O(n)$ 。

C++

```
class Solution {
    /**
     * param A : an integer rotated sorted array and duplicates are allowed
     * param target : an integer to be search
     * return : a boolean
     */
public:
    bool search(vector<int> &A, int target) {
        if (A.empty()) {
            return false;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (target == A[mid]) {
                return true;
            }
            if (A[start] < A[mid]) {
                // situation 1, numbers between start and mid are sorted
                if (A[start] <= target && target < A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else if (A[start] > A[mid]) {
                // situation 2, numbers between mid and end are sorted
                if (A[mid] < target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }
    }
};
```

```
        } else {  
            // increment start  
            ++start;  
        }  
    }  
  
    if (A[start] == target || A[end] == target) {  
        return true;  
    }  
    return false;  
}  
};
```

源码分析

在 `A[start] == A[mid]` 时递增start序号即可。

Find Minimum in Rotated Sorted Array

Source

- lintcode: [\(159\) Find Minimum in Rotated Sorted Array](#)

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

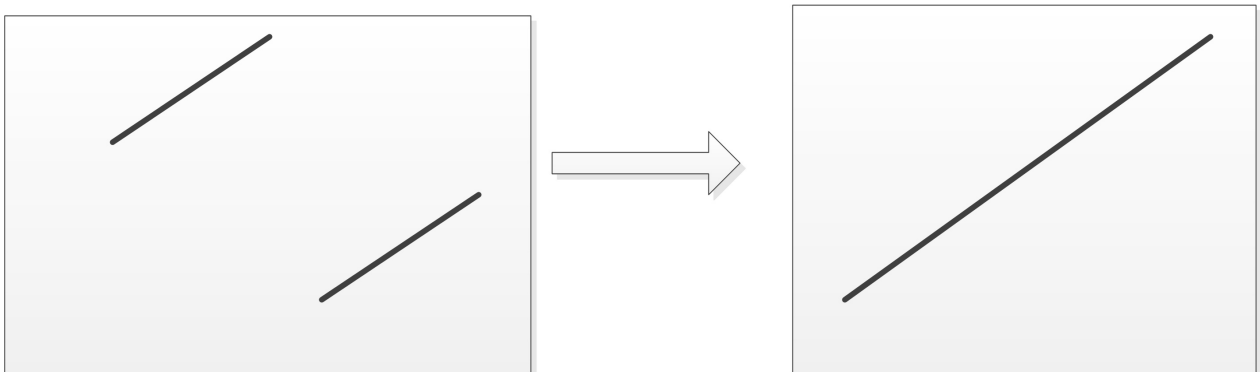
You may assume no duplicate exists in the array.

Example

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

题解

如前节所述，对于旋转数组的分析可使用画图的方法，如下图所示，升序数组经旋转后可能为如下两种形式。



最小值可能在上图中的两种位置出现，如果仍然使用数组首部元素作为target去比较，则需要考虑图中右侧情况。使用逆向思维分析，如果使用数组尾部元素分析，则无需图中右侧的特殊情况。

C++

```
class Solution {
public:
    /**
     * @param num: a rotated sorted array
     * @return: the minimum number in the array
     */
    int findMin(vector<int> &num) {
        if (num.empty()) {
            return -1;
        }
    }
};
```

```

vector<int>::size_type start = 0;
vector<int>::size_type end = num.size() - 1;
vector<int>::size_type mid;
while (start + 1 < end) {
    mid = start + (end - start) / 2;
    if (num[mid] < num[end]) {
        end = mid;
    } else {
        start = mid;
    }
}

if (num[start] < num[end]) {
    return num[start];
} else {
    return num[end];
}
};

```

源码分析

仅需注意使用 `num[end]` 作为判断依据即可，由于题中已给无重复数组的条件，故无需处理 `num[mid] == num[end]` 特殊条件。

Find Minimum in Rotated Sorted Array II

Source

- lintcode: [\(160\) Find Minimum in Rotated Sorted Array II](#)

题解

由于此题输入可能有重复元素，因此在 `num[mid] == num[end]` 时无法使用二分的方法缩小start或者end的取值范围。此时只能使用递增start/递减end逐步缩小范围。

C++

```

class Solution {
public:
    /**
     * @param num: a rotated sorted array
     * @return: the minimum number in the array
     */
    int findMin(vector<int> &num) {
        if (num.empty()) {
            return -1;
        }
    }
};

```



```
vector<int>::size_type start = 0;
vector<int>::size_type end = num.size() - 1;
vector<int>::size_type mid;
while (start + 1 < end) {
    mid = start + (end - start) / 2;
    if (num[mid] > num[end]) {
        start = mid;
    } else if (num[mid] < num[end]) {
        end = mid;
    } else {
        --end;
    }
}

if (num[start] < num[end]) {
    return num[start];
} else {
    return num[end];
}
};
```

Search a 2D Matrix II

Source

- lintcode: [\(38\) Search a 2D Matrix II](#)

Write an efficient algorithm that searches for a value in an $m \times n$ matrix, return the occurrence of the value.

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行或者一列元素（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 target: 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.length == 0) {
            return occurrence;
        }
        if (matrix[0] == null || matrix[0].length == 0) {
            return occurrence;
        }

        int row = matrix.length - 1;
        int column = matrix[0].length - 1;
        int index_row = 0, index_column = column;
        int number;

        if (target < matrix[0][0] || target > matrix[row][column]) {

```

```
        return occurence;
    }

    while (index_row < row + 1 && index_column + 1 > 0) {
        number = matrix[index_row][index_column];
        if (target == number) {
            occurence++;
            index_column--;
        } else if (target < number) {
            index_column--;
        } else if (target > number) {
            index_row++;
        }
    }

    return occurence;
}
}
```

源码分析

1. 首先对输入做异常处理，不仅要考虑到matrix为空串，还要考虑到matrix[0]也为空串。
2. 注意循环终止条件。
3. 在找出 target 后应继续向左搜索其他可能相等的元素，下方比当前元素大，故排除此列。

Reference

[Searching a 2D Sorted Matrix Part II | LeetCode](#)

Median of two Sorted Arrays

Source

- lintcode: [\(65\) Median of two Sorted Arrays](#)

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

Example

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

For A = [1,2,3] B = [4,5], the median is 3

Challenge

Time Complexity $O(\log n)$

题解

何谓"Median"? 由题目意思可得即为两个数组中一半数据比它大，另一半数据比它小的那个数。详见 [中位数 - 维基百科，自由的百科全书](#)，题中已有信息两个数组均为有序，题目要求时间复杂度为 $O(\log)$ ，因此应该往二分法上想。

在两个数组中找第k大数->找中位数即为找第k大数的一个特殊情况——第 $(A.length + B.length) / 2$ 大数。因此首先需要解决找第k大数的问题。这个联想确实有点牵强...

使用归并的思想逐个比较找出中位数的复杂度为 $O(n)$ ，显然不符要求，接下来考虑使用二分法。由于是找第k大数，使用二分法则比较 $A[k/2 - 1]$ 和 $B[k/2 - 1]$ ，并思考这两个元素和第k大元素的关系。

1. $A[k/2 - 1] \leq B[k/2 - 1] \Rightarrow$ A和B合并后的第k大数中必包含 $A[0] \sim A[k/2 - 1]$ ，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 $B[0] \sim B[k/2 - 1]$

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);
    } 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);
    }
}
};

```

源码分析

此题的边界条件较多，不容易直接从代码看清思路。首先分析找k大的辅助程序。

1. 如果 `A_start > A.size() - 1`，意味着A中无数提供，故仅能从B中取，所以只能是B从 `B_start` 开始的第k个数。下面的B...分析方法类似。

2. k为1时，无需再递归调用，直接返回较小值。
3. 以A为例，取出自 A_start 开始的第 $k / 2$ 个数，若下标 $A_start + k / 2 - 1 < A.size()$ ，则可取此下标对应的元素，否则置为int的最大值，便于后面进行比较，免去了诸多边界条件的判断。
4. 比较 $A_key > B_key$ ，取小的折半递归调用findKth。

接下来分析 findMedianSortedArrays：

1. 首先考虑异常情况，A, B都为空，A/B其中一个为空。
2. A+B 的长度为偶数时返回len / 2和 len / 2 + 1的均值，为奇数时则返回len / 2 + 1

Java

```
class Solution {
    /**
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    public double findMedianSortedArrays(int[] A, int[] B) {
        // write your code here
        int len = A.length + B.length;
        if (len % 2 == 0) {
            return (findKth(A, 0, B, 0, len/2) + findKth(A, 0, B, 0, len/2+1)) / 2.0;
        } else {
            return findKth(A, 0, B, 0, len/2 + 1);
        }
    }

    //find kth number of two sorted array
    public static int findKth(int[] A, int A_start, int[] B, int B_start, int k) {
        if (A_start >= A.length) {
            return B[B_start + k - 1];
        }
        if (B_start >= B.length) {
            return A[A_start + k - 1];
        }
        if (k == 1) {
            return Math.min(A[A_start], B[B_start]);
        }

        int A_key = (A_start + k/2 - 1 < A.length) // if one array is too short
            ? A[A_start + k/2 - 1] : Integer.MAX_VALUE; // trick
        int B_key = (B_start + k/2 - 1 < B.length) // if one array is too short
            ? B[B_start + k/2 - 1] : Integer.MAX_VALUE; // trick

        if (A_key < B_key) {
            return findKth(A, A_start + k/2, B, B_start, k - k/2);
        } else {
            return findKth(A, A_start, B, B_start + k/2, k - k/2);
        }
    }
}
```

源码分析

1. 本题用非递归的方法非常麻烦，递归的方法减少了很多边界的判断。
2. 递归的条件比较重要，可以用极端情况时参数的状况来入手，即看 `[]A`, `[]B`, `k` 谁先达到极端情况。
3. 本解法中有一个小技巧，就是当 `[]A`, `[]B` 中某一个数组太短了，无法取 $k/2$ ，则返回无穷大，设置了 `Integer.MAX_VALUE`。

reference

- [九章算法 | Median of Two Sorted Arrays](#)
- [LeetCode: Median of Two Sorted Arrays 解题报告 - Yu's Garden - 博客园](#)

Sqrt x

Source

- leetcode: [Sqrt\(x\) | LeetCode OJ](#)
- lintcode: [\(141\) Sqrt\(x\)](#)

题解 - 二分搜索

由于只要求整数部分，故对于任意正整数 x ，设其整数部分为 k ，显然有 $1 \leq k \leq x$ ，求解 k 的值也就转化为了在有序数组中查找满足某种约束条件的元素，显然二分搜索是解决此类问题的良方。

Python

```
class Solution:
    # @param {integer} x
    # @return {integer}
    def mySqrt(self, x):
        if x < 0:
            return -1
        elif x == 0:
            return 0

        start, end = 1, x
        while start + 1 < end:
            mid = start + (end - start) / 2
            if mid**2 == x:
                return mid
            elif mid**2 > x:
                end = mid
            else:
                start = mid

        return start
```

源码分析

1. 异常检测，先处理小于等于0的值。
2. 使用二分搜索的经典模板，注意不能使用 `start < end`，否则在给定值1时产生死循环。
3. 最后返回平方根的整数部分 `start`。

二分搜索过程很好理解，关键是最后的返回结果还需不需要判断？比如是取 `start`, `end`, 还是 `mid`? 我们首先来分析下二分搜索的循环条件，由 `while` 循环条件 `start + 1 < end` 可知，`start` 和 `end` 只可能有两种关系，一个是 `end == 1 || end == 2` 这一特殊情况，返回值均为1，另一个就是循环终止时 `start` 恰好在 `end` 前一个元素。设值 x 的整数部分为 k ，那么在执行二分搜索的过程中 $start \leq k \leq end$ 关系一直存在，也就是说在没有找到 $mid^2 == x$ 时，循环退出时有 $start < k < end$ ，取整的话显然就

是 `start` 了。

复杂度分析

经典的二分搜索，时间复杂度为 $O(\log n)$ ，使用了 `start`，`end`，`mid` 变量，空间复杂度为 $O(1)$ 。

除了使用二分法求平方根近似解之外，还可使用牛顿迭代法进一步提高运算效率，欲知后事如何，请猛戳[求平方根sqrt\(\)函数的底层算法效率问题 -- 简明现代魔法](#)，不得不感叹算法的魔力！

Wood Cut

Source

- lintcode: [\(183\) Wood Cut](#)

Given n pieces of wood with length $L[i]$ (integer array).
Cut them into small pieces to guarantee you could have equal or more than k pieces with the same length.
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

$O(n \log \text{Len})$, where Len is the longest length of the wood.

题解 - 二分搜索

这道题要直接想到二分搜索其实不容易，但是看到题中 Challenge 的提示后你大概就能想到往二分搜索上靠了。首先来分析下题意，题目意思是说给出 n 段木材 $L[i]$ ，将这 n 段木材切分为至少 k 段，这 k 段等长，求能从 n 段原材料中获得的最长单段木材长度。以 $k=7$ 为例，要将 L 中的原材料分为 7 段，能得到的最大单段长度为 114, $232/114 = 2$, $124/114 = 1$, $456/114 = 4$, $2 + 1 + 4 = 7$ 。

理清题意后我们就来想想如何用算法的形式表示出来，显然在计算如 2, 1, 4 等分片数时我们进行了取整运算，在计算机中则可以使用下式表示： $\sum_{i=1}^n \frac{L[i]}{l} \geq k$

其中 l 为单段最大长度，显然有 $1 \leq l \leq \max(L[i])$ 。单段长度最小为 1，最大不可能超过给定原材料中的最大木材长度。

注意求和与取整的顺序，是先求 $L[i]/l$ 的单个值，而不是先对 $L[i]$ 求和。

分析到这里就和题 [Sqrt x](#) 差不多一样了，要求的是 l 的最大可能取值，同时 l 可以看做是从有序序列 $[1, \max(L[i])]$ 的一个元素，典型的二分搜索！

Python

```
class Solution:
    """
    @param L: Given n pieces of wood with length L[i]
    @param k: An integer
    return: The maximum length of the small pieces.
    """
```

```
def woodCut(self, L, k):
    if sum(L) < k:
        return 0

    max_len = max(L)
    start, end = 1, max_len
    while start + 1 < end:
        mid = start + (end - start) / 2
        pieces_sum = sum([len_i / mid for len_i in L])
        if pieces_sum < k:
            end = mid
        else:
            start = mid

    # corner case
    if end == 2 and sum([len_i / 2 for len_i in L]) >= k:
        return 2

    return start
```

源码分析

1. 异常处理，若对 L 求和所得长度都小于 k，那么肯定无解。
2. 初始化 start 和 end，使用二分搜索。
3. 使用 list comprehension 求 $\sum_{i=1}^n \frac{L[i]}{l}$ 。
4. 若求得的 pieces_sum 小于 k，则说明 mid 偏大，下一次循环应缩小 mid，对应为将当前 mid 赋给 end。
5. 与一般的二分搜索不同，即使有 pieces_sum == k 也不应立即返回 mid，因为这里使用了取整运算，满足 pieces_sum == k 的值不止一个，应取其中最大的 mid，具体实现中可以将 pieces_sum < k 写在前面，大于等于的情况直接用 start = end 代替。
6. 排除 end == 2 之后返回 start 即可。

简单对第6条做一些说明，首先需要进行二分搜索的前提是 `sum(L) >= k` 且 end 不满足 `end == 1 || end == 2`，end 为2时单独考虑即可。

复杂度分析

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

Reference

- [Wood Cut | 九章算法](#)

Bit Manipulation

位运算的题大多较为灵活，涉及较多的按位与/或/异或等特性。

Reference

- [位运算简介及实用技巧（一）：基础篇 | Matrix67: The Aha Moments](#)
- *cc150* chapter 8.5 and chapter 9.5

Single Number

「找单数」系列题，技巧性较强，需要灵活运用位运算的特性。

Source

- lintcode: [\(82\) Single Number](#)

Given $2*n + 1$ numbers, every numbers occurs twice except one, find it.

Example

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

Challenge

One-pass, constant extra space

题解

根据题意，共有 $2*n + 1$ 个数，且有且仅有一个数落单，要找出相应的「单数」。鉴于有空间复杂度的要求，不可能使用另外一个数组来保存每个数出现的次数，考虑到异或运算的特性，根据 $x \oplus x = 0$ 和 $x \oplus 0 = x$ 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 $x \wedge 0 = x$

Single Number II

Source

- lintcode: [\(83\) Single Number II](#)

Given $3 \cdot n + 1$ 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 extra 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 $3 \cdot n + 1$ ，因此我们很自然地想到如果有种运算能满足「三三运算」为0该有多好！对于三个相同的数来说，其相加的和必然是3的倍数，仅仅使用这一个特性还不足以将单数找出来，我们再来挖掘隐含的信息。以3为例，若使用不进位加法，三个3相加的结果为：

```
0011
0011
0011
----
0033
```

注意到其中的奥义了么？三个相同的数相加，不仅其和能被3整除，其二进制位上的每一位也能被3整除！因此我们只需要一个和 `int` 类型相同大小的数组记录每一位累加的结果即可。时间复杂度约为 $O((3n + 1) \cdot \text{sizeof}(\text{int}) \cdot 8)$

C++ bit by bit

```
class Solution {
public:
    /**
     * @param A : An integer array
     * @return : An integer
     */
    int singleNumberII(vector<int> &A) {
        if (A.empty()) {
            return 0;
        }

        int result = 0, bit_i_sum = 0;

        for (int i = 0; i != 8 * sizeof(int); ++i) {
```



```

        bit_i_sum = 0;
        for (int j = 0; j != A.size(); ++j) {
            // get the *i*th bit of A
            bit_i_sum += ((A[j] >> i) & 1);
        }
        // set the *i*th bit of result
        result |= ((bit_i_sum % 3) << i);
    }

    return result;
}
};

```

源码解析

1. 异常处理
2. 循环处理返回结果 `result` 的 `int` 类型的每一位，要么自增1，要么保持原值。注意 `i` 最大可取 $8 \cdot \text{sizeof}(\text{int}) - 1$, 字节数=>位数的转换
3. 对第 `i` 位处理完的结果模3后更新 `result` 的第 `i` 位，由于 `result` 初始化为0，故使用或操作即可完成

Reference

[Single Number II - Leetcode Discuss](#) 中抛出了这么一道扩展题：

Given an array of integers, every element appears `k` times except for one. Find that single

@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 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[1];
}
```

Single Number III

O(1) Check Power of 2

Source

- lintcode: [\(142\) O\(1\) Check Power of 2](#)

Using $O(1)$ time to check whether an integer n is a power of 2.

Example

For $n=4$, return true;

For $n=5$, return false;

Challenge

$O(1)$ time

题解

乍看起来挺简单的一道题目，可之前若是没有接触过神奇的位运算技巧遇到这种题就有点不知从哪入手了，咳咳，我第一次接触到这个题就是在七牛的笔试题中看到的，泪奔 :-)

简单点来考虑可以连除2求余，看最后的余数是否为1，但是这种方法无法在 $O(1)$ 的时间内解出，所以我们必须要想点别的办法了。2的整数幂若用二进制来表示，则其中必只有一个1，其余全是0，那么怎么才能用一个式子把这种特殊的关系表示出来了？传统的位运算如按位与、按位或和按位异或等均无法直接求解，我就不卖关子了，比较下 $x - 1$ 和 x 的关系试试？以 $x=4$ 为例。

```
0100 ==> 4
0011 ==> 3
```

两个数进行按位与就为0了！如果不是2的整数幂则无上述关系，反证法可证之。

Python

```
class Solution:
    """
    @param n: An integer
    @return: True or false
    """
    def checkPowerOf2(self, n):
        if n < 1:
            return False
        else:
            return (n & (n - 1)) == 0
```

C++

```

class Solution {
public:
    /*
     * @param n: An integer
     * @return: True or false
     */
    bool checkPowerOf2(int n) {
        if (1 > n) {
            return false;
        } else {
            return 0 == (n & (n - 1));
        }
    }
};

```

Java

```

class Solution {
    /*
     * @param n: An integer
     * @return: True or false
     */
    public boolean checkPowerOf2(int n) {
        if (n < 1) {
            return false;
        } else {
            return (n & (n - 1)) == 0;
        }
    }
};

```

源码分析

除了考虑正整数之外，其他边界条件如小于等于0的整数也应考虑在内。在比较0和 $(n \& (n - 1))$ 的值时，需要用括号括起来避免优先级结合的问题。

复杂度分析

$O(1)$.

扩展

关于2的整数幂还有一道有意思的题，比如 [Next Power of 2 - GeeksforGeeks](#)，有兴趣的可以去围观下。

Convert Integer A to Integer B

Source

- CC150, lintcode: [\(181\) Convert Integer A to Integer B](#)

Determine the number of bits required to convert integer A to integer B

Example

Given n = 31, m = 14, return 2

(31)₁₀=(11111)₂

(14)₁₀=(01110)₂

题解

比较两个数不同的比特位个数，显然容易想到可以使用异或处理两个整数，相同的位上为0，不同的位上为1，故接下来只需将异或后1的个数求出即可。容易想到的方法是移位后和1按位与得到最低位的结果，使用计数器记录这一结果，直至最后操作数为0时返回最终值。这种方法需要遍历元素的每一位，有咩有更为高效的做法呢？还记得之前做过的 [O1 Check Power of 2](#) 吗？ $x \& (x - 1)$ 既然可以检查2的整数次幂，那么如何才能进一步得到所有1的个数呢？——将异或得到的数分拆为若干个2的整数次幂，计算得到有多少个2的整数次幂即可。

以上的分析过程对于正数来说是毫无问题的，但问题就在于如果出现了负数如何破？不确定的时候就来个实例测测看，以-2为例， $(-2) \& (-2 - 1)$ 的计算如下所示(简单起见这里以8位为准)：

```

11111110 <==> -2    -2 <==> 11111110
+
11111111 <==> -1    -3 <==> 11111101
=
11111101              11111100

```

细心的你也许发现了对于负数来说，其表现也是我们需要的—— $x \& (x - 1)$ 的含义即为将二进制比特位的值为1的最低位置零。逐步迭代直至最终值为0时返回。

C/C++ 和 Java 中左溢出时会直接将高位丢弃，正好方便了我们的计算，但是在 Python 中就没这么幸运了，因为溢出时会自动转换类型，Orz... 所以使用 Python 时需要对负数专门处理，转换为求其补数中0的个数。

Python

```

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，故处理负数时需要补数中0的个数，间接求得原异或得到的数中1的个数。

考虑到负数的可能，C/C++, Java 中循环终止条件为 `a_xor_b != 0`，而不是 `a_xor_b > 0`。

复杂度分析

取决于异或后数中1的个数，`O(max(ones in a ^ b))`。

关于 Python 中位运算的一些坑总结在参考链接中。

Reference

- [BitManipulation - Python Wiki](#)
- [5. Expressions — Python 2.7.10rc0 documentation](#)
- [Python之位移操作符所带来的困惑 - 旁观者 - 博客园](#)

Factorial Trailing Zeroes

Source

- leetcode: [Factorial Trailing Zeroes | LeetCode OJ](#)
- lintcode: [\(2\) Trailing Zeros](#)

Write an algorithm which computes the number of trailing zeros in n factorial.

Example

11! = 39916800, so the out should be 2

Challenge

$O(\log N)$ time

题解1 - Iterative

找阶乘数中末尾的连零数量，容易想到的是找相乘能为10的整数倍的数，如 2×5 , 1×10 等，遥想当初做阿里笔试题时遇到过类似的题，当时想着算算5和10的个数就好了，可万万没想到啊，25可以变为两个5相乘！真是蠢死了... 根据数论里面的知识，任何正整数都可以表示为它的质因数的乘积 [wikipedia](#)。所以比较准确的思路应该是计算质因数5和2的个数，取小的即可。质因数2的个数显然要大于5的个数，故只需要计算给定阶乘数中质因数中5的个数即可。原题的问题即转化为求阶乘数中质因数5的个数，首先可以试着分析下100以内的数，再试试100以上的数，聪明的你一定想到了可以使用求余求模等方法：)

Python

```
class Solution:
    # @param {integer} n
    # @return {integer}
    def trailingZeroes(self, n):
        if n < 0:
            return -1

        count = 0
        while n > 0:
            n /= 5
            count += n

        return count
```

C++

```
class Solution {
public:
    int trailingZeroes(int n) {
        if (n < 0) {
```

```

        return -1;
    }

    int count = 0;
    for (; n > 0; n /= 5) {
        count += (n / 5);
    }

    return count;
}
};

```

Java

```

public class Solution {
    public int trailingZeroes(int n) {
        if (n < 0) {
            return -1;
        }

        int count = 0;
        for (; n > 0; n /= 5) {
            count += (n / 5);
        }

        return count;
    }
}

```

源码分析

1. 异常处理，小于0的数返回-1.
2. 先计算5的正整数幂都有哪些，不断使用 $n/5$ 即可知质因数5的个数。
3. 在循环时使用 $n /= 5$ 而不是 $i *= 5$ ，可有效防止溢出。

lintcode 和 leetcode 上的方法名不一样，在两个 OJ 上分别提交的时候稍微注意下。

复杂度分析

关键在于 $n /= 5$ 执行的次数，时间复杂度 $\log_5 n$ ，使用了 `count` 作为返回值，空间复杂度 $O(1)$ 。

题解2 - Recursive

可以使用迭代处理的程序往往用递归，而且往往更为优雅。递归的终止条件为 $n \leq 0$ 。

Python

```

class Solution:
    # @param {integer} n

```

```
# @return {integer}
def trailingZeroes(self, n):
    if n == 0:
        return 0
    elif n < 0:
        return -1
    else:
        return n / 5 + self.trailingZeroes(n / 5)
```

C++

```
class Solution {
public:
    int trailingZeroes(int n) {
        if (n == 0) {
            return 0;
        } else if (n < 0) {
            return -1;
        } else {
            return n / 5 + trailingZeroes(n / 5);
        }
    }
};
```

Java

```
public class Solution {
    public int trailingZeroes(int n) {
        if (n == 0) {
            return 0;
        } else if (n < 0) {
            return -1;
        } else {
            return n / 5 + trailingZeroes(n / 5);
        }
    }
}
```


源码分析

这里将负数输入视为异常，返回-1而不是0。注意使用递归时务必注意收敛和终止条件的返回值。这里递归层数最多不超过 $\log_5 n$ ，因此效率还是比较高的。

复杂度分析

递归层数最大为 $\log_5 n$ ，返回值均在栈上，可以认为没有使用辅助的堆空间。

Reference

-  [Prime factor - Wikipedia, the free encyclopedia](#) ↵
- [Count trailing zeroes in factorial of a number - GeeksforGeeks](#)

Linked List - 链表

本节包含链表的一些常用操作，如删除、插入和合并等。

常见错误有 遍历链表不向前递推节点，遍历链表前未保存头节点，返回链表节点指针错误。

Remove Duplicates from Sorted List

Source

- lintcode: [\(112\) Remove Duplicates from Sorted List](#)

Given a sorted linked list, delete all duplicates such that each element appear only once

Example

Given 1->1->2, return 1->2.

Given 1->1->2->3->3, return 1->2->3.

题解

遍历之，遇到当前节点和下一节点的值相同时，删除下一节点，改变当前节点next的指针值。

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) {
            if (node->val == node->next->val) {
                ListNode *temp = node->next;
                node->next = node->next->next;
                delete temp;
            } 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`，便于后面进行delete
3. 不相等时往后指针往后遍历。

Java

```

/**
 * http://www.jiuzhang.com/solutions/remove-duplicates-from-sorted-list/
 */

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;
    }
}

```

Java版有个好处：不用自己管理内存，故不需要进行delete操作。

Remove Duplicates from Sorted List II

Source

- lintcode: [\(113\) Remove Duplicates from Sorted List II](#)

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct nodes. The list should be sorted after it.

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`，因为 `dummy->val` 有可能与第一个节点的值相等。故在判断`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;

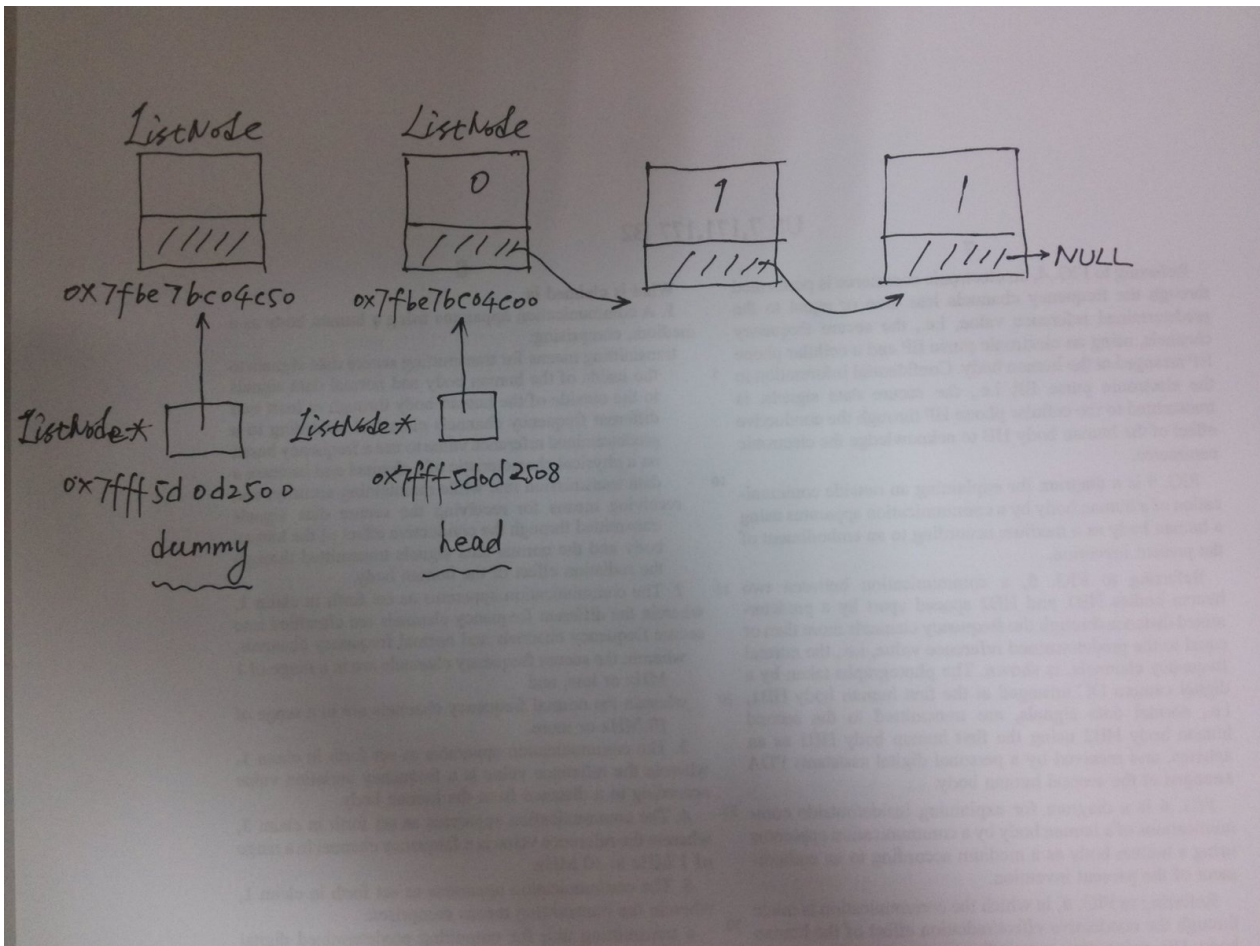
        while (node->next != NULL && node->next->next != NULL) {
            if (node->next->val == node->next->next->val) {
                int val = node->next->val;
                while (node->next != NULL && val == node->next->val) {
                    ListNode *temp = node->next;
                    node->next = 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 0x7fff5d0d2500，其保存着指针变量值为 0x7fbe7bc04c50。head 指针变量本身的地址为 0x7fff5d0d2508，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。

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 || head->next == 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 = node->next->val;
            while (node->next != NULL && val == node->next->val) {
                ListNode *temp = node->next;
                node->next = node->next->next;
                delete temp;
            }
        } else {
            node = node->next;
        }
    }

    return dummy->next;
}
};

```

源码分析

1. 首先考虑异常情况，head和head->next均考虑可减少后面的麻烦。
2. new一个dummy变量，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。

Partition List

Source

- lintcode: [\(96\) Partition List](#)

Given a linked list and a value x , partition it such that all nodes less than x come before all 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 $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 2 \rightarrow \text{null}$ and $x = 3$,
return $1 \rightarrow 2 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow \text{null}$.

题解

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，乍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。由于头节点不确定，我们可以使用dummy节点这个大杀器。

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 x: an integer
     * @return: a ListNode
     */
    ListNode *partition(ListNode *head, int x) {
        if (NULL == head) {
            return NULL;
        }
    }
}
```

```
ListNode *leftDummy = new ListNode(0);
ListNode *rightDummy = new ListNode(0);
ListNode *left = leftDummy;
ListNode *right = rightDummy;

while (head != NULL) {
    if (head->val < x) {
        left->next = head;
        left = head;
    } else {
        right->next = head;
        right = head;
    }
    head = head->next;
}

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

Two Lists Sum

Source

- CC150 - [\(167\) 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 $O(\max(L1, L2))$ 。

C++ - Recursion

除了使用迭代，对于链表类问题也比较适合使用递归实现。

To-be done.

Reference

- CC150 Chapter 9.2 题2.5，中文版 p123
- [Add two numbers represented by linked lists | Set 1 - GeeksforGeeks](#)

Two Lists Sum Advanced

Source

- CC150 - [Add two numbers represented by linked lists | Set 2 - GeeksforGeeks](#)

Given two numbers represented by two linked lists, write a function that returns sum list. The sum list is linked list representation of addition of two input numbers.

Example

Input:

First List: 5->6->3 // represents number 563

Second List: 8->4->2 // represents number 842

Output

Resultant list: 1->4->0->5 // represents number 1405

Challenge

Not allowed to modify the lists.

Not allowed to use explicit extra space.

题解1 - 反转链表

在题 [Two Lists Sum | Data Structure and Algorithm](#) 的基础上改了下数位的表示方式，前者低位在前，高位在后，这个题的高位在前，低位在后。很自然地可以联想到先将链表反转，而后再使用 Two Lists Sum 的解法。

Reference

- [Add two numbers represented by linked lists | Set 2 - GeeksforGeeks](#)

Remove Nth Node From End of List

Source

- lintcode: [\(174\) Remove Nth Node From End of List](#)

Given a linked list, remove the nth node from the end of list and return its head.

Note

The minimum number of nodes in list is n.

Example

Given linked list: 1->2->3->4->5->null, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5->null.

Challenge

O(n) time

题解

简单题，使用快慢指针解决此题，需要注意最后删除的是否为头节点。让快指针先走 n 步，直至快指针走到终点，找到需要删除节点之前的一个节点，改变 node->next 域即可。

C++

```
/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @param n: An integer.
     * @return: The head of linked list.
     */
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        if (NULL == head || n < 0) {
            return NULL;
        }
    }
}
```

```

        ListNode *preN = head;
        ListNode *tail = head;
        // slow fast pointer
        int index = 0;
        while (index < n) {
            if (NULL == tail) {
                return NULL;
            }
            tail = tail->next;
            ++index;
        }

        if (NULL == tail) {
            return head->next;
        }

        while (tail->next) {
            tail = tail->next;
            preN = preN->next;
        }
        preN->next = preN->next->next;

        return head;
    }
};

```

以上代码单独判断了是否需要删除头节点的情况，在遇到头节点不确定的情况下，引入 `dummy` 节点将会使代码更加优雅，改进的代码如下。

C++ dummy node

```

/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @param n: An integer.
     * @return: The head of linked list.
     */
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        if (NULL == head || n < 1) {
            return NULL;
        }

        ListNode *dummy = new ListNode(0);

```

```
dummy->next = head;
ListNode *preDel = dummy;

for (int i = 0; i != n; ++i) {
    if (NULL == head) {
        return NULL;
    }
    head = head->next;
}

while (head) {
    head = head->next;
    preDel = preDel->next;
}
preDel->next = preDel->next->next;

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

源码分析

引入 `dummy` 节点后画个图分析下就能确定 `head` 和 `preDel` 的转移关系了。

Linked List Cycle

Source

- leetcode: [Linked List Cycle | LeetCode OJ](#)
- lintcode: [\(102\) Linked List Cycle](#)

Given a linked list, determine if it has a cycle in it.

Example

Given -21->10->4->5, tail connects to node index 1, return true

Challenge

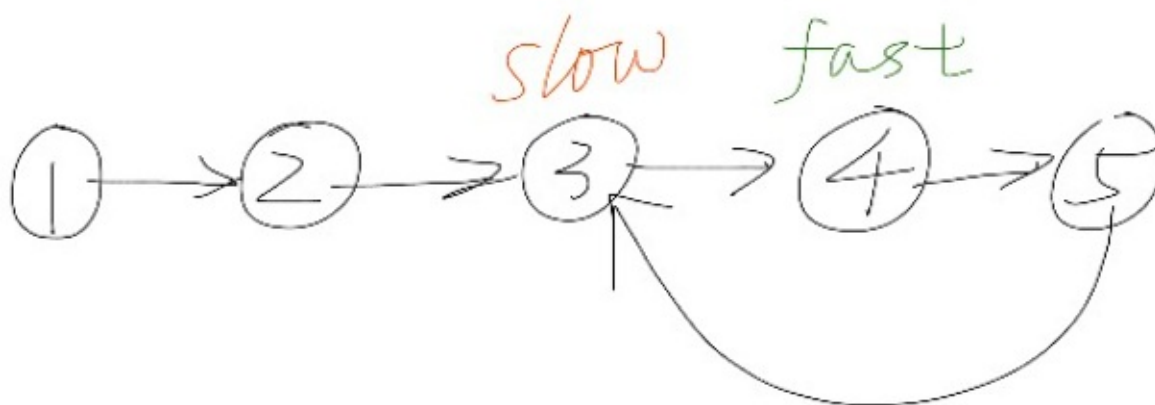
Follow up:

Can you solve it without using extra space?

题解 - 快慢指针

对于带环链表的检测，效率较高且易于实现的一种方式为使用快慢指针。快指针每次走两步，慢指针每次走一步，如果快慢指针相遇(快慢指针所指内存为同一区域)则有环，否则快指针会一直走到 NULL 为止退出循环，返回 false。

快指针走到 NULL 退出循环即可确定此链表一定无环这个很好理解。那么带环的链表快慢指针一定会相遇吗？先来看看下图。



在有环的情况下，最终快慢指针一定都走在环内，加入第 i 次遍历时快指针还需要 k 步才能追上慢指针，由于快指针比慢指针每次多走一步。那么每遍历一次快慢指针间的间距都会减少1，直至最终相遇。故快慢指针相遇一定能确定该链表有环。

C++

```
/**
```

```

* Definition of ListNode
* class ListNode {
* public:
*     int val;
*     ListNode *next;
*     ListNode(int val) {
*         this->val = val;
*         this->next = NULL;
*     }
* }
*/
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: True if it has a cycle, or false
     */
    bool hasCycle(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return false;
        }

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

        return false;
    }
};

```

源码分析

1. 异常处理，将 `head->next` 也考虑在内有助于简化后面的代码。
2. 慢指针初始化为 `head`，快指针初始化为 `head` 的下一个节点，这是快慢指针初始化的一种方法，有时会简化边界处理，但有时会增加麻烦，比如该题的进阶版。

复杂度分析

1. 在无环时，快指针每次走两步走到尾部节点，遍历的时间复杂度为 $O(n/2)$ 。
2. 有环时，最坏的时间复杂度近似为 $O(n)$ 。最坏情况下链表的头尾相接，此时快指针恰好在慢指针前一个节点，还需 n 次快慢指针相遇。最好情况和无环相同，尾节点出现环。

故总的时间复杂度可近似为 $O(n)$ 。

Reference

- [Linked List Cycle | 九章算法](#)

Linked List Cycle II

Source

- leetcode: [Linked List Cycle II | LeetCode OJ](#)
- lintcode: [\(103\) Linked List Cycle II](#)

Given a linked list, return the node where the cycle begins. If there is no cycle, return

Example

Given -21->10->4->5, tail connects to node index 1, return node 10

Challenge

Follow up:

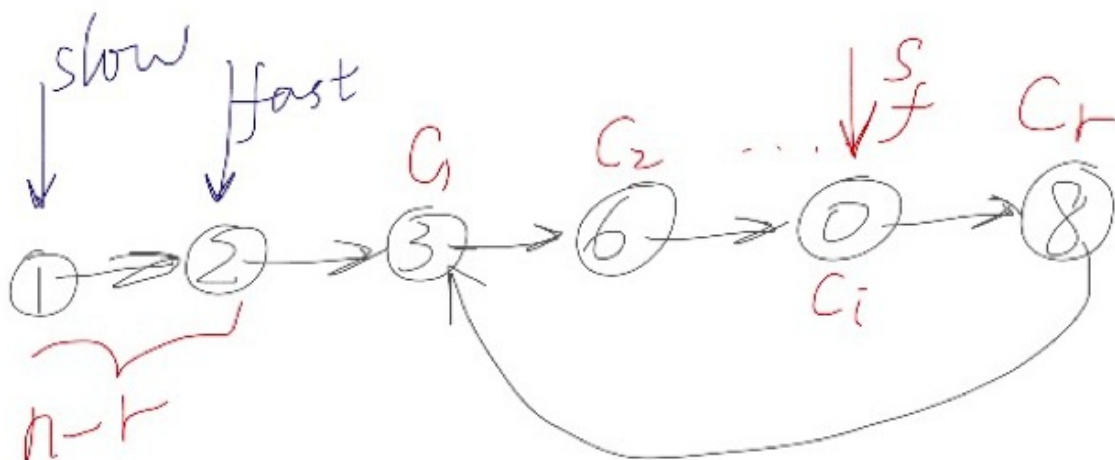
Can you solve it without using extra space?

题解 - 快慢指针

题 [Linked List Cycle | Data Structure and Algorithm](#) 的升级版，题目要求不适用额外空间，则必然还是使用快慢指针解决问题。首先设组成环的节点个数为 r ，链表中节点个数为 n 。首先我们来分析下在链表有环时都能推出哪些特性：

1. 快慢指针第一次相遇时快指针比慢指针多走整数个环，这个容易理解，相遇问题。
2. 每次相遇都在同一个节点。第一次相遇至第二次相遇，快指针需要比慢指针多走一个环的节点个数，而快指针比慢指针多走的步数正好是慢指针自身移动的步数，故慢指针恰好走了一圈回到原点。

从以上两个容易得到的特性可知，在仅仅知道第一次相遇时的节点还不够，相遇后如果不改变既有策略则必然找不到环的入口。接下来我们分析下如何从第一次相遇的节点走到环的入口节点。还是让我们先从实际例子出发，以下图为例。



slow 和 fast 节点分别初始化为节点 1 和 2，假设快慢指针第一次相遇的节点为 0，对应于环中的第 i 个节点 C_i ，那么此时慢指针正好走了 $n - r - 1 + i$ 步，快指针则走了 $2 \cdot (n - r - 1 + i)$ 步，且存在¹: $n - r - 1 + i + 1 = l \cdot r$. (之所以在 i 后面加1是因为快指针初始化时多走了一步) 快慢指针第一次相遇时慢指针肯定没有走完整个环，且慢指针走的步数即为整数个环节点数，由性质1和性质2可联合推出。

现在分析下相遇的节点和环的入口节点之间的关联，要从环中第 i 个节点走到环的入口节点，则按照顺时针方向移动²: $(l \cdot r - i + 1)$ 个节点 (l 为某个非负整数) 即可到达。现在来看看式¹和式²间的关系。由式¹可以推知 $n - r = l \cdot r - i$. 从头节点走到环的入口节点所走的步数可用 $n - r$ 表示，故在快慢指针第一次相遇时让另一节点从头节点出发，慢指针仍从当前位置迭代，第二次相遇时的位置即为环的入口节点！

由于此题快指针初始化为头节点的下一个节点，故分析起来稍微麻烦些，且在第一次相遇后需要让慢指针先走一步，否则会出现死循环。

对于该题来说，快慢指针都初始化为头节点会方便很多，故以下代码使用头节点对快慢指针进行初始化。

C++

```
/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: The node where the cycle begins.
     *         if there is no cycle, return null
     */
    ListNode *detectCycle(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return NULL;
        }

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

```
    }  
    return NULL;  
}  
};
```

源码分析

1. 异常处理。
2. 找第一次相遇的节点。
3. 将 `fast` 置为头节点，并只走一步，直至快慢指针第二次相遇，返回慢指针所指的节点。

复杂度分析

第一次相遇的最坏时间复杂度为 $O(n)$, 第二次相遇的最坏时间复杂度为 $O(n)$. 故总的时间复杂度近似为 $O(n)$, 空间复杂度 $O(1)$.

Reference

- [Linked List Cycle II | 九章算法](#)

Reverse Linked List- 链表翻转

Source

- lintcode: [\(35\) Reverse Linked List](#)

Reverse a linked list.

Example

For linked list 1->2->3, the reversed linked list is 3->2->1

Challenge

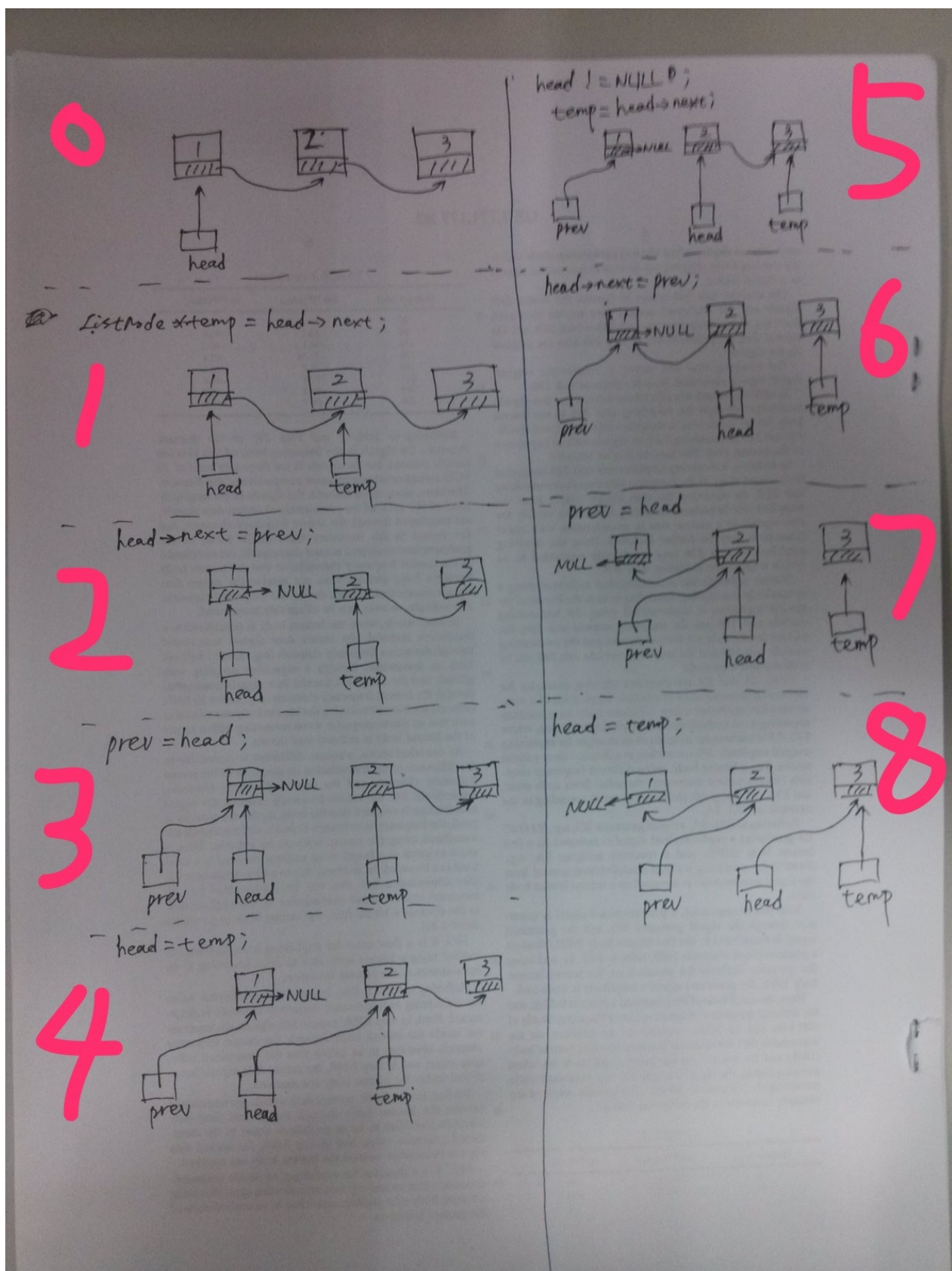
Reverse it in-place and in one-pass

题解

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 . 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

C++

```

/**
 * http://www.jiuzhang.com/solutions/reverse-linked-list/
 * 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) {
        ListNode *prev = NULL;
        while (head) {
            ListNode *temp = head->next;
            head->next = prev;
            prev = head;
            head = temp;
        }

        return prev;
    }
};

```

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。

Reference

1. [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](#)

Reverse Linked List II

Source

- lintcode: [\(36\) Reverse Linked List II](#)

Reverse a linked list from position m to n.

Note

Given m, n satisfy the following condition: $1 \leq m \leq n \leq \text{length of list}$.

Example

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

Challenge

Reverse it in-place and in one-pass

题解

此题在上题的基础上加了位置要求，只翻转指定区域的链表。由于链表头节点不确定，祭出我们的dummy杀器。此题边界条件处理特别tricky，需要特别注意。

1. 由于只翻转指定区域，分析受影响的区域为第m-1个和第n+1个节点
2. 找到第m个节点，使用for循环n-m次，使用上题中的链表翻转方法
3. 处理第m-1个和第n+1个节点
4. 返回dummy->next

C++

```
/**
 * Definition of singly-linked-list:
 *
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The head of linked list.
     * @param m: The start position need to reverse.
     * @param n: The end position need to reverse.
     * @return: The new head of partial reversed linked list.
     */
}
```

```

*/
ListNode *reverseBetween(ListNode *head, int m, int n) {
    if (head == NULL || m > n) {
        return NULL;
    }

    ListNode *dummy = new ListNode(0);
    dummy->next = head;
    ListNode *node = dummy;

    for (int i = 1; i != m; ++i) {
        if (node == NULL) {
            return NULL;
        } else {
            node = node->next;
        }
    }

    ListNode *premNode = node;
    ListNode *mNode = node->next;
    ListNode *nNode = mNode, *postnNode = nNode->next;
    for (int i = m; i != n; ++i) {
        if (postnNode == NULL) {
            return NULL;
        }

        ListNode *temp = postnNode->next;
        postnNode->next = nNode;
        nNode = postnNode;
        postnNode = temp;
    }
    premNode->next = nNode;
    mNode->next = postnNode;

    return dummy->next;
}
};

```

源码分析

1. 处理异常
2. 使用dummy辅助节点
3. 找到premNode——m节点之前的一个节点
4. 以nNode和postnNode进行遍历翻转，注意考虑在遍历到n之前postnNode可能为空
5. 连接premNode和nNode, `premNode->next = nNode;`
6. 连接mNode和postnNode, `mNode->next = postnNode;`

务必注意**node** 和**node->next**的区别！！，node指代节点，而 `node->next` 指代节点的下一连接。

Merge Two Sorted Lists

Source

- lintcode: [\(165\) Merge Two Sorted Lists](#)
- leetcode: [Merge Two Sorted Lists | LeetCode OJ](#)

Merge two sorted linked lists and return it as a new list.
The new list should be made by splicing together the nodes of the first two lists.

Example

Given 1->3->8->11->15->null, 2->null, return 1->2->3->8->11->15->null

题解

此题为两个链表的合并，合并后的表头节点不一定，故应联想到使用 dummy 节点。链表节点的插入主要涉及节点 next 指针值的改变，两个链表的合并操作则涉及到两个节点的 next 值变化，若每次合并一个节点都要改变两个节点 next 的值且要对 NULL 指针做异常处理，势必会异常麻烦。嗯，第一次做这个题时我就是这么想的... 下面看看相对较好的思路。

首先 dummy 节点还是必须要用到，除了 dummy 节点外还引入一个 lastNode 节点充当下一次合并时的头节点。在 l1 或者 l2 的某一个节点为空指针 NULL 时，退出 while 循环，并将非空链表的头部链接到 lastNode->next 中。

C++

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *lastNode = dummy;
        while ((NULL != l1) && (NULL != l2)) {
            if (l1->val < l2->val) {
                lastNode->next = l1;
                l1 = l1->next;
            } else {
                lastNode->next = l2;
                l2 = l2->next;
            }
        }
    }
};
```

```

        lastNode = lastNode->next;
    }

    // do not forget this line!
    lastNode->next = (NULL != l1) ? l1 : l2;

    return dummy->next;
}
};

```

源码分析

1. 异常处理，包含在 `dummy->next` 中。
2. 引入 `dummy` 和 `lastNode` 节点，此时 `lastNode` 指向的节点为 `dummy`
3. 对非空 `l1, l2` 循环处理，将 `l1/l2` 的较小者链接到 `lastNode->next`，往后递推 `lastNode`
4. 最后处理 `l1/l2` 中某一链表为空退出 `while` 循环，将非空链表头链接到 `lastNode->next`
5. 返回 `dummy->next`，即最终的首指针

注意 `lastNode` 的递推并不影响 `dummy->next` 的值，因为 `lastNode` 和 `dummy` 是两个不同的指针变量。

链表的合并为常用操作，务必非常熟练，以上的模板非常精炼，有两个地方需要记牢。1. 循环结束条件中为条件与操作；2. 最后处理 `lastNode->next` 指针的值。

复杂度分析

最好情况下，一个链表为空，时间复杂度为 $O(1)$ 。最坏情况下，`lastNode` 遍历两个链表中的每一个节点，时间复杂度为 $O(l1 + l2)$ 。空间复杂度近似为 $O(1)$ 。

Reference

- [Merge Two Sorted Lists | 九章算法](#)

Merge k Sorted Lists

Source

- leetcode: [Merge k Sorted Lists | LeetCode OJ](#)
- lintcode: [\(104\) Merge k Sorted Lists](#)

题解1 - 选择归并(TLE)

参考 [Merge Two Sorted Lists | Data Structure and Algorithm](#) 中对两个有序链表的合并方法，这里我们也可以采用从 k 个链表中选择其中最小值的节点链接到 `lastNode->next` (和选择排序思路有点类似)，同时该节点所在的链表表头节点往后递推一个。直至 `lastNode` 遍历完 k 个链表的所有节点，此时表头节点均为 `NULL`，返回 `dummy->next`。

这种方法非常简单直接，但是时间复杂度较高，容易出现 TLE。

C++

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

class Solution {
public:
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.empty()) {
            return NULL;
        }

        ListNode *dummy = new ListNode(INT_MAX);
        ListNode *last = dummy;

        while (true) {
            int count = 0;
            int index = -1, tempVal = INT_MAX;
            for (int i = 0; i != lists.size(); ++i) {
                if (NULL == lists[i]) {

```



```

        ++count;
        if (count == lists.size()) {
            last->next = NULL;
            return dummy->next;
        }
        continue;
    }

    // choose the min value in non-NULL ListNode
    if (NULL != lists[i] && lists[i]->val <= tempVal) {
        tempVal = lists[i]->val;
        index = i;
    }
}

last->next = lists[index];
last = last->next;
lists[index] = lists[index]->next;
}
}
};

```

源码分析

1. 由于头节点不定，我们使用 dummy 节点。
2. 使用 last 表示每次归并后的新链表末尾节点。
3. count 用于累计链表表头节点为 NULL 的个数，若与 vector 大小相同则代表所有节点均已遍历完。
4. tempVal 用于保存每次比较 vector 中各链表表头节点中的最小值，index 保存本轮选择归并过程中最小值对应的链表索引，用于循环结束前递推该链表表头节点。

复杂度分析

由于每次 for 循环只能选择出一个最小值，总的时间复杂度最坏情况下为 $O(k \cdot \sum_{i=1}^k l_i)$ 。空间复杂度近似为 $O(1)$ 。

题解2 - 迭代调用 Merge Two Sorted Lists (TLE)

鉴于题解1时间复杂度较高，题解2中我们可以反复利用时间复杂度相对较低的 [Merge Two Sorted Lists | Data Structure and Algorithm](#)。即先合并链表1和2，接着将合并后的新链表再与链表3合并，如此反复直至 vector 内所有链表均已完全合并 [soulmachine](#)。

C++

```

/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {

```

```

*         this->val = val;
*         this->next = NULL;
*     }
* }
*/

class Solution {
public:
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.empty()) {
            return NULL;
        }

        ListNode *head = lists[0];
        for (int i = 1; i != lists.size(); ++i) {
            head = merge2Lists(head, lists[i]);
        }

        return head;
    }

private:
    ListNode *merge2Lists(ListNode *left, ListNode *right) {
        ListNode *dummy = new ListNode(0);
        ListNode *last = dummy;

        while (NULL != left && NULL != right) {
            if (left->val < right->val) {
                last->next = left;
                left = left->next;
            } else {
                last->next = right;
                right = right->next;
            }
            last = last->next;
        }

        last->next = (NULL != left) ? left : right;

        return dummy->next;
    }
};

```

源码分析

实现合并两个链表的子方法后就没啥难度了，`mergeKLists` 中左半部分链表初始化为 `lists[0]`，`for` 循环后迭代归并 `head` 和 `lists[i]`。

复杂度分析

合并两个链表时最差时间复杂度为 $O(l_1 + l_2)$ ，那么在以上的实现中总的时间复杂度可近似认为是

$l_1 + l_1 + l_2 + \dots + l_1 + l_2 + \dots + l_k = O(\sum_{i=1}^k (k-i) \cdot l_i)$. 比起题解1复杂度是要小一点，但量级上仍然差太多。实际运行时间也证明了这一点，题解2的运行时间差不多时题解1的一半。那么还有没有进一步降低时间复杂度的可能呢？当然是有的，且看下题分解...

题解3 - 二分调用 Merge Two Sorted Lists

题解2中 merge2Lists 优化空间不大，那咱们就来看看 mergeKLists 中的 for 循环，仔细观察可得知第 i 个链表 l_i 被遍历了 $k-i$ 次，如果我们使用二分法对其进行归并呢？从中间索引处进行二分归并后，每个链表参与合并的次数变为 $\log k$ ，故总的时间复杂度可降至 $\log k \cdot \sum_{i=1}^k l_i$ 。优化幅度较大。

C++

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

class Solution {
public:
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.empty()) {
            return NULL;
        }

        return helper(lists, 0, lists.size() - 1);
    }

private:
    ListNode *helper(vector<ListNode *> &lists, int start, int end) {
        if (start == end) {
            return lists[start];
        } else if (start + 1 == end) {
            return merge2Lists(lists[start], lists[end]);
        }

        ListNode *left = helper(lists, start, start + (end - start) / 2);
        ListNode *right = helper(lists, start + (end - start) / 2 + 1, end);

        return merge2Lists(left, right);
    }
}
```

```

ListNode *merge2Lists(ListNode *left, ListNode *right) {
    ListNode *dummy = new ListNode(0);
    ListNode *last = dummy;

    while (NULL != left && NULL != right) {
        if (left->val < right->val) {
            last->next = left;
            left = left->next;
        } else {
            last->next = right;
            right = right->next;
        }
        last = last->next;
    }
    last->next = (NULL != left) ? left : right;

    return dummy->next;
}
};

```

源码分析

由于需要建立二分递归模型，另建一私有方法 `helper` 引入起止位置较为方便。下面着重分析 `helper`。

1. 分两种边界条件处理，分别是 `start == end` 和 `start + 1 == end`。虽然第二种边界条件可以略去，但是加上会节省递归调用的栈空间。
2. 使用分治思想理解 `helper`，`left` 和 `right` 的边界处理建议先分析几个简单例子，做到不重不漏。
3. 注意 `merge2Lists` 中传入的参数，为 `lists[start]` 而不是 `start` ...

在 `mergeKLists` 中调用 `helper` 时传入的 `end` 参数为 `lists.size() - 1`，而不是 `lists.size()`。

复杂度分析

题解中已分析过，最坏的时间复杂度为 $\log k \cdot \sum_{i=1}^k l_i$ ，空间复杂度近似为 $O(1)$ 。

优化后的运行时间显著减少！由题解2中的500+ms 减至40ms 以内。

Reference

- [soulmachine](#). [soulmachine的LeetCode 题解](#) ←

Sort List

Source

- leetcode: [Sort List | LeetCode OJ](#)
- lintcode: [\(98\) Sort List](#)

Sort a linked list in $O(n \log n)$ time using constant space complexity.

题解1 - 归并排序(链表长度求中间节点)

链表的排序操作，对于常用的排序算法，能达到 $O(n \log n)$ 的复杂度有快速排序(平均情况)，归并排序，堆排序。快速排序不一定能保证其时间复杂度一定满足要求，归并排序和堆排序都能满足复杂度的要求。在数组排序中，归并排序通常需要使用 $O(n)$ 的额外空间，也有原地归并的实现，代码写起来略微麻烦一点。但是对于链表这种非随机访问数据结构，所谓的「排序」不过是指针 next 值的变化而已，主要通过指针操作，故仅需要常数级别的额外空间，满足题意。堆排序通常需要构建二叉树，在这道题中不太适合。

既然确定使用归并排序，我们就来思考归并排序实现的几个要素。

1. 按长度等分链表，归并虽然不严格要求等分，但是等分能保证线性对数的时间复杂度。由于链表不能随机访问，故可以先对链表进行遍历求得其长度。
2. 合并链表，细节已在 [Merge Two Sorted Lists | Data Structure and Algorithm](#) 中详述。

在按长度等分链表时进行「后序归并」——先求得左半部分链表的表头，再求得右半部分链表的表头，最后进行归并操作。

由于递归等分链表的操作需要传入链表长度信息，故需要另建一辅助函数。新鲜出炉的源码如下。

```
/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: You should return the head of the sorted linked list,
     *         using constant space complexity.
     */
    ListNode *sortList(ListNode *head) {
```

```

    if (NULL == head) {
        return NULL;
    }

    // get the length of List
    int len = 0;
    ListNode *node = head;
    while (NULL != node) {
        node = node->next;
        ++len;
    }

    return sortListHelper(head, len);
}

private:
ListNode *sortListHelper(ListNode *head, const int length) {
    if ((NULL == head) || (0 >= length)) {
        return head;
    }

    ListNode *midNode = head;

    int count = 1;
    while (count < length / 2) {
        midNode = midNode->next;
        ++count;
    }

    ListNode *rList = sortListHelper(midNode->next, length - length / 2);
    midNode->next = NULL;
    ListNode *lList = sortListHelper(head, length / 2);

    return mergeList(lList, rList);
}

ListNode *mergeList(ListNode *l1, ListNode *l2) {
    ListNode *dummy = new ListNode(0);
    ListNode *lastNode = dummy;
    while ((NULL != l1) && (NULL != l2)) {
        if (l1->val < l2->val) {
            lastNode->next = l1;
            l1 = l1->next;
        } else {
            lastNode->next = l2;
            l2 = l2->next;
        }

        lastNode = lastNode->next;
    }

    lastNode->next = (NULL != l1) ? l1 : l2;

    return dummy->next;
}
};

```

源码分析

1. 归并子程序没啥好说的了，见 [Merge Two Sorted Lists | Data Structure and Algorithm](#).
2. 在递归处理链表长度时，分析方法和 [Convert Sorted List to Binary Search Tree | Data Structure and Algorithm](#) 一致，`count` 表示遍历到链表中间时表头指针需要移动的节点数。在纸上分析几个简单例子后即可确定，由于这个题需要的是「左右」而不是二叉搜索树那道题需要三分——「左中右」，故将 `count` 初始化为1更为方便，左半部分链表长度为 $\text{length} / 2$ ，这两个值的确定最好是先用纸笔分析再视情况取初值，不可死记硬背。
3. 找到中间节点后首先将其作为右半部分链表处理，然后将其 `next` 值置为 `NULL`，否则归并子程序无法正确求解。这里需要注意的是 `midNode` 是左半部分的最后一个节点，`midNode->next` 才是链表右半部分的起始节点。
4. 递归模型中左、右、合并三者的顺序可以根据分治思想确定，即先找出左右链表，最后进行归并(因为归并排序的前提是两个子链表各自有序)。

复杂度分析

遍历求得链表长度，时间复杂度为 $O(n)$ ，「折半取中」过程中总共有 $\log(n)$ 层，每层找中点需遍历 $n/2$ 个节点，故总的时间复杂度为 $n/2 \cdot O(\log n)$ (折半取中)，每一层归并排序的时间复杂度介于 $O(n/2)$ 和 $O(n)$ 之间，故总的时间复杂度为 $O(n \log n)$ ，空间复杂度为常数级别，满足题意。

题解2 - 归并排序(快慢指针求中间节点)

除了遍历链表求得总长外，还可使用看起来较为巧妙的技巧如「快慢指针」，快指针每次走两步，慢指针每次走一步，最后慢指针所指的节点即为中间节点。使用这种特技的关键之处在于如何正确确定快慢指针的起始位置。

C++

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

```

    }

    ListNode *midNode = findMiddle(head);
    ListNode *rList = sortList(midNode->next);
    midNode->next = NULL;
    ListNode *lList = sortList(head);

    return mergeList(lList, rList);
}

private:
    ListNode *findMiddle(ListNode *head) {
        if (NULL == head || NULL == head->next) {
            return head;
        }

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

        return slow;
    }

    ListNode *mergeList(ListNode *l1, ListNode *l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *lastNode = dummy;
        while ((NULL != l1) && (NULL != l2)) {
            if (l1->val < l2->val) {
                lastNode->next = l1;
                l1 = l1->next;
            } else {
                lastNode->next = l2;
                l2 = l2->next;
            }

            lastNode = lastNode->next;
        }

        lastNode->next = (NULL != l1) ? l1 : l2;

        return dummy->next;
    }
};

```

源码分析

1. 异常处理不仅考虑了 `head`，还考虑了 `head->next`，可减少辅助程序中的异常处理。
2. 使用快慢指针求中间节点时，将 `fast` 初始化为 `head->next` 可有效避免无法分割两个节点如 `1->2->null` [fast_slow_pointer](#)。
 - 求中点的子程序也可不做异常处理，但前提是主程序 `sortList` 中对 `head->next` 做了检测。
3. 最后进行 `merge` 归并排序。

在递归和迭代程序中，需要尤其注意终止条件的确定，以及循环语句中变量的自增，以防出现死循环

或访问空指针。

复杂度分析

同上。

Reference

- [Sort List | 九章算法](#)
- [fast_slow_pointer](#), [LeetCode: Sort List 解题报告 - Yu's Garden - 博客园](#) ↩

Reorder List

Source

- leetcode: [Reorder List | LeetCode OJ](#)
- lintcode: [\(99\) Reorder List](#)

Given a singly linked list L: $L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$,
reorder it to: $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

You must do this in-place without altering the nodes' values.

Example

For example,

Given 1->2->3->4->null, reorder it to 1->4->2->3->null.

题解1 - 链表长度(TLE)

直观角度来考虑，如果把链表视为数组来处理，那么我们要做的就是依次将下标之和为 n 的两个节点链接到一块儿，使用两个索引即可解决问题，一个索引指向 i ，另一个索引则指向其之后的第 $n - 2*i$ 个节点（对于链表来说实际上需要获取的是其前一个节点），直至第一个索引大于第二个索引为止即处理完毕。

既然依赖链表长度信息，那么要做的第一件事就是遍历当前链表获得其长度喽。获得长度后即对链表进行遍历，小心处理链表节点的断开及链接。用这种方法会提示 TLE，也就是说还存在较大的优化空间！

C++ - TLE

```
/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: void
     */
    void reorderList(ListNode *head) {
        if (NULL == head || NULL == head->next || NULL == head->next->next) {
            return;
        }
    }
};
```

```

    }

    ListNode *last = head;
    int length = 0;
    while (NULL != last) {
        last = last->next;
        ++length;
    }

    last = head;
    for (int i = 1; i < length - i; ++i) {
        ListNode *beforeTail = last;
        for (int j = i; j < length - i; ++j) {
            beforeTail = beforeTail->next;

            ListNode *temp = last->next;
            last->next = beforeTail->next;
            last->next->next = temp;
            beforeTail->next = NULL;
            last = temp;
        }
    }
};

```

源码分析

1. 异常处理，对于节点数目在两个以内的无需处理。
2. 遍历求得链表长度。
3. 遍历链表，第一个索引处的节点使用 `last` 表示，第二个索引处的节点的前一个节点使用 `beforeTail` 表示。
4. 处理链表的链接与断开，迭代处理下一个 `last`。

复杂度分析

1. 遍历整个链表获得其长度，时间复杂度为 $O(n)$ 。
2. 双重 `for` 循环的时间复杂度为 $(n-2) + (n-4) + \dots + 2 = O(\frac{1}{2} \cdot n^2)$ 。
3. 总的时间复杂度可近似认为是 $O(n^2)$ ，空间复杂度为常数。

使用这种方法务必注意 `i` 和 `j` 的终止条件，若取 `i < length + 1 - i`，则在处理最后两个节点时会出现环，且尾节点会被删掉。在对节点进行遍历时务必注意保留头节点的信息！

题解2 - 反转链表后归并

既然题解1存在较大的优化空间，那我们该从哪一点出发进行优化呢？擒贼先擒王，题解1中时间复杂度最高的地方在于双重 `for` 循环，在对第二个索引进行遍历时，`j` 每次都从 `i` 处开始遍历，要是 `j` 能从链表尾部往前遍历该有多好啊！这样就能大大降低时间复杂度了，可惜本题的链表只是单向链表... 有什么特技可以在单向链表中进行反向遍历吗？还真有——反转链表！一语惊醒梦中人。

C++

```

/**
 * Definition of ListNode
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: void
     */
    void reorderList(ListNode *head) {
        if (NULL == head || NULL == head->next || NULL == head->next->next) {
            return;
        }

        ListNode *middle = findMiddle(head);
        ListNode *right = reverse(middle->next);
        middle->next = NULL;

        merge(head, right);
    }

private:
    void merge(ListNode *left, ListNode *right) {
        ListNode *dummy = new ListNode(0);
        while (NULL != left && NULL != right) {
            dummy->next = left;
            left = left->next;
            dummy = dummy->next;
            dummy->next = right;
            right = right->next;
            dummy = dummy->next;
        }

        dummy->next = (NULL != left) ? left : right;
        //delete dummy; /* bug, delete the tail node */
    }

    ListNode *reverse(ListNode *head) {
        ListNode *newHead = NULL;
        while (NULL != head) {
            ListNode *temp = head->next;
            head->next = newHead;
            newHead = head;
            head = temp;
        }

        return newHead;
    }
}

```

```

ListNode *findMiddle(ListNode *head) {
    if (NULL == head || NULL == head->next) {
        return head;
    }

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

    return slow;
}
};

```

源码分析

相对于题解1，题解2更多地利用了链表的常用操作如反转、找中点、合并。

1. 找中点：我在九章算法模板的基础上增加了对 `head->next` 的异常检测，增强了鲁棒性。
2. 反转：非常精炼的模板，记牢！
3. 合并：也可使用九章提供的模板，思想是一样的，需要注意 `left`，`right` 和 `dummy` 三者的赋值顺序，不能更改任何一步。
4. 对于 `new` 出的内存如何释放？代码中注释掉的为错误方法，你知道为什么吗？

复杂度分析

找中点一次，时间复杂度近似为 $O(n)$ 。反转链表一次，时间复杂度近似为 $O(n/2)$ 。合并左右链表一次，时间复杂度近似为 $O(n/2)$ 。故总的时间复杂度为 $O(n)$ 。

Reference

- [Reorder List | 九章算法](#)

Reverse - 翻转法

本章主要介绍『三步翻转法』的使用。

Recover Rotated Sorted Array

Source

- lintcode: [\(39\) Recover Rotated Sorted Array](#)

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

Example

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

Challenge

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

Clarification

What is rotated array:

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

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

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

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

Java

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

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

```

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

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

```

C++

```

/**
 * forked from
 * http://www.jiuzhang.com/solutions/recover-rotated-sorted-array/
 */
class Solution {
private:
    void reverse(vector<int> &nums, vector<int>::size_type start, vector<int>::size_type
        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:
    vector<int> 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`

Rotate String

Source

- lintcode: [\(8\) Rotate String](#)

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

Example

Given "abcdefg"

for offset=0, return "abcdefg"

for offset=1, return "gabcdef"

for offset=2, return "fgabcde"

for offset=3, return "efgabcd"

...

题解

常见的翻转法应用题，仔细观察规律可知翻转的分割点在从数组末尾数起的offset位置。

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()) {
            return A;
        }

        string::size_type sizeA = A.size();
        offset %= sizeA;
        if (offset == 0) {
            return A;
        }

        reverse(A, 0, sizeA - offset - 1);
        reverse(A, sizeA - offset, sizeA - 1);
        reverse(A, 0, sizeA - 1);

        return A;
    }
};
```

```
    }

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

源码分析

1. 异常处理，A为空或者offset模sizeA后为0
2. offset可能超出A的大小，应模sizeA后再用
3. 三步翻转法

Reverse Words in a String

Source

- lintcode: [\(53\) Reverse Words in a String](#)

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

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

Example
Clarification

- What constitutes a word?
A sequence of non-space characters constitutes a word.
- Could the input string contain leading or trailing spaces?
Yes. However, your reversed string should not contain leading or trailing spaces.
- How about multiple spaces between two words?
Reduce them to a single space in the reversed string.

题解

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

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

C++

```
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

Maximum Depth of Binary Tree# Binary Tree - 二叉树

二叉树的基本概念在 [Binary Tree | Algorithm](#) 中有简要的介绍，这里就二叉树的一些应用做一些实战演练。

二叉树的遍历大致可分为前序、中序、后序三种方法。

Binary Tree Preorder Traversal - 前序遍历

Source

- lintcode: [\(66\) Binary Tree Preorder Traversal](#)

Given a binary tree, return the preorder traversal of its nodes' values.

Note

Given binary tree {1,#,2,3},

```
  1
   \
    2
   /
  3
```

return [1,2,3].

Example

Challenge

Can you do it without recursion?

题解 - 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

```
"""
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++ Traverse - 递归遍历

```

/**
 * 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> val_vec;
        traverse(root, val_vec);

        return val_vec;
    }

private:
    void traverse(TreeNode *root, vector<int> &ret) {
        if (root == NULL) {
            return;
        }

        ret.push_back(root->val);
        traverse(root->left, ret);
        traverse(root->right, ret);
    }
};

```

源码分析

使用了辅助递归函数 `traverse`，传值时注意应使用 `vector` 的引用。

题解 - 分治

使用分治的方法和递归类似，但是不同的是递归是将结果作为参数传入递归函数中，而分治则是先将结果保留，随后再合并到最终结果中。

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> val_vec;

        // NULL or leaf(叶子节点)
        if (root == NULL) {
            return val_vec;
        }

        // Divide (分)
        vector<int> left = preorderTraversal(root->left);
        vector<int> right = preorderTraversal(root->right);

        // Conquer (治)
        val_vec.push_back(root->val);
        val_vec.insert(val_vec.end(), left.begin(), left.end());
        val_vec.insert(val_vec.end(), right.begin(), right.end());

        return val_vec;
    }
};

```

源码分析

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。

题解 - 迭代

迭代时需要利用栈来保存遍历到的节点, 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, 迭代到其为叶子节点(NULL)为止。

Python


```

"""
Definition of TreeNode:
class TreeNode:
    def __init__(self, val):
        this.val = val
        this.left, this.right = None, None
"""

class Solution:
    """
    @param root: The root of binary tree.
    @return: Preorder in ArrayList which contains node values.
    """
    def preorderTraversal(self, root):
        result = []
        if root == None:
            return result

        stack = []
        stack.append(root)
        while stack:
            node = stack.pop()
            result.append(node.val)
            if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)

        return result

```

C++ Iteration

```

/**
 * 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> val_vec;
        stack<const TreeNode *> s;

```

```

    if (root == NULL) {
        return val_vec;
    }

    s.push(root);
    while (!s.empty()) {
        const TreeNode *node = s.top();
        s.pop();

        val_vec.push_back(node->val);

        if (node->right != NULL) {
            s.push(node->right);
        }
        if (node->left != NULL) {
            s.push(node->left);
        }
    }

    return val_vec;
}
};

```

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。

Binary Tree Inorder Traversal

Source

- lintcode: [\(67\) Binary Tree Inorder Traversal](#)

Given a binary tree, return the inorder traversal of its nodes' values.

Example

Given binary tree {1,#,2,3},

```
  1
   \
    2
   /
  3
```

return [1,3,2].

Challenge

Can you do it without recursion?

题解 - 递归版

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。

C++ Recursion

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
/**
 * @param root: The root of binary tree.
 * @return: Inorder in vector which contains node values.
 */
public:
    vector<int> inorderTraversal(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);
        ret.push_back(root->val);
        traverse(root->right, ret);
    }
};

```

题解 - 迭代版

使用辅助栈，空间复杂度 $O(n)$ ，时间复杂度 $O(n)$ 。

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有如下四步构成。

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。

C++ Iteration

```

/**
 * 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) {  
            s.push(root);  
            root = root->left;  
        } else {  
            root = s.top();  
            s.pop();  
            result.push_back(root->val);  
  
            root = root->right;  
        }  
    }  
  
    return result;  
}  
};
```

Binary Tree Postorder Traversal

Source

- lintcode: [\(68\) Binary Tree Postorder Traversal](#)

Given a binary tree, return the postorder traversal of its nodes' values.

Example

Given binary tree {1,#,2,3},

```

  1
   \
    2
   /
  3

```

return [3,2,1].

Challenge

Can you do it without recursion?

题解 - 递归

首先使用递归便于理解。

C++ Recursion

```

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

```

题解 - 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」，由于是最后才将元素取出。因此需要区分左右的访问记录。

C++ Iteration

```

/**
 * 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: Postorder in vector which contains node values.
     */
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        const TreeNode *prevNode = NULL;
        const TreeNode *currNode = root;
        stack<const TreeNode *> s;

        if (root == NULL) {
            return result;
        }

        s.push(root);
        while (!s.empty()) {
            currNode = s.top();
            if (prevNode == NULL || prevNode->left == currNode || prevNode->right == curr

```

```

        // traverse down the tree (left first)
        if (currNode->left) {
            s.push(currNode->left);
        } else if (currNode->right) {
            s.push(currNode->right);
        }
    } else if (currNode->left == prevNode) {
        // traverse up the tree from the left to right
        // the left node has been visited
        if (currNode->right) {
            s.push(currNode->right);
        }
    } else {
        // traverse up the tree from the right
        // visit current node
        result.push_back(currNode->val);
        s.pop();
    }

    prevNode = currNode;
}

return result;
}
};

```

源码解析

使用 `prevNode` 记录之前的访问节点，`currNode` 记录目前正在访问/操作的节点。每次进入while循环时给 `currNode` 赋值，结束时 `prevNode = currNode`。

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。

Binary Tree Level Order Traversal

Source

- lintcode: [\(69\) Binary Tree Level Order Traversal](#)

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right level by level)

Example

Given binary tree {3,9,20,#,#,15,7},

```

      3
     / \
    9  20
   / \  \
  15  7  #

```

return its level order traversal as:

```

[
  [3],
  [9,20],
  [15,7]
]

```

Challenge

Using only 1 queue to implement it.

题解 - 使用队列

此题为广搜的基础题，使用一个队列保存每层的节点即可。

C++ queue

```

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

class Solution {
/**
 * @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;
    }
};

```

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

Maximum Depth of Binary Tree

Source

- lintcode: [\(97\) Maximum Depth of Binary Tree](#)

Given a binary tree, find its maximum depth.

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

Example

Given a binary tree as follow:



The maximum depth is 3

题解 - 递归

树遍历的题最方便的写法自然是递归，不过递归调用的层数过多可能会导致栈空间溢出，因此需要适当考虑递归调用的层数。我们首先来看看使用递归如何解这道题，要求二叉树的最大深度，直观上来讲使用深度优先搜索判断左右子树的深度孰大孰小即可，从根节点往下一层树的深度即自增1，遇到 `NULL` 时即返回 0。

由于对每个节点都会使用一次 `maxDepth`，故时间复杂度为 $O(n)$ ，树的深度最大为 n ，最小为 $\log_2 n$ ，故空间复杂度介于 $O(\log n)$ 和 $O(n)$ 之间。

C++ Recursion

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

```

    * }
    */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);

        return max(left_depth, right_depth) + 1;
    }
};

```

Java Recursion

```

/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        // write your code here
        if (root == null) {
            return 0;
        }
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}

```

题解 - 迭代(显式栈)

使用递归可能会导致栈空间溢出，这里使用显式栈空间(使用堆内存)来代替之前的隐式栈空间。从上节递归版的代码(先处理左子树，后处理右子树，最后返回其中的较大值)来看，是可以使用类似后序遍历的迭代思想去实现的。

首先使用后序遍历的模板，在每次迭代循环结束处比较栈当前的大小和当前最大值 `max_depth` 进行比较。

C++ Iterative with stack

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        TreeNode *curr = NULL, *prev = NULL;
        stack<TreeNode *> s;
        s.push(root);

        int max_depth = 0;

        while(!s.empty()) {
            curr = s.top();
            if (!prev || prev->left == curr || prev->right == curr) {
                if (curr->left) {
                    s.push(curr->left);
                } else if (curr->right) {
                    s.push(curr->right);
                }
            } else if (curr->left == prev) {
                if (curr->right) {
                    s.push(curr->right);
                }
            } else {
                s.pop();
            }

            prev = curr;

            if (s.size() > max_depth) {
                max_depth = s.size();
            }
        }

        return max_depth;
    }
};
```

```
    }
};
```

题解 - 迭代(队列)

在使用了递归/后序遍历求解树最大深度之后，我们还可以直接从问题出发进行分析，树的最大深度即为广度优先搜索中的层数，故可以直接使用广度优先搜索求出最大深度，在原结果返回处使用 `++max_depth` 替代即可。

C++ Iterative with queue

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        queue<TreeNode *> q;
        q.push(root);

        int max_depth = 0;
        while(!q.empty()) {
            int size = q.size();
            for (int i = 0; i != size; ++i) {
                TreeNode *node = q.front();
                q.pop();

                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }

            ++max_depth;
        }
    }
};
```

```

        return max_depth;
    }
};

```

Java Iterative with queue

```

/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int level = 0;
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        int curNum = 1; //num of nodes left in current level
        int nextNum = 0; //num of nodes in next level

        while(!queue.isEmpty()) {
            TreeNode n = queue.poll();
            curNum--;
            if (n.left != null) {
                queue.add(n.left);
                nextNum++;
            }
            if (n.right != null) {
                queue.add(n.right);
                nextNum++;
            }
            if (curNum == 0) {
                curNum = nextNum;
                nextNum = 0;
                level++;
            }
        }
        return level;
    }
}

```

Balanced Binary Tree

Source

- lintcode: [\(93\) Balanced Binary Tree](#)

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the

Example

Given binary tree A={3,9,20,#,#,15,7}, B={3,#,20,15,7}

```

A)  3          B)  3
   / \         \
  9  20        20
   / \       / \
  15  7      15  7
  
```

The binary tree A is a height-balanced binary tree, but B is not.

题解 - 递归

根据题意，平衡树的定义是两子树的深度差最大不超过1，显然使用递归进行分析较为方便。既然使用递归，那么接下来就需要分析递归调用的终止条件。和之前的 [Maximum Depth of Binary Tree | Algorithm](#) 类似，`NULL == root` 必然是其中一个终止条件，返回 0；根据题意还需的另一终止条件应为「左右子树高度差大于1」，但对应此终止条件的返回值是多少？—— `INT_MAX` OR `INT_MIN`？想想都不合适，为何不在传入参数中传入 `bool` 指针或者 `bool` 引用咧？并以此变量作为最终返回值，此法看似可行，先来看看鄙人最开始想到的这种方法。

C++ Recursion with extra bool variable

```

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



```

    * @param root: The root of binary tree.
    * @return: True if this Binary tree is Balanced, or false.
    */
    bool isBalanced(TreeNode *root) {
        if (NULL == root) {
            return true;
        }

        bool result = true;
        maxDepth(root, result);

        return result;
    }

private:
    int maxDepth(TreeNode *root, bool &isBalanced) {
        if (NULL == root) {
            return 0;
        }

        int leftDepth = maxDepth(root->left, isBalanced);
        int rightDepth = maxDepth(root->right, isBalanced);
        if (abs(leftDepth - rightDepth) > 1) {
            isBalanced = false;
            // speed up the recursion process
            return INT_MAX;
        }

        return max(leftDepth, rightDepth) + 1;
    }
};

```

源码解析

如果在某一次子树高度差大于1时，返回 `INT_MAX` 以减少不必要的计算过程，加速整个递归调用的过程。

初看起来上述代码好像还不错的样子，但是在看了九章的实现后，瞬间觉得自己弱爆了... 首先可以确定 `abs(leftDepth - rightDepth) > 1` 肯定是需要特殊处理的，如果返回 `-1` 呢？乍一看似乎在下一步返回 `max(leftDepth, rightDepth) + 1` 时会出错，再进一步想想，我们能否不让 `max...` 这一句执行呢？如果返回了 `-1`，其接盘侠必然是 `leftDepth` 或者 `rightDepth` 中的一个，因此我们只需要在判断子树高度差大于1的同时也判断下左右子树深度是否为 `-1` 即可都返回 `-1`，不得不说这种处理方法要精妙的多，赞！

C++ Recursion without extra bool variable

```

/**
 * forked from http://www.jiuzhang.com/solutions/balanced-binary-tree/
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;

```

```

*         this->left = this->right = NULL;
*     }
* }
*/
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: True if this Binary tree is Balanced, or false.
     */
    bool isBalanced(TreeNode *root) {
        return (-1 != maxDepth(root));
    }

private:
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        if (leftDepth == -1 || rightDepth == -1 || \
            abs(leftDepth - rightDepth) > 1) {
            return -1;
        }

        return max(leftDepth, rightDepth) + 1;
    }
};

```

Binary Tree Maximum Path Sum

Source

- lintcode: [\(94\) Binary Tree Maximum Path Sum](#)

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

Example

Given the below binary tree,

```

      1
     / \
    2   3
Return 6.

```

题解1 - 递归中仅返回子树路径长度

如题目右侧的四颗半五角星所示，本题属于中等偏难的那一类题。题目很短，要求返回最大路径和。乍看一下感觉使用递归应该很快就能搞定，实则不然，因为从题目来看路径和中不一定包含根节点！也就是说可以起止于树中任一连通节点。

弄懂题意后我们就来剖析剖析，本着由简入难的原则，我们先来分析若路径和包含根节点，如何才能使其路径和达到最大呢？选定根节点后，路径和中必然包含有根节点的值，剩下的部分则为左子树和右子树，要使路径和最大，则必然要使左子树和右子树中的路径长度都取最大。

注意区分包含根节点的路径和(题目要的答案)和左子树/右子树部分的路径长度(答案的一个组成部分)。路径和=根节点+左子树路径长度+右子树路径长度

```

    -10
   /  \
  2    -3
 / \  / \
3  4 5  7

```

如上所示，包含根节点 -10 的路径和组成的节点应为 4 -> 2 -> -10 <- -3 <- 7，对于左子树而言，其可能的路径组成节点为 3 -> 2 或 4 -> 2，而不是像根节点的路径和那样为 3 -> 2 <- 4。这种差异也就造成了我们不能很愉快地使用递归来求得最大路径和。

我们使用分治的思想来分析路径和/左子树/右子树，设 $f(\text{root})$ 为 root 的子树到 root 节点(含)路径长度的最大值，那么我们有 $f(\text{root}) = \text{root} \rightarrow \text{val} + \max(f(\text{root} \rightarrow \text{left}), f(\text{root} \rightarrow \text{right}))$

递归模型已初步建立起来，接下来就是分析如何将左右子树的路径长度和最终题目要求的「路径和」挂钩。设 $g(\text{root})$ 为当「路径和」中根节点为 root 时的值，那么我们有 $g(\text{root}) = \text{root} \rightarrow \text{val} + f(\text{root} \rightarrow \text{left}) + f(\text{root} \rightarrow \text{right})$

顺着这个思路，我们可以遍历树中的每一个节点求得 $g(node)$ 的值，输出最大值即可。如果不采取任何记忆化存储的措施，其时间复杂度必然是指数级别的。嗯，先来看看这个思路的具体实现，后续再对其进行优化。遍历节点我们使用递归版的前序遍历，求单边路径长度采用递归。

C++ Recursion + Iteration(Not Recommended)

Time Limit Exceeded

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxPathSum(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int result = INT_MIN;
        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();

            int temp_path_sum = node->val + singlePathSum(node->left) \
                                + singlePathSum(node->right);

            if (temp_path_sum > result) {
                result = temp_path_sum;
            }

            if (NULL != node->right) s.push(node->right);
            if (NULL != node->left) s.push(node->left);
        }

        return result;
    }

private:
    int singlePathSum(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }
    }
}
```

```

    }

    int path_sum = max(singlePathSum(root->left), singlePathSum(root->right));
    return max(0, (root->val + path_sum));
}
};

```

源码分析

注意 `singlePathSum` 中最后的返回值，如果其路径长度 `path_sum` 比0还小，那么取这一段路径反而会减少最终的路径和，故不应取这一段，我们使用0表示这一隐含意义。

题解2 - 递归中同时返回子树路径长度和路径和

C++ using `std::pair`

上题求路径和和左右子树路径长度是分开求得的，因此导致了时间复杂度剧增的恶劣情况，从题解的递推关系我们可以看出其实是在一次递归调用过程中同时求得路径和和左右子树的路径长度的，只不过此时递归程序需要返回的不再是一个值，而是路径长度和路径和这一组值！C++中我们可以使用 `pair` 或者自定义新的数据类型来相对优雅地解决。

```

/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
private:
    pair<int, int> helper(TreeNode *root) {
        if (NULL == root) {
            return make_pair(0, INT_MIN);
        }

        pair<int, int> leftTree = helper(root->left);
        pair<int, int> rightTree = helper(root->right);

        int single_path_sum = max(leftTree.first, rightTree.first) + root->val;
        single_path_sum = max(0, single_path_sum);

        int max_sub_sum = max(leftTree.second, rightTree.second);
        int max_path_sum = root->val + leftTree.first + rightTree.first;
        max_path_sum = max(max_sub_sum, max_path_sum);

        return make_pair(single_path_sum, max_path_sum);
    }
}

```

```

public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxPathSum(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        return helper(root).second;
    }
};

```

源码分析

除了使用 `pair` 对其进行封装，也可使用嵌套类新建一包含单路径长度和全局路径和两个变量的类，不过我用 C++ 写的没编译过... 老是提示 `...private`，遂用 `pair` 改写之。建议使用 `class` 而不是 `pair` 封装 `single_path_sum` 和 `max_path_sum` [pair_is_harmful](#)。

这种方法难以理解的地方在于这种实现方式的正确性，较为关键的语句为 `max_path_sum = max(max_sub_sum, max_path_sum)`，这行代码是如何体现题目中以下的这句话的呢？

The path may start and end at any node in the tree.

简单来讲，题解2从两个方面予以保证：

1. 采用「冒泡」法返回不经过根节点的路径和的较大值。
2. 递推子树路径长度(不变值)而不是到该节点的路径和(有可能变化)，从而保证了这种解法的正确性。

如果还不理解的建议就以上面那个根节点为-10的例子画一画。

C++ using self-defined class

```

/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    class ResultType {
    public:
        int singlePath, maxPath;
        ResultType(int s, int m):singlePath(s), maxPath(m) {}
    };
};

```

```

private:
    ResultType helper(TreeNode *root) {
        if (root == NULL) {
            ResultType *nullResult = new ResultType(0, INT_MIN);
            return *nullResult;
        }
        // Divide
        ResultType left = helper(root->left);
        ResultType right = helper(root->right);

        // Conquer
        int singlePath = max(left.singlePath, right.singlePath) + root->val;
        singlePath = max(singlePath, 0);

        int maxPath = max(left.maxPath, right.maxPath);
        maxPath = max(maxPath, left.singlePath + right.singlePath + root->val);

        ResultType *result = new ResultType(singlePath, maxPath);
        return *result;
    }

public:
    int maxPathSum(TreeNode *root) {
        ResultType result = helper(root);
        return result.maxPath;
    }
};

```

源码分析

1. 如果不用 `ResultType *xxx = new ResultType ...` 再 `return *xxx` 的方式，则需要在自定义 class 中重载 `new` operator。
2. 如果遇到 `...private` 的编译错误，则是因为自定义 class 中需要把成员声明为 `public`，否则需要把访问这个成员的函数也做 class 内部处理。

Reference

- [pair_is_harmful](#). [std::pair considered harmful! « Modern Maintainable Code](#) - 作者指出了 `pair` 不能滥用的原因，如不可维护，信息量小。↩
- [Binary Tree Maximum Path Sum | 九章算法](#)

Lowest Common Ancestor

Source

- lintcode: [\(88\) Lowest Common Ancestor](#)

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of th

The lowest common ancestor is the node with largest depth which is the ancestor of both n
Example

```

      4
     / \
    3   7
     / \
    5   6
  
```

For 3 and 5, the LCA is 4.
For 5 and 6, the LCA is 7.
For 6 and 7, the LCA is 7.

题解1 - 自底向上

初次接触这种题可能会没有什么思路，在没有思路的情况下我们就从简单例子开始分析！首先看 3 和 5，这两个节点分居根节点 4 的两侧，如果可以从子节点往父节点递推，那么他们将在根节点 4 处第一次重合；再来看看 5 和 6，这两个都在根节点 4 的右侧，沿着父节点往上递推，他们将在节点 7 处第一次重合；最后来看看 6 和 7，此时由于 7 是 6 的父节点，故 7 即为所求。从这三个基本例子我们可以总结出两种思路——自顶向下(从前往后递推)和自底向上(从后往前递推)。

顺着上述实例的分析，我们首先看看自底向上的思路，自底向上的实现用一句话来总结就是——如果遍历到的当前节点是 A/B 中的任意一个，那么我们就向父节点汇报此节点，否则递归到节点为空时返回空值。具体来说会有如下几种情况：

1. 当前节点不是两个节点中的任意一个，此时应判断左右子树的返回结果。
 - 若左右子树均返回非空节点，那么当前节点一定是所求的根节点，将当前节点逐层向前汇报。// 两个节点分居树的两侧
 - 若左右子树仅有一个子树返回非空节点，则将此非空节点向父节点汇报。// 节点仅存在于树的一侧
 - 若左右子树均返回 NULL，则向父节点返回 NULL。// 节点不在这棵树中
2. 当前节点即为两个节点中的一个，此时向父节点返回当前节点。

根据此递归模型容易看出应该使用中序遍历来实现。

C++ Recursion From Bottom to Top


```

/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of the binary search tree.
     * @param A and B: two nodes in a Binary.
     * @return: Return the least common ancestor(LCA) of the two nodes.
     */
    TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *A, TreeNode *B) {
        // return either A or B or NULL
        if (NULL == root || root == A || root == B) return root;

        TreeNode *left = lowestCommonAncestor(root->left, A, B);
        TreeNode *right = lowestCommonAncestor(root->right, A, B);

        // A and B are on both sides
        if ((NULL != left) && (NULL != right)) return root;

        // either left or right or NULL
        return (NULL != left) ? left : right;
    }
};

```

源码分析

结合例子和递归的整体思想去理解代码，在 `root == A || root == B` 后即层层上浮(自底向上)，直至找到最终的最小公共祖先节点。

最后一行 `return (NULL != left) ? left : right;` 将非空的左右子树节点和空值都包含在内了，十分精炼！[leetcode](#)

细心的你也许会发现，其实题解的分析漏掉了一种情况，即树中可能只含有 A/B 中的一个节点！这种情况应该返回空值，但上述实现均返回非空节点。重复节点就不考虑了，太复杂了...

不会漏掉 A/B 中只有一个节点的情况的方法

```

public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode A, TreeNode B) {
        if (root == null || root == A || root == B) {
            return root;
        }
    }
}

```

```

        // Divide
        TreeNode left = lowestCommonAncestor(root.left, A, B);
        TreeNode right = lowestCommonAncestor(root.right, A, B);

        // Conquer
        if (left != null && right != null) {
            return root;
        }
        if (left != null) {
            return left;
        }
        if (right != null) {
            return right;
        }
        return null;
    }
}

```

其实这个代码只是把上一个版本的代码最后简洁的判断语句改成复杂的多层判断就可以了。同样是分治法实现。

题解 - 自底向上(计数器)

为了解决上述方法可能导致误判的情况，我们可以对返回结果添加计数器来解决。由于此计数器的值只能由子树向上递推，故不能再使用中序遍历，而应该改用后序遍历。

定义 `pair<TreeNode *, int> result(node, counter)` 表示遍历到某节点时的返回结果，返回的 `node` 表示LCA路径中的可能的最小节点，相应的计数器 `counter` 则表示目前和 A 或者 B 匹配的节点数，若计数器为2，则表示已匹配过两次，该节点即为所求，若只匹配过一次，还需进一步向上递推。表述地可能比较模糊，还是看看代码吧。

C++ Post-order(counter)

```

/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of the binary search tree.
     * @param A and B: two nodes in a Binary.
     * @return: Return the least common ancestor(LCA) of the two nodes.
     */
    TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *A, TreeNode *B) {

```

```

        if ((NULL == A) || (NULL == B)) return NULL;

        pair<TreeNode *, int> result = helper(root, A, B);

        if (A != B) {
            return (2 == result.second) ? result.first : NULL;
        } else {
            return (1 == result.second) ? result.first : NULL;
        }
    }

private:
    pair<TreeNode *, int> helper(TreeNode *root, TreeNode *A, TreeNode *B) {
        TreeNode * node = NULL;
        if (NULL == root) return make_pair(node, 0);

        pair<TreeNode *, int> left = helper(root->left, A, B);
        pair<TreeNode *, int> right = helper(root->right, A, B);

        // return either A or B
        int count = max(left.second, right.second);
        if (A == root || B == root) return make_pair(root, ++count);

        // A and B are on both sides
        if (NULL != left.first && NULL != right.first) return make_pair(root, 2);


        // return either left or right or NULL
        return (NULL != left.first) ? left : right;
    }
};

```

源码分析

在 `A == B` 时，计数器返回1的节点即为我们需要的节点，否则只取返回2的节点，如此便保证了该方法的正确性。对这种实现还有问题的在下面评论吧。

Reference

- [leetcode](#). [Lowest Common Ancestor of a Binary Tree Part I | LeetCode](#) - 清晰易懂的题解和实现。

- [Lowest Common Ancestor of a Binary Tree Part II | LeetCode](#) - 如果存在指向父节点的指针，我们能否有更好的解决方案？
- [Lowest Common Ancestor of a Binary Search Tree \(BST\) | LeetCode](#) - 二叉搜索树中求最小公共祖先。
- [Lowest Common Ancestor | 九章算法](#) - 第一种和第二种方法可以在知道父节点时使用，但第二种 Divide and Conquer 才是本题需要的思想（第二种解法可以轻易改成不需要 parent 的指针的）。

Binary Search Tree - 二叉搜索树

二叉搜索树的定义及简介在 [Binary Search Trees | Algorithm](#) 中已经有所介绍。简单来说就是当前节点的值大于等于左子结点的值，而小于右子节点的值。

Insert Node in a Binary Search Tree

Source

- lintcode: [\(85\) Insert Node in a Binary Search Tree](#)

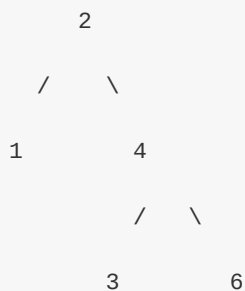
Given a binary search tree and a new tree node, insert the node into the tree. You should

Example

Given binary search tree as follow:



after Insert node 6, the tree should be:



Challenge

Do it without recursion

题解 - 递归

二叉树的题使用递归自然是最好理解的，代码也简洁易懂，缺点就是递归调用时栈空间容易溢出，故实际实现中一般使用迭代替代递归，性能更佳嘛。不过迭代的缺点就是代码量稍(很)大，逻辑也可能不是那么好懂。

既然确定使用递归，那么接下来就应该考虑具体的实现问题了。在递归的具体实现中，主要考虑如下两点：

1. 基本条件/终止条件 - 返回值需斟酌。
2. 递归步/条件递归 - 能使原始问题收敛。

首先来找找递归步，根据二叉查找树的定义，若插入节点的值若大于当前节点的值，则继续与当前节点的

右子树的值进行比较；反之则继续与当前节点的左子树的值进行比较。题目的要求是返回最终二叉搜索树的根节点，从以上递归步的描述中似乎还难以对应到实际代码，这时不妨分析下终止条件。

有了递归步，终止条件也就水到渠成了，若当前节点为空时，即返回结果。问题是——返回什么结果？当前节点为空时，说明应该将「插入节点」插入到上一个遍历节点的左子节点或右子节点。对应到程序代码中即为 `root->right = node` 或者 `root->left = node`。也就是说递归步使用 `root->right/left = func(...)` 即可。

C++ Recursion

```
/**
 * forked from http://www.jiuzhang.com/solutions/insert-node-in-binary-search-tree/
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    TreeNode* insertNode(TreeNode* root, TreeNode* node) {
        if (NULL == root) {
            return node;
        }

        if (node->val <= root->val) {
            root->left = insertNode(root->left, node);
        } else {
            root->right = insertNode(root->right, node);
        }

        return root;
    }
};
```

Java Recursion

```
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
```

```

public TreeNode insertNode(TreeNode root, TreeNode node) {
    if (root == null) {
        return node;
    }
    if (root.val > node.val) {
        root.left = insertNode(root.left, node);
    } else {
        root.right = insertNode(root.right, node);
    }
    return root;
}
}

```

题解 - 迭代

看过了以上递归版的题解，对于这个题来说，将递归转化为迭代的思路也是非常清晰易懂的。迭代比较当前节点的值和插入节点的值，到了二叉树的最后一层时选择是链接至左子结点还是右子节点。

C++

```

/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    TreeNode* insertNode(TreeNode* root, TreeNode* node) {
        if (NULL == root) {
            return node;
        }

        TreeNode* tempNode = root;
        while (NULL != tempNode) {
            if (node->val <= tempNode->val) {
                if (NULL == tempNode->left) {
                    tempNode->left = node;
                    return root;
                }
                tempNode = tempNode->left;
            } else {
                if (NULL == tempNode->right) {
                    tempNode->right = node;

```

```

        return root;
    }
    tempNode = tempNode->right;
}
}
return root;
}
};

```

源码分析

在 `NULL == tempNode->right` 或者 `NULL == tempNode->left` 时需要在链接完 `node` 后立即返回 `root`，避免死循环。

Java Iterative

```

public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    public TreeNode insertNode(TreeNode root, TreeNode node) {
        // write your code here
        if (root == null) return node;
        if (node == null) return root;

        TreeNode rootcopy = root;
        while (root != null) {
            if (root.val <= node.val && root.right == null) {
                root.right = node;
                break;
            }
            else if (root.val > node.val && root.left == null) {
                root.left = node;
                break;
            }
            else if (root.val <= node.val) root = root.right;
            else root = root.left;
        }
        return rootcopy;
    }
}

```


Validate Binary Search Tree

Source

- lintcode: [\(95\) Validate Binary Search Tree](#)

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Example

An example:

```

      1
     /\
    2  3
     /
    4
     \
    5
  
```

The above binary tree is serialized as "{1,2,3,#,#,4,#,#,5}".

题解1 - 递归(比较左右子节点的 key)

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。

C++ Recursion(naive implementation)

```

/**
 * 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 (NULL == root) {
            return true;
        }

        if (NULL == root->left) {
            if ((NULL != root->right) && (root->val >= root->right->val)) {
                return false;
            } else {
                return isValidBST(root->right);
            }
        }

        if (NULL == root->right) {
            if ((NULL != root->left) && (root->val <= root->left->val)) {
                return false;
            } else {
                return isValidBST(root->left);
            }
        }

        if ((root->val > root->left->val) && (root->val < root->right->val)) {
            return isValidBST(root->left) && isValidBST(root->right);
        } else {
            return false;
        }
    }
};

```

源码分析

这种递归方法虽然非常直观，但是需要处理节点指针值为空的边界条件，否则在进行取值操作时会产生运行时错误，这些边界处理使得代码看起来十分不雅观。那么有没有什么优雅的方式处理这种情况呢？还真有，下面就来介绍一种不错的思路。

题解2 - 递归(引入极值简化代码)

对于如何处理复杂边界条件，目前我所知的有两种有效的方法：

1. 引入 dummy node，这个对于链表头不确定时特别有效。
2. 引入 INT_MAX，INT_MIN 等最大/最小值，对于比较类问题很有效(需要考虑到本身值就为最值，这种情况一般极少见)。

显然，对于这个问题，dummy node 是不太适用，我们来探讨下取最值的可能性。从以上代码可知边界处理的关键在于需要判断 root->left 和 root->right 是否为 NULL，key 的比较则相对固定，分别为 root->left->val 和 root->right->val，由此我们自然可以联想到可以使用 left_val 和 right_val 对左右子

树的 key 进行「包装」，在子节点与根节点的 key 进行比较之前对 left_val 和 right_val 赋值即可（这一思想在 lintcode: (65) Median of two Sorted Arrays 和 lintcode: (93) Balanced Binary Tree 中也有使用）。

C++ Recursion(wrapper for root->*->val)

```
/**
 * 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 (NULL == root) {
            return true;
        }

        // wrap key root->left->val, LONG_MIN for true
        long int left_val = (NULL == root->left ? LONG_MIN : root->left->val);
        // wrap key root->right->val, LONG_MAX for true
        long int right_val = (NULL == root->right ? LONG_MAX : root->right->val);

        if ((root->val > left_val) && (root->val < right_val)) {
            return isValidBST(root->left) && isValidBST(root->right);
        } else {
            return false;
        }
    }
};
```

源码分析

根据题解部分的分析，我们使用 left_val 「包装」比较 key 时要用到的 root->left->val，当 root->left 为空时我们置 left_val 为最小值，保证返回结果为 true ——也就是不影响原来的结果；对 right_val 的分析也是同理可得。由于我们使用了 left_val 对原来的子节点值进行了「包装」，故在比较 key 大小时使用 root->val > left_val 是有一丁点bug的，因为有可能出现 root->val == LONG_MIN，这种极端测试用例面试中自己心里有数就好了。

后记：使用了「包装」的方法虽然简化了代码明晰了思路，但是在测试用例中如果出现较多的空节点时，朴素版的代码运行效率会高一些。嗯，不要仅仅只为了代码的优雅而忽略了运行时的效率！

题解3 - 迭代(使用栈改写递归)

看过了上述递归版的思路，客官要不要来一碗迭代版的清酒？理论上讲，任何递归版的程序都可以用迭代实现，当然「理论上讲」一般也就意味着这玩意儿我们远远地看着她就好，可远观而不可亵玩焉... 不卖关子了，我们挑一个简单的递归版程序来试试，嗯，就是上题包装过的递归程序。

函数的每一次递归调用都相当于是一次入栈操作，返回结果则相当于是一次出栈操作。顺着这个思路我们使用一个栈来压入当前节点的左右子节点，入栈前比较其是否符合定义。

C++ Iteration(using stack)

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: True if the binary tree is BST, or false
     */
    bool isValidBST(TreeNode *root) {
        if (NULL == root) {
            return true;
        }

        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();

            // wrap the value of node->left->val
            long int left_val = (NULL == node->left ? LONG_MIN : node->left->val);
            // wrap the value of node->right->val
            long int right_val = (NULL == node->right ? LONG_MAX : node->right->val);

            if ((node->val > left_val) && (node->val < right_val)) {
                if (NULL != node->right) {
                    s.push(node->right);
                }
                if (NULL != node->left) {
                    s.push(node->left);
                }
            } else {
                return false;
            }
        }
    }
}
```

```

    }

    return true;
}
};

```

题解4 - 迭代(使用栈)

从题解3的迭代程序来看这道题似乎可以很方便地直接使用栈解决之。我们来看看迭代的直接思路，压入当前节点的非空子节点的同时判断其是否符合题目所给定义，直至栈为空——所有节点均已处理完。

C++ Iteration

```

/**
 * 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 (NULL == root) {
            return true;
        }

        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();

            // compare the right node
            if (NULL != node->right) {
                if (node->val < node->right->val) {
                    s.push(node->right);
                } else {
                    return false;
                }
            } // end if (NULL...)

            // compare the left node
            if (NULL != node->left) {
                if (node->val > node->left->val) {
                    s.push(node->left);
                }
            }
        }
    }
};

```

```
        } else {  
            return false;  
        }  
    } // end if (NULL...)  
}  
  
return true;  
}  
};
```

Reference

- [LeetCode: Validate Binary Search Tree 解题报告 - Yu's Garden - 博客园](#) - 提供了4种不同的方法，思路可以参考。

Search Range in Binary Search Tree

Source

- lintcode: [\(11\) Search Range in Binary Search Tree](#)

Given two values k_1 and k_2 (where $k_1 < k_2$) and a root pointer to a Binary Search Tree. Find all the keys of tree in range k_1 to k_2 . i.e. print all x such that $k_1 \leq x \leq k_2$ and x is in the tree. Return all the keys in ascending order.

Example

For example, if $k_1 = 10$ and $k_2 = 22$, then your function should print 12, 20 and 22.



题解 - 中序遍历

中等偏易难度题，本题涉及到二叉查找树的按序输出，应马上联想到二叉树的中序遍历，对于二叉查找树而言，使用中序遍历即可得到有序元素。对每次访问的元素加以判断即可得最后结果，由于 OJ 上给的模板不适合递归处理，新建一个私有方法即可。

C++ In-order Recursion

```

/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of the binary search tree.
     * @param k1 and k2: range k1 to k2.
     * @return: Return all keys that  $k_1 \leq \text{key} \leq k_2$  in ascending order.
     */

```

```

    */
    vector<int> searchRange(TreeNode* root, int k1, int k2) {
        vector<int> result;
        inorder_dfs(result, root, k1, k2);

        return result;
    }

private:
    void inorder_dfs(vector<int> &ret, TreeNode *root, int k1, int k2) {
        if (NULL == root) {
            return;
        }

        inorder_dfs(ret, root->left, k1, k2);
        if ((root->val >= k1) && (root->val <= k2)) {
            ret.push_back(root->val);
        }
        inorder_dfs(ret, root->right, k1, k2);
    }
};

```

源码分析

以上为题解思路的简易实现，可以优化的地方为「剪枝过程」的处理——不递归遍历不可能有解的节点。优化后的 `inorder_dfs` 如下：

```

void inorder_dfs(vector<int> &ret, TreeNode *root, int k1, int k2) {
    if (NULL == root) {
        return;
    }

    if ((NULL != root->left) && (root->val > k1)) {
        inorder_dfs(ret, root->left, k1, k2);
    } // cut-off for left sub tree

    if ((root->val >= k1) && (root->val <= k2)) {
        ret.push_back(root->val);
    } // add valid value

    if ((NULL != root->right) && (root->val < k2)) {
        inorder_dfs(ret, root->right, k1, k2);
    } // cut-off for right sub tree
}

```

「剪枝」的判断条件容易出错，应将当前节点的值与 `k1` 和 `k2` 进行比较而不是其左子节点或右子节点的值。

Convert Sorted Array to Binary Search Tree

Source

- leetcode - [Convert Sorted Array to Binary Search Tree | LeetCode OJ](#)

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

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;  
}  
};
```

源码分析

从题解的分析中可以看出中间根节点的建立至关重要！由于数组是可以进行随机访问的，故可取数组中间的索引为根节点，左右子树节点可递归求解。虽然这种递归的过程和「二分搜索」的模板非常像，但是切记本题中根据所给升序序列建立平衡二叉搜索树的过程中需要做到不重不漏，故边界处理需要异常小心，不能再套用 `start + 1 < end` 的模板了。

复杂度分析

递归调用 `middleNode` 方法时每个 `key` 被访问一次，故时间复杂度可近似认为是 $O(n)$ 。

Reference

- [Convert Sorted Array to Binary Search Tree | 九章算法](#)

Convert Sorted List to Binary Search Tree

tags: linked list, binary tree, binary search tree, recursion

Source

- leetcode - [Convert Sorted List to Binary Search Tree | LeetCode OJ](#)
- lintcode - [\(106\) Convert Sorted List to Binary Search Tree](#)

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

题解 - 折半取中

题 [Convert Sorted Array to Binary Search Tree | Data Structure and Algorithm](#) 的升级版，不过这里把「有序数组」换成了「有序链表」。我们可以参考上题的题解思路，思考如何才能在链表中找到「中间节点」。对于本题的单向链表来说，要想知道中间位置的节点，则必须需要知道链表的长度，因此我们就自然联想到了可以通过遍历链表来求得其长度。求得长度我们就知道了链表中间位置节点的索引了，进而根据头节点和当前节点则可将链表分为左右两半形成递归模型。到这里还只能算是解决了问题的一半，这道题另一比较麻烦的地方在于边界条件的取舍，很难第一次就 AC，下面结合代码做进一步的分析。

C++

```
/**
 * 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，首先是递归的终止条件，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` 开始往后递归长度，为 `length` 的链表。故 `length` 为 0 时表示不访问链表中的任一节点，也就是说应该返回 `NULL`。

再来分析链表的中间位置如何确定，我们引入计数器 `count` 来表示目前需要遍历 `count` 个链表节点数目才能得到中间位置的节点。看看四种不同链表长度下的表现。

1. 链表长度为1时，中间位置即为自身，计数器的值为0.
2. 链表长度为2时，中间位置可选第一个节点，也可选第二个节点，相应的计数器值为0或1.
3. 链表长度为3时，中间位置为第二个节点，相应的计数器应为1，表示从表头节点往后递推一个节点。
4. 链表长度为4时，... 计数器的值为1或者2.

从以上四种情况我们可以推断出 `count` 的值可取为 `length / 2` 或者 `length / 2 + 1`，简单起见我们先取 `length / 2` 试试，对应的边界条件即为 `count < length / 2`，`count` 初始值为0. 经过 `count` 次迭代后，目前 `lnode` 即为所需的链表中间节点，取出其值初始化为 `TreeNode` 的根节点。

确定根节点后还需要做的事情就是左子树和右子树中链表头和链表长度的取舍。首先来看看左子树根节点的确定，`count` 的含义为到达中间节点前遍历过的链表节点数目，那么从另一方面来说它就是前半部分链表的长度！故将此长度 `length / 2` 作为得到左子树根节点所需的链表长度参数。除掉链表前半部分节点和中间位置节点这两部分外，剩下的链表长度即为 `length - 1 - length / 2`。

```
length - 1 - length / 2 != length / 2 - 1
```

有没有觉得可以进一步化简为 `length / 2 - 1`？我首先也是这么做的，后来发现一直遇到 `TERMSIG=11` 错误信息，这种错误一般是指针乱指或者指针未初始化就去访问。但自己仔细检查后发现并没有这种错误，于是乎在本地做单元测试，发现原来是死循环造成了栈空间溢出(猜的)！也就是说边界条件有问题！可自己的分析明明是没啥问题的啊...

在这种情况下我默默地打开了九章的参考代码，发现他们竟然没有用 `length / 2 - 1`，而是 `length - 1 - length / 2`。立马意识到这两者可能并不相等。用错误数据试了下，长度为1或者3时两者即不相等。知道对于整型数来说，`1 / 2` 为0，但是却没能活学活用，血泪的教训。：-(一个美好的下午就没了。

在测试出错的时候，还是要相信测试数据的力量，而不是凭自己以前认为对的方式去解决问题。

复杂度分析

首先遍历链表得到链表长度，复杂度为 $O(n)$ 。递归遍历链表时，每个链表节点被访问一次，故时间复杂度为 $O(n)$ ，两者加起来总的复杂度仍为 $O(n)$ 。

进一步简化代码

```
class Solution {
public:
    TreeNode *sortedListToBST(ListNode *head) {
        int length = 0;
        ListNode *curr = head;
        while (curr != NULL) {
            curr = curr->next;
            ++length;
        }
        return helper(head, length);
    }
private:
    TreeNode *helper(ListNode *&pos, int length) {
```

```

        if (length <= 0) {
            return NULL;
        }

        TreeNode *left = helper(pos, length / 2);
        TreeNode *root = new TreeNode(pos->val); // the sequence cannot be changed!
                                                // this is important difference of the s

        pos = pos->next;
        root->left = left;
        root->right = helper(pos, length - length / 2 - 1);
        return root;
    }
};

```

源码分析

1. 可以进一步简化 helper 函数代码，注意参数的接口设计。
2. 即是把传入的链表指针向前递进 n 步，并返回经过的链表节点转化成的二分查找树的根节点。
3. 注意注释中的那两句实现，new root 和 new left 不可调换顺序。这才是精简的要点。但是这种方法不如上面的分治法容易理解。

O(nlogn) 的实现，避免 length 边界

```

/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    public TreeNode sortedListToBST(ListNode head) {
        if (head == null) {
            return null;
        }
        return helper(head);
    }
}

```

```

private TreeNode helper(ListNode head) {
    if (head == null) {
        return null;
    }
    if (head.next == null) {
        return new TreeNode(head.val);
    }

    ListNode pre = null;
    ListNode slow = head, fast = head;

    while (fast != null && fast.next != null) {
        pre = slow;
        slow = slow.next;
        fast = fast.next.next;
    }
    pre.next = null;

    TreeNode root = new TreeNode(slow.val);
    TreeNode L = helper(head);
    TreeNode R = helper(slow.next);
    root.left = L;
    root.right = R;

    return root;
}

```

源码分析

1. 如果想避免上述 length 边界搞错的问题，可以使用分治法遍历树求中点的方法。
2. 但这种时间复杂度是 $O(n\log n)$ ，性能上还是比 $O(n)$ 差一点。

Reference

- [Convert Sorted List to Binary Search Tree | 九章算法](#)

Binary Search Tree Iterator

Source

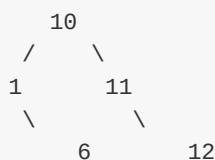
- lintcode: [\(86\) Binary Search Tree Iterator](#)

Design an iterator over a binary search tree with the following rules:

- Elements are visited in ascending order (i.e. an in-order traversal)
- next() and hasNext() queries run in $O(1)$ time in average.

Example

For the following binary search tree, in-order traversal by using iterator is [1, 6, 10,



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。

Backtracking - 回溯法

回溯法包含了多类问题，模板类似。

排列组合模板->搜索问题(是否要排序，哪些情况要跳过)

使用回溯法的一般步骤：

1. 确定所给问题的解空间：首先应明确定义问题的解空间，解空间中至少包含问题的一个解。
2. 确定结点的扩展搜索规则
3. 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

Reference

- [Steven Skiena: Lecture15 - Backtracking](#)
- [全面解析回溯法：算法框架与问题求解 - 五岳 - 博客园](#)
- [五大常用算法之四：回溯法 - 红脸书生 - 博客园](#)
- [演算法筆記 - Backtracking](#)

Subsets - 子集

子集类问题类似Combination。

Source

- lintcode: [\(17\) Subsets](#)

Given a set of distinct integers, return all possible subsets.

Note

Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

Example

If S = [1,2,3], a solution is:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

题解

1. 首先对数组按升序排序
2. 回溯法递归

Java

```
class Solution {
    /**
     * @param S: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    public ArrayList<ArrayList<Integer>> subsets(ArrayList<Integer> S) {

        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (S == null || S.size() == 0) {
            return result;
        }

        ArrayList<Integer> list = new ArrayList<Integer>();
        Collections.sort(S);
```

```

        backtrack(result, list, S, 0);
        return result;
    }

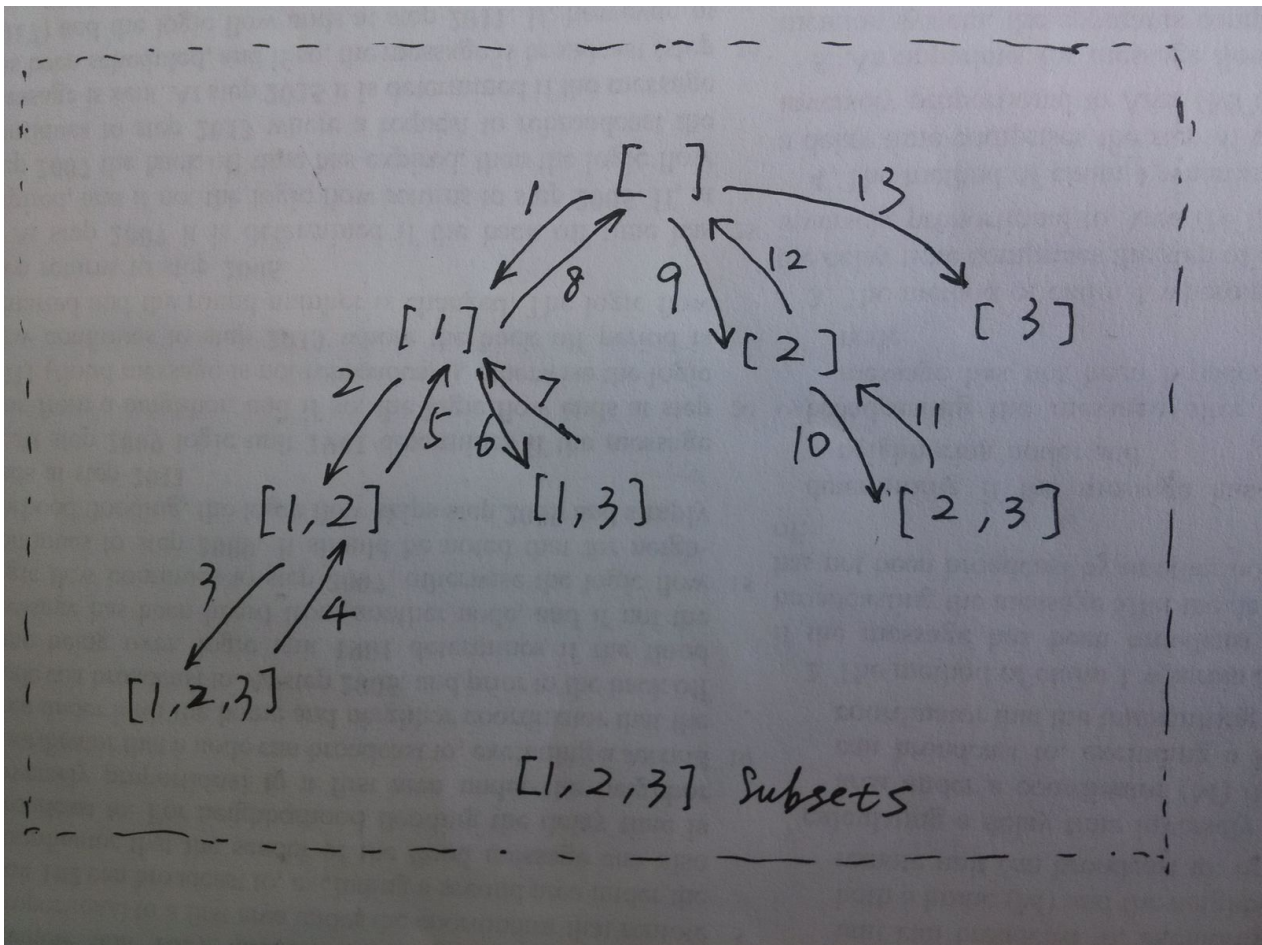
    private void backtrack(ArrayList<ArrayList<Integer>> result,
        ArrayList<Integer> list, ArrayList<Integer> num, int pos) {
        result.add(new ArrayList<Integer>(list));

        for (int i = pos; i < num.size(); i++) {
            list.add(num.get(i));
            backtrack(result, list, num, i + 1);
            list.remove(list.size() - 1);
        }
    }
}

```

Notice: `backTrack(result, list, num, i + 1);`中的『`i + 1`』不可误写为『`pos + 1`』，第一次写subsets的时候在这坑了很久... :(

回溯法可用图示和函数运行的堆栈图来理解，强烈建议使用图形和递归的思想分析，以数组 `[1, 2, 3]` 进行分析。下图所示为 `list` 及 `result` 动态变化的过程，箭头向下表示 `list.add` 及 `result.add` 操作，箭头向上表示 `list.remove` 操作。



如果你不相信以上的图形化分析，还可以自己在纸上分析代码的调用关系，下面以数组 `[1, 2]` 为例分析回溯法的调用栈。

1. 首先由主函数 `subsets` 进入，初始化 `result` 为`[]`，接着进行异常处理，随后初始化 `list` 为`[]`，递归调用 `backTrack()`，`num = [1, 2]`。
2. `result = []`，`list = []`，`pos = 0`。调用 `result.add()` 加入`[]`，`result = [[]]`。进入 `for` 循环，`num.size() = 2`。
 - o `i = 0`，
 - i. `list.add(num[0]) -> list = [1]`，递归调用 `backTrack()` 前，`result = [[]]`，`list = [1]`，`pos = 1`
 - ii. 递归调用 `backTrack([[]], [1], [1, 2], 1)`
 - `result.add([1]) -> result = [[], [1]]`
 - `i = 1`，`for(i = 1 < 2)`
 - i. `list.add(num[1]) -> list = [1, 2]`
 - ii. 递归调用 `backTrack([[], [1]], [1, 2], [1, 2], 2)`
 - `result.add([1, 2]) -> result = [[], [1], [1, 2]]`
 - `i = 2` 退出`for`循环，退出此次调用
 - iii. `list.remove(2 - 1) -> list = [1]`
 - iv. `i++ -> i = 2`
 - `i = 2`，退出`for`循环，退出此次调用
 - iii. `list.remove()` -> `list = []`
 - iv. `i++ -> i = 1`，进入下一次循环
 - o `i = 1`，`for(i = 1 < 2)`
 - i. `list.add(num[1]) -> list = [2]`
 - ii. 递归调用 `backTrack([[], [1], [1, 2]], [2], [1, 2], 2)`
 - `result.add([2]) -> result = [[], [1], [1, 2], [2]]`
 - `i = 2` 退出`for`循环，退出此次调用
 - iii. `list.remove(1 - 1) -> list = []`
 - iv. `i++ -> i = 2`
 - o `i = 2`，退出`for`循环，退出此次调用
3. 返回结果 `result`

C++

```
class Solution {
public:
    /**
     * @param S: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    vector<vector<int>> subsets(vector<int> &nums) {
        vector<vector<int>> result;
        if (nums.empty()) {
            return result;
        }

        vector<int> list;
        backTrack(result, list, nums, 0);

        return result;
    }
}
```

```
private:
    void backtrack(vector<vector<int>> &result, vector<int> &list, \
                  vector<int> &nums, int pos) {
        result.push_back(list);

        for (int i = pos; i != nums.size(); ++i) {
            list.push_back(nums[i]);
            backtrack(result, list, nums, i + 1);
            list.pop_back();
        }
    }
};
```

Unique Subsets

Source

- lintcode: [\(18\) Unique Subsets](#)

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

Note

Each element in a subset must be in non-descending order.

The ordering between two subsets is free.

The solution set must not contain duplicate subsets.

Example

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

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

题解

此题在上一题的基础上加了有重复元素的情况，因此需要对回溯函数进行一定的剪枝，对于排列组合的模板程序，剪枝通常可以从两个地方出发，一是在返回结果 `result.add` 之前进行剪枝，另一个则是在 `list.add` 处剪枝，具体使用哪一种需要视情况而定，哪种简单就选谁。

由于此题所给数组不一定有序，故首先需要排序。有重复元素对最终结果的影响在于重复元素最多只能出现 `n` 次(重复个数为 `n` 时)。具体分析过程如下(此分析过程改编自 [九章算法](#))。

以 $[1, 2_1, 2_2]$ 为例，若不考虑重复，组合有 $[], [1], [1, 2_1], [1, 2_1, 2_2], [1, 2_2], [2_1], [2_1, 2_2], [2_2]$ 。其中重复的有 $[1, 2_2], [2_2]$ 。从中我们可以看出只能从重复元素的第一个持续往下添加到列表中，而不能取第二个或之后的重复元素。参考上一题 Subsets 的模板，能代表「重复元素的第一个」即为 for 循环中的 `pos` 变量，`i == pos` 时，`i` 处所代表的变量即为某一层遍历中得「第一个元素」，因此去重时只需判断 `i != pos && s[i] == s[i - 1]`。

C++

```
class Solution {
public:
    /**
     * @param S: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    vector<vector<int>> > subsetsWithDup(const vector<int> &S) {
        vector<vector<int>> > result;
        if (S.empty()) {
            return result;
        }

        vector<int> list;
        vector<int> source(S);
        sort(source.begin(), source.end());
        backtrack(result, list, source, 0);

        return result;
    }

private:
    void backtrack(vector<vector<int>> &ret, vector<int> &list,
                  vector<int> &s, int pos) {

        ret.push_back(list);

        for (int i = pos; i != s.size(); ++i) {
            if (i != pos && s[i] == s[i - 1]) {
                continue;
            }
            list.push_back(s[i]);
            backtrack(ret, list, s, i + 1);
            list.pop_back();
        }
    }
};
```

Reference - 参考

- [\[NineChap 1.2\] Permutation - Woodstock Blog](#)
- [九章算法 - subsets模板](#)
- [LeetCode: Subsets 解题报告 - Yu's Garden - 博客园](#)
- [九章算法 | Subsets II](#)

Permutation - 排列

排列常见的有数字全排列，字符串排列等。

Permutations - 全排列

Source

- lintcode: [\(15\) Permutations](#)

Given a list of numbers, return all possible permutations.

Example

For nums [1,2,3], the permutaions are:

```
[  
  [1,2,3],  
  [1,3,2],  
  [2,1,3],  
  [2,3,1],  
  [3,1,2],  
  [3,2,1]  
]
```

Challenge

Do it without recursion

题解

使用之前subsets的模板，但是在取结果时只能取 `list.size() == nums.size()` 的解，且在添加list元素的时候需要注意除重。

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();
        }
    }
};

```

源码分析

在除重时使用了标准库 `find` (不可使用时间复杂度更低的 `binary_search` , 因为 `list` 中元素不一定有序), 时间复杂度为 $O(N)$, 也可使用 `hashmap` 记录 `nums` 中每个元素是否被添加到 `list` 中, 这样一来空间复杂度为 $O(N)$, 查找的时间复杂度为 $O(1)$.

在 `list.size() == nums.size()` 时, 已经找到需要的解, 及时 `return` 避免后面不必要的 `for` 循环调用开销。

使用回溯法解题的关键在于如何确定正确解及排除不符合条件的解(剪枝)。

Unique Permutations

Source

- lintcode: [\(16\) Unique Permutations](#)

Given a list of numbers with duplicate number in it. Find all unique permutations.

Example

For numbers [1,2,2] the unique permutations are:

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

Challenge
Do it without recursion.

题解

在上题的基础上进行剪枝，剪枝的过程和 [Subsets | Algorithm](#) 中的 Unique Subsets 一题极为相似。为了便于分析，我们可以先分析简单的例子，以 $[1, 2_1, 2_2]$ 为例。按照上题 Permutations 的解法，我们可以得到如下全排列。

1. $[1, 2_1, 2_2]$
2. $[1, 2_2, 2_1]$
3. $[2_1, 1, 2_2]$
4. $[2_1, 2_2, 1]$
5. $[2_2, 1, 2_1]$
6. $[2_2, 2_1, 1]$

从以上结果我们注意到 1 和 2 重复，5 和 3 重复，6 和 4 重复，从重复的解我们可以发现其共同特征均是第二个 2_2 在前，而第一个 2_1 在后，因此我们的剪枝方法为：对于有相同的元素来说，我们只取不重复的一次。嗯，这样说还是有点模糊，下面以 $[1, 2_1, 2_2]$ 和 $[1, 2_2, 2_1]$ 进行说明。

首先可以确定 $[1, 2_1, 2_2]$ 是我们要的一个解，此时 list 为 $[1, 2_1, 2_2]$ ，经过两次 `list.pop_back()` 之后，list 为 $[1]$ ，如果不进行剪枝，那么接下来要加入 list 的将为 2_2 ，那么我们剪枝要做的就是避免将 2_2 加入到 list 中，如何才能做到这一点呢？我们仍然从上述例子出发进行分析，在第一次加入 2_2 时，相对应的 `visited[1]` 为 `true` (对应 2_1)，而在第二次加入 2_2 时，相对应的 `visited[1]` 为 `false`，因为在 list 为 $[1, 2_1]$ 时，执行 `list.pop_back()` 后即置为 `false`。

一句话总结即为：在遇到当前元素和前一个元素相等时，如果前一个元素 `visited[i - 1] == false`，那么我们就跳过当前元素并进入下一次循环，这就是剪枝的关键所在。另一点需要特别注意的是这种剪枝的方法能使用的前提是提供的 `nums` 是有序数组，否则无效。

C++

```
class Solution {
public:
```

```

/**
 * @param nums: A list of integers.
 * @return: A list of unique permutations.
 */
vector<vector<int>> > permuteUnique(vector<int> &nums) {
    vector<vector<int>> > ret;
    if (nums.empty()) {
        return ret;
    }

    // important! sort before call `backTrack`
    sort(nums.begin(), nums.end());
    vector<bool> visited(nums.size(), false);
    vector<int> list;
    backTrack(ret, list, visited, nums);

    return ret;
}

private:
void backTrack(vector<vector<int>> > &result, vector<int> &list, \
               vector<bool> &visited, vector<int> &nums) {
    if (list.size() == nums.size()) {
        result.push_back(list);
        // avoid unnecessary call for `for loop`, but not essential
        return;
    }

    for (int i = 0; i != nums.size(); ++i) {
        if (visited[i] || (i != 0 && nums[i] == nums[i - 1] \
                        && !visited[i - 1])) {
            continue;
        }
        visited[i] = true;
        list.push_back(nums[i]);
        backTrack(result, list, visited, nums);
        list.pop_back();
        visited[i] = false;
    }
}
};

```

源码解析

Unique Subsets 和 Unique Permutations 的源码模板非常经典！建议仔细研读并体会其中奥义。

后记：在理解 Unique Subsets 和 Unique Permutations 的模板我花了差不多一整天时间才基本理解透彻，建议在想不清楚某些问题时先分析简单的问题，在纸上一步一步分析直至理解完全。

Reference

- [九章算法 | Permutation II](#)

Dynamic Programming - 动态规划

动态规划是一种「分治」的思想，通俗一点来说就是「大事化小，小事化无」的艺术。在将大问题化解为小问题的「分治」过程中，保存对这些问题已经处理好的结果，并供后面处理更大规模的问题时直接使用这些结果。嗯，感觉讲了和没讲一样，还是不会使用动规的思想解题...

下面看看知乎上的熊大大对动规比较「正经」的描述。

动态规划是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。

以上定义言简意赅，可直接用于实战指导，不愧是参加过NOI的。

动规的思想虽然好理解，但是要真正活用起来就需要下点功夫了。建议看看下面知乎上的回答。

动态规划最重要的两个要点：

1. 状态(状态不太好找，可先从转化方程入手分析)
2. 状态间的转化方程(从题目的隐含条件出发寻找递推关系)

其他的要点则是如初始化状态的确定(由状态和转化方程得知)，需要的结果(状态转移的终点)

动态规划问题中一般从以下四个角度考虑：

1. 状态(State)
2. 状态间的转移方程(Function)
3. 状态的初始化(Initialization)
4. 返回结果(Answer)

动规适用的情形：

1. 最大值/最小值
2. 有无可行解
3. 求方案个数(如果需要列出所有方案，则一定不是动规，因为全部方案为指数级别复杂度，所有方案需要列出时往往用递归)
4. 给出的数据不可随便调整位置

纸上得来终觉浅，绝知此事要躬行。光说不练假把戏，下面就来几道DP的题练练手。

Reference

1. [什么是动态规划？动态规划的意义是什么？ - 知乎](#) - 熊大大和王勃的回答值得细看，适合作为动态规划的科普和入门。维基百科上对动态规划的描述感觉太过学术。
2. [动态规划：从新手到专家](#) - Topcoder上的一篇译作。

Triangle - Find the minimum path sum from top to bottom

Source

- lintcode: [\(109\) Triangle](#)

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to

Note

Bonus point if you are able to do this using only $O(n)$ extra space, where n is the total

Example

For example, given the following triangle

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

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

题解

题中要求最短路径和，每次只能访问下行的相邻元素，将triangle视为二维坐标。此题方法较多，下面分小节详述。

Method 1 - Traverse without hashmap

首先考虑最容易想到的方法——递归遍历，逐个累加所有自上而下的路径长度，最后返回这些不同的路径长度的最小值。由于每个点往下都有2条路径，使用此方法的时间复杂度约为 $O(2^n)$ ，显然是不可接受的解，不过我们还是先看看其实现思路。

C++ Traverse without hashmap

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int>> &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        int result = INT_MAX;
```

```

        dfs(0, 0, 0, triangle, result);

        return result;
    }

private:
    void dfs(int x, int y, int sum, vector<vector<int>> &triangle, int &result) {
        const int n = triangle.size();
        if (x == n) {
            if (sum < result) {
                result = sum;
            }
            return;
        }

        dfs(x + 1, y, (sum + triangle[x][y]), triangle, result);
        dfs(x + 1, y + 1, (sum + triangle[x][y]), triangle, result);
    }
};

```

源码分析

dfs() 的循环终止条件为 `x == n`，而不是 `x == n - 1`，主要是方便在递归时sum均可使用 `sum + triangle[x][y]`，而不必根据不同的y和y+1改变，代码实现相对优雅一些。理解方式则变为从第x行走到第x+1行时的最短路径和，也就是说在此之前并不将第x行的元素值计算在内。

这种遍历的方法时间复杂度如此之高的主要原因是因为在n较大时递归计算了之前已经得到的结果，而这些结果计算一次后即不再变化，可再次利用。因此我们可以使用hashmap记忆已经计算得到的结果从而对其进行优化。

Method 2 - Divide and Conquer without hashmap

既然可以使用递归遍历，当然也可以使用「分治」的方法来解。「分治」与之前的遍历区别在于「分治」需要返回每次「分治」后的计算结果，下面看代码实现。

C++ Divide and Conquer without hashmap

```

class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int>> &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        int result = dfs(0, 0, triangle);

        return result;
    }
}

```

```
private:
    int dfs(int x, int y, vector<vector<int> > &triangle) {
        const int n = triangle.size();
        if (x == n) {
            return 0;
        }

        return min(dfs(x + 1, y, triangle), dfs(x + 1, y + 1, triangle)) + triangle[x][y];
    }
};
```

使用「分治」的方法代码相对简洁一点，接下来我们使用hashmap保存triangle中不同坐标的点计算得到的路径和。

Method 3 - Divide and Conquer with hashmap

新建一份大小和triangle一样大小的hashmap，并对每个元素赋以 INT_MIN 以做标记区分。

C++ Divide and Conquer with hashmap

```
class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int> > &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        vector<vector<int> > hashmap(triangle);
        for (int i = 0; i != hashmap.size(); ++i) {
            for (int j = 0; j != hashmap[i].size(); ++j) {
                hashmap[i][j] = INT_MIN;
            }
        }
        int result = dfs(0, 0, triangle, hashmap);

        return result;
    }

private:
    int dfs(int x, int y, vector<vector<int> > &triangle, vector<vector<int> > &hashmap)
        const int n = triangle.size();
        if (x == n) {
            return 0;
        }

        // INT_MIN means no value yet
        if (hashmap[x][y] != INT_MIN) {
            return hashmap[x][y];
        }
        int x1y = dfs(x + 1, y, triangle, hashmap);
```

```

        int x1y1 = dfs(x + 1, y + 1, triangle, hashmap);
        hashmap[x][y] = min(x1y, x1y1) + triangle[x][y];

        return hashmap[x][y];
    }
};

```

由于已经计算出的最短路径值不再重复计算，计算复杂度由之前的 $O(2^n)$ ，变为 $O(n^2)$ ，每个坐标的元素仅计算一次，故共计算的次数为 $1 + 2 + \dots + n \approx O(n^2)$ 。

Method 4 - Dynamic Programming

从主章节中对动态规划的简介我们可以知道使用动态规划的难点和核心在于状态的定义及转化方程的建立。那么问题来了，到底如何去找适合这个问题的状态及转化方程呢？

我们仔细分析题中可能的状态和转化关系，发现从 `triangle` 中坐标为 $triangle[x][y]$ 的元素出发，其路径只可能为 $triangle[x][y] \rightarrow triangle[x+1][y]$ 或者 $triangle[x][y] \rightarrow triangle[x+1][y+1]$ 。以点 (x, y) 作为参考，那么可能的状态 $f(x, y)$ 就可以是：

1. 从 (x, y) 出发走到最后一行的最短路径和
2. 从 $(0, 0)$ 走到 (x, y) 的最短路径和

如果选择1作为状态，则相应的状态转移方程为：

$$f_1(x, y) = \min\{f_1(x+1, y), f_1(x+1, y+1)\} + triangle[x][y]$$

如果选择2作为状态，则相应的状态转移方程为：

$$f_2(x, y) = \min\{f_2(x-1, y), f_2(x-1, y-1)\} + triangle[x][y]$$

两个状态所对应的初始状态分别为 $f_1(n-1, y), 0 \leq y \leq n-1$ 和 $f_2(0, 0)$ 。在代码中应注意考虑边界条件。下面分别就这种不同的状态进行动态规划。

C++ From Bottom to Top

```

class Solution {
public:
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    int minimumTotal(vector<vector<int>> &triangle) {
        if (triangle.empty()) {
            return -1;
        }

        vector<vector<int>> > hashmap(triangle);

        // get the total row number of triangle
        const int N = triangle.size();
    }
};

```



```

        for (int i = 0; i != N; ++i) {
            hashmap[N-1][i] = triangle[N-1][i];
        }

        for (int i = N - 2; i >= 0; --i) {
            for (int j = 0; j < i + 1; ++j) {
                hashmap[i][j] = min(hashmap[i + 1][j], hashmap[i + 1][j + 1]) + triangle[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。

Knapsack - 背包问题

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Knapsack without repetition - 01背包问题

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

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

现在从以上状态定义出发寻找相应的状态转移方程。 $K(i-1, \omega)$ 为 $K(i, \omega)$ 的子问题, 如果不放第 i 件物品, 那么问题即转化为「前 $i-1$ 件物品放入容量为 ω 的背包」, 此时背包内获得的总价值为

$K(i-1, \omega)$; 如果放入第 i 件物品, 那么问题即转化为「前 $i-1$ 件物品放入容量为 $\omega - \omega_i$ 的背包」, 此时背包内获得的总价值为 $K(i-1, \omega - \omega_i) + v_i$. 新的状态转移方程用数学语言来表述即为:

$$K(i, \omega) = \max\{K(i-1, \omega), K(i-1, \omega - \omega_i) + v_i\}$$

Reference

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

Backpack

Source

- lintcode: [\(92\) Backpack](#)

Given n items with size $A[i]$, an integer m denotes the size of a backpack. How full you can

Note

You can not divide any item into small pieces.

Example

If we have 4 items with size $[2, 3, 5, 7]$, the backpack size is 11, we can select 2, 3 and

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

题解

本题是典型的01背包问题，每种类型的物品最多只能选择一件。先来看看 [九章算法](#) 的### 题解

1. 状态: $result[i][S]$ 表示前 i 个物品，取出一些物品能否组成体积和为 S 的背包
2. 状态转移方程: $f[i][S] = f[i-1][S - A[i]] \text{ or } f[i-1][S]$ ($A[i]$ 为第 i 个物品的大小)
 - 欲从前 i 个物品中取出一些组成体积和为 S 的背包，可从两个状态转换得到。
 - i. $f[i-1][S - A[i]]$: 放入第 i 个物品，前 $i-1$ 个物品能否取出一些体积和为 $S - A[i]$ 的背包。
 - ii. $f[i-1][S]$: 不放入第 i 个物品，前 $i-1$ 个物品能否取出一些组成体积和为 S 的背包。
3. 状态初始化: $f[1 \cdots n][0] = true$; $f[0][1 \cdots m] = false$. 前 $1 \sim n$ 个物品组成体积和为 0 的背包始终为真，其他情况为假。
4. 返回结果: 寻找使 $f[n][S]$ 值为 true 的最大 S ($1 \leq S \leq m$)

C++ 2D vector for result

```
class Solution {
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    int backPack(int m, vector<int> A) {
        if (A.empty() || m < 1) {
            return 0;
        }

        const int N = A.size() + 1;
        const int M = m + 1;
        vector<vector<bool>> result;
```

```

        result.resize(N);
        for (vector<int>::size_type i = 0; i != N; ++i) {
            result[i].resize(M);
            std::fill(result[i].begin(), result[i].end(), false);
        }

        result[0][0] = true;
        for (int i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                if (j < A[i - 1]) {
                    result[i][j] = result[i - 1][j];
                } else {
                    result[i][j] = result[i - 1][j] || result[i - 1][j - A[i - 1]];
                }
            }
        }

        // return the largest i if true
        for (int i = M; i > 0; --i) {
            if (result[N - 1][i - 1]) {
                return (i - 1);
            }
        }
        return 0;
    }
};

```

源码分析

1. 异常处理
2. 初始化结果矩阵，注意这里需要使用 `resize` 而不是 `reserve`，否则可能会出现段错误
3. 实现状态转移逻辑，一定要分 $j < A[i - 1]$ 与否来讨论
4. 返回结果，只需要比较 `result[N - 1][i - 1]` 的结果，返回true的最大值

状态转移逻辑中代码可以进一步简化，即：

```

        for (int i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                result[i][j] = result[i - 1][j];
                if (j >= A[i - 1] && result[i - 1][j - A[i - 1]]) {
                    result[i][j] = true;
                }
            }
        }
    }

```

考虑背包问题的核心——状态转移方程，如何优化此转移方程？原始方案中用到了二维矩阵来保存result，注意到result的第i行仅依赖于第i-1行的结果，那么能否用一维数组来代替这种隐含的关系呢？我们在内循环j处递减即可。如此即可避免 `result[i][S]` 的值由本轮 `result[i][S-A[i]]` 递推得到。

C++ 1D vector for result

```

class Solution {

```

```

public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    int backPack(int m, vector<int> A) {
        if (A.empty() || m < 1) {
            return 0;
        }

        const int N = A.size();
        vector<bool> result;
        result.resize(m + 1);
        std::fill(result.begin(), result.end(), false);

        result[0] = true;
        for (int i = 0; i != N; ++i) {
            for (int j = m; j >= 0; --j) {
                if (j >= A[i] && result[j - A[i]]) {
                    result[j] = true;
                }
            }
        }

        // return the largest i if true
        for (int i = m; i > 0; --i) {
            if (result[i]) {
                return i;
            }
        }
        return 0;
    }
};

```

Backpack II

Source

- lintcode: [\(125\) Backpack II](#)

Given n items with size A[i] and value V[i], and a backpack with size m. What's the maxim

Note

You cannot divide item into small pieces and the total size of items you choose should sm

Example

Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and a backpack with size 10.

题解

首先定义状态 $K(i, w)$ 为前 i 个物品放入size为 w 的背包中所获得的最大价值，则相应的状态转移方程

为： $K(i, w) = \max\{K(i-1, w), K(i-1, w-w_i) + v_i\}$

详细分析过程见本节。

C++ 2D vector for result

```
class Solution {
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    int backPackII(int m, vector<int> A, vector<int> V) {
        if (A.empty() || V.empty() || m < 1) {
            return 0;
        }
        const int N = A.size() + 1;
        const int M = m + 1;
        vector<vector<int>> result;
        result.resize(N);
        for (vector<int>::size_type i = 0; i != N; ++i) {
            result[i].resize(M);
            std::fill(result[i].begin(), result[i].end(), 0);
        }

        for (vector<int>::size_type i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                if (j < A[i - 1]) {
                    result[i][j] = result[i - 1][j];
                } else {
                    int temp = result[i - 1][j - A[i - 1]] + V[i - 1];
                    result[i][j] = max(temp, result[i - 1][j]);
                }
            }
        }

        return result[N - 1][M - 1];
    }
};
```

源码分析

1. 使用二维矩阵保存结果result
2. 返回result矩阵的右下角元素——背包size限制为m时的最大价值

按照第一题backpack的思路，这里可以使用一维数组进行空间复杂度优化。优化方法为逆序求 $result[j]$ ，优化后的代码如下：

C++ 1D vector for result

```
class Solution {
```



```

public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    int backPackII(int m, vector<int> A, vector<int> V) {
        if (A.empty() || V.empty() || m < 1) {
            return 0;
        }

        const int M = m + 1;
        vector<int> result;
        result.resize(M);
        std::fill(result.begin(), result.end(), 0);

        for (vector<int>::size_type i = 0; i != A.size(); ++i) {
            for (int j = m; j >= 0; --j) {
                if (j < A[i]) {
                    // result[j] = result[j];
                } else {
                    int temp = result[j - A[i]] + V[i];
                    result[j] = max(temp, result[j]);
                }
            }
        }

        return result[M - 1];
    }
};

```

Reference

- [Lintcode: Backpack - neverlandly - 博客园](#)
- [Lintcode: Backpack II - neverlandly - 博客园](#)
- [九章算法 | 背包问题](#)
- [崔添翼 § 翼若垂天之云 > 《背包问题九讲》2.0 alpha1](#)

Matrix

本节主要总结矩阵类动态规划问题，根据动态规划解题的四要素，矩阵类动态规划问题可以从以下四个方面进行分析：

1. State: $f[x][y]$ 从起点走到坐标(x,y)的值
2. Function: 走到坐标(x,y)之前的状态转移
3. Initialization: 初始状态 - 起点
4. Answer: 终点

Minimum Path Sum

Source

- lintcode: [\(110\) Minimum Path Sum](#)

Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom

Note

You can only move either down or right at any point in time.

题解

1. State: $f[x][y]$ 从坐标(0,0)走到(x,y)的最短路径和
2. Function: $f[x][y] = (x, y) + \min\{f[x-1][y], f[x][y-1]\}$
3. Initialization: $f[0][0] = A[0][0]$, $f[i][0] = \text{sum}(0,0 \rightarrow i,0)$, $f[0][i] = \text{sum}(0,0 \rightarrow 0,i)$
4. Answer: $f[m-1][n-1]$

注意最后返回为 $f[m-1][n-1]$ 而不是 $f[m][n]$.

首先看看如下正确但不合适的答案，OJ上会出现 **TLE**。未使用hashmap并且使用了递归的错误版本。

C++ dfs without hashmap: Wrong answer

```
class Solution {
public:
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    int minPathSum(vector<vector<int>> &grid) {
        if (grid.empty()) {
            return 0;
        }

        const int m = grid.size() - 1;
        const int n = grid[0].size() - 1;

        return helper(grid, m, n);
    }

private:
    int helper(vector<vector<int>> &grid_in, int x, int y) {
        if (0 == x && 0 == y) {
            return grid_in[0][0];
        }
        if (0 == x) {
```

```

        return helper(grid_in, x, y - 1) + grid_in[x][y];
    }
    if (0 == y) {
        return helper(grid_in, x - 1, y) + grid_in[x][y];
    }

    return grid_in[x][y] + min(helper(grid_in, x - 1, y), helper(grid_in, x, y - 1));
}
};

```

使用迭代思想进行求解的正确实现：

C++ Iterative

```

class Solution {
public:
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    int minPathSum(vector<vector<int>> &grid) {
        if (grid.empty() || grid[0].empty()) {
            return 0;
        }

        const int M = grid.size();
        const int N = grid[0].size();
        vector<vector<int>> ret(M, vector<int>(N, 0));

        ret[0][0] = grid[0][0];
        for (int i = 1; i != M; ++i) {
            ret[i][0] = grid[i][0] + ret[i - 1][0];
        }
        for (int i = 1; i != N; ++i) {
            ret[0][i] = grid[0][i] + ret[0][i - 1];
        }

        for (int i = 1; i != M; ++i) {
            for (int j = 1; j != N; ++j) {
                ret[i][j] = grid[i][j] + min(ret[i - 1][j], ret[i][j - 1]);
            }
        }

        return ret[M - 1][N - 1];
    }
};

```

源码分析

1. 异常处理，不仅要分析grid还要分析grid[0]
2. 对返回结果矩阵进行初始化，注意ret[0][0]须单独初始化以便使用ret[i-1]
3. 递推时i和j均从1开始
4. 返回结果ret[M-1][N-1]，注意下标是从0开始的

此题还可进行空间复杂度优化，和背包问题类似，使用一维数组代替二维矩阵也行，具体代码可参考 [水中的鱼: \[LeetCode\] Minimum Path Sum 解题报告](#)

优化空间复杂度，要么对行遍历进行优化，要么对列遍历进行优化，通常我们习惯先按行遍历再按列遍历，有状态转移方程 $f[x][y] = (x, y) + \min\{f[x-1][y], f[x][y-1]\}$ 知，想要优化行遍历，那么 $f[y]$ 保存的值应为第 x 行第 y 列的和。由于无行下标信息，故初始化时仅能对第一个元素初始化，分析时建议画图理解。

C++ 1D vector

```
class Solution {
public:
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    int minPathSum(vector<vector<int>> &grid) {
        if (grid.empty() || grid[0].empty()) {
            return 0;
        }

        const int M = grid.size();
        const int N = grid[0].size();
        vector<int> ret(N, INT_MAX);

        ret[0] = 0;

        for (int i = 0; i != M; ++i) {
            ret[0] = ret[0] + grid[i][0];
            for (int j = 1; j != N; ++j) {
                ret[j] = grid[i][j] + min(ret[j], ret[j - 1]);
            }
        }

        return ret[N - 1];
    }
};
```

初始化时需要设置为 `INT_MAX`，便于 `i = 0` 时取 `ret[j]`。

Unique Paths

Source

- lintcode: [\(114\) Unique Paths](#)

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram).

The robot can only move either down or right at any point in time. The robot is trying to

How many possible unique paths are there?

Note

m and n will be at most 100.

题解

题目要求：给定 $m \times n$ 矩阵，求左上角到右下角的路径总数，每次只能向左或者向右前进。按照动态规划中矩阵类问题的通用方法：

1. State: $f[m][n]$ 从起点到坐标 (m,n) 的路径数目
2. Function: $f[m][n] = f[m-1][n] + f[m][n-1]$ 分析终点与左边及右边节点的路径数，发现从左边或者右边到达终点的路径一定不会重合，相加即为唯一的路径总数
3. Initialization: $f[i][j] = 1$, 到矩阵中任一节点均至少有一条路径，其实关键之处在于给第0行和第0列初始化，免去了单独遍历第0行和第0列进行初始化
4. Answer: $f[m-1][n-1]$

C++

```
class Solution {
public:
    /**
     * @param n, m: positive integer (1 <= n, m <= 100)
     * @return an integer
     */
    int uniquePaths(int m, int n) {
        if (m < 1 || n < 1) {
            return 0;
        }

        vector<vector<int>> > ret(m, vector<int>(n, 1));

        for (int i = 1; i != m; ++i) {
            for (int j = 1; j != n; ++j) {
                ret[i][j] = ret[i-1][j] + ret[i][j-1];
            }
        }
    }
}
```

```

        return ret[m - 1][n - 1];
    }
};

```

源码分析

1. 异常处理，虽然题目有保证为正整数，但还是判断一下以防万一
2. 初始化二维矩阵，值均为1
3. 按照转移矩阵函数进行累加
4. 任何 `ret[m - 1][n - 1]`

Unique Paths II

Source

- lintcode: [\(115\) Unique Paths II](#)

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

Note

m and n will be at most 100.

Example

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

```

[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]

```

The total number of unique paths is 2.

题解

在上题的基础上加了obstacle这么一个限制条件，那么也就意味着凡是遇到障碍点，其路径数马上变为0，需要注意的是初始化环节和上题有较大不同。首先来看看错误的初始化实现。

C++ initialization error

```

class Solution {
public:
    /**

```

```

* @param obstacleGrid: A list of lists of integers
* @return: An integer
*/
int uniquePathsWithObstacles(vector<vector<int>> &obstacleGrid) {
    if(obstacleGrid.empty() || obstacleGrid[0].empty()) {
        return 0;
    }

    const int M = obstacleGrid.size();
    const int N = obstacleGrid[0].size();

    vector<vector<int>> > ret(M, vector<int>(N, 0));

    for (int i = 0; i != M; ++i) {
        if (0 == obstacleGrid[i][0]) {
            ret[i][0] = 1;
        }
    }
    for (int i = 0; i != N; ++i) {
        if (0 == obstacleGrid[0][i]) {
            ret[0][i] = 1;
        }
    }

    for (int i = 1; i != M; ++i) {
        for (int j = 1; j != N; ++j) {
            if (obstacleGrid[i][j]) {
                ret[i][j] = 0;
            } else {
                ret[i][j] = ret[i - 1][j] + ret[i][j - 1];
            }
        }
    }

    return ret[M - 1][N - 1];
}
};

```

源码分析

错误之处在于初始化第0行和第0列时，未考虑到若第0行/列有一个坐标出现障碍物，则当前行/列后的元素路径数均为0！

C++

```

class Solution {
public:
    /**
     * @param obstacleGrid: A list of lists of integers
     * @return: An integer
     */
    int uniquePathsWithObstacles(vector<vector<int>> &obstacleGrid) {
        if(obstacleGrid.empty() || obstacleGrid[0].empty()) {
            return 0;
        }
    }
}

```



```

const int M = obstacleGrid.size();
const int N = obstacleGrid[0].size();

vector<vector<int>> > ret(M, vector<int>(N, 0));

for (int i = 0; i != M; ++i) {
    if (obstacleGrid[i][0]) {
        break;
    } else {
        ret[i][0] = 1;
    }
}
for (int i = 0; i != N; ++i) {
    if (obstacleGrid[0][i]) {
        break;
    } else {
        ret[0][i] = 1;
    }
}

for (int i = 1; i != M; ++i) {
    for (int j = 1; j != N; ++j) {
        if (obstacleGrid[i][j]) {
            ret[i][j] = 0;
        } else {
            ret[i][j] = ret[i - 1][j] + ret[i][j - 1];
        }
    }
}

return ret[M - 1][N - 1];
};

```

源码分析

1. 异常处理
2. 初始化二维矩阵(全0阵)，尤其注意遇到障碍物时应 `break` 跳出当前循环
3. 递推路径数
4. 返回`ret[M - 1][N - 1]`

Sequence

本节主要总结序列类动态规划题，按照动态规划的四要素，此类题可从以下四个角度分析。

1. State: $f[i]$ 前 i 个位置/数字/字母...
2. Function: $f[i] = f[i-1]$... 找递推关系
3. Initialization: 根据题意进行必要的初始化
4. Answer: $f[n-1]$

Climbing Stairs

Source

- lintcode: [\(111\) Climbing Stairs](#)

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example

Given an example $n=3$, $1+1+1=2+1=1+2=3$

return 3

题解

题目问的是到达顶端的方法数，我们采用序列类问题的通用分析方法，可以得到如下四要素：

1. State: $f[i]$ 爬到第 i 级的方法数
2. Function: $f[i]=f[i-1]+f[i-2]$
3. Initialization: $f[0]=1, f[1]=1$
4. Answer: $f[n]$

尤其注意状态转移方程的写法， $f[i]$ 只可能由两个中间状态转化而来，一个是 $f[i-1]$ ，由 $f[i-1]$ 到 $f[i]$ 其方法总数并未增加；另一个是 $f[i-2]$ ，由 $f[i-2]$ 到 $f[i]$ 隔了两个台阶，因此有1+1和2两个方法，因此容易写成 $f[i]=f[i-1]+f[i-2]+1$ ，但仔细分析后能发现，由 $f[i-2]$ 到 $f[i]$ 的中间状态 $f[i-1]$ 已经被利用过一次，故 $f[i]=f[i-1]+f[i-2]$

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 $O(n)$ ，仔细观察后可以发现在状态转移方程中，我们可以使用三个变量来替代长度为 $n+1$ 的数组。具体代码可参考 [climbing-stairs | 九章算法](#)

C++

```

class Solution {
public:
    /**
     * @param n: An integer
     * @return: An integer
     */
    int climbStairs(int n) {
        if (n < 1) {
            return 0;
        }

        int ret0 = 1, ret1 = 1, ret2 = 1;

        for (int i = 2; i != n + 1; ++i) {
            ret0 = ret1 + ret2;
            ret2 = ret1;
            ret1 = ret0;
        }

        return ret0;
    }
};

```

Jump Game

Source

- lintcode:

(116) Jump Game

Given an array of non-negative integers, you are initially positioned at the first index

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Example

A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.

题解(自顶向下-动态规划)

1. State: $f[i]$ 从起点出发能否达到 i
2. Function: $f[i] = \text{OR} (f[j], j < i \ \&\& \ j + A[j] \geq i)$, 状态 j 转移到 i , 所有小于 i 的下标 j 的元素中是否存在能从 j 跳转到 i 得
3. Initialization: $f[0] = \text{true}$;
4. Answer: 递推到第 $N - 1$ 个元素时, $f[N-1]$

这种自顶向下的方法需要使用额外的 $O(n)$ 空间, 保存小于 $N-1$ 时的状态。且时间复杂度在恶劣情况下有可能变为 $1^2 + 2^2 + \dots + n^2 = O(n^3)$, 出现 TLE 无法AC的情况, 不过工作面试能给出这种动规的实现就挺好的了。

C++ from top to bottom

```
class Solution {
public:
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    bool canJump(vector<int> A) {
        if (A.empty()) {
            return true;
        }

        vector<bool> jumpto(A.size(), false);
        jumpto[0] = true;
```

```

        for (int i = 1; i != A.size(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (jumpto[j] && (j + A[j] >= i)) {
                    jumpto[i] = true;
                    break;
                }
            }
        }

        return jumpto[A.size() - 1];
    }
};

```

题解(自底向上-贪心法)

题意为问是否能从起始位置到达最终位置，我们首先分析到达最终位置的条件，从坐标 i 出发所能到达最远的位置为 $f[i] = i + A[i]$ ，如果要到达最终位置，即存在某个 i 使得 $f[i] \geq N - 1$ ，而想到达 i ，则又需存在某个 j 使得 $f[j] \geq i - 1$ 。依此类推直到下标为0。

以下分析形式虽为动态规划，实则贪心法！

1. State: $f[i]$ 从 i 出发能否到达最终位置
2. Function: $f[j] = j + A[j] \geq i$, 状态 j 转移到 i , 置为 true
3. Initialization: 第一个为 true 的元素为 $A.size() - 1$
4. Answer: 递推到第 0 个元素时，若其值为 true 返回 true

C++ greedy, from bottom to top

```

class Solution {
public:
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    bool canJump(vector<int> A) {
        if (A.empty()) {
            return true;
        }

        int index_true = A.size() - 1;
        for (int i = A.size() - 1; i >= 0; --i) {
            if (i + A[i] >= index_true) {
                index_true = i;
            }
        }

        return 0 == index_true ? true : false;
    }
};

```

题解(自顶向下-贪心法)

针对上述自顶向下可能出现时间复杂度过高的情况，下面使用贪心思想对*i*进行递推，每次遍历A中的一个元素时更新最远可能到达的元素，最后判断最远可能到达的元素是否大于 `A.size() - 1`

C++ greedy, from top to bottom

```
class Solution {
public:
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    bool canJump(vector<int> A) {
        if (A.empty()) {
            return true;
        }

        int farthest = A[0];

        for (int i = 1; i != A.size(); ++i) {
            if ((i <= farthest) && (i + A[i] > farthest)) {
                farthest = i + A[i];
            }
        }

        return farthest >= A.size() - 1;
    }
};
```

Jump Game II

Source

- lintcode: [\(117\) Jump Game II](#)

Given an array of non-negative integers, you are initially positioned at the first index

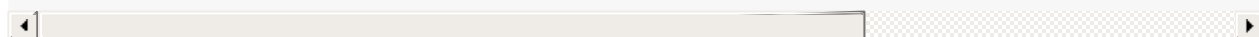
Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

Example

Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1,



题解(自顶向下-动态规划)

首先来看看使用动态规划的解法，由于复杂度较高在A元素较多时会出现TLE，因为时间复杂度接近

$O(n^3)$. 工作面试中给出动规的实现就挺好的。

1. State: $f[i]$ 从起点跳到这个位置最少需要多少步
2. Function: $f[i] = \text{MIN}(f[j]+1, j < i \ \&\& \ j + A[j] \geq i)$ 取出所有能从j到i中的最小值
3. Initialization: $f[0] = 0$, 即一个元素时不需移位即可到达
4. Answer: $f[n-1]$

C++ Dynamic Programming

```
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return -1;
        }

        const int N = A.size() - 1;
        vector<int> steps(N, INT_MAX);
        steps[0] = 0;

        for (int i = 1; i != N + 1; ++i) {
            for (int j = 0; j != i; ++j) {
                if ((steps[j] != INT_MAX) && (j + A[j] >= i)) {
                    steps[i] = steps[j] + 1;
                    break;
                }
            }
        }

        return steps[N];
    }
};
```

源码分析

状态转移方程为

```
if ((steps[j] != INT_MAX) && (j + A[j] >= i)) {
    steps[i] = steps[j] + 1;
    break;
}
```

其中break即体现了MIN操作，最开始满足条件的j即为最小步数。

题解(贪心法-自底向上)

使用动态规划解Jump Game的题复杂度均较高，这里可以使用贪心法达到线性级别的复杂度。

贪心法可以使用自底向上或者自顶向下，首先看看我最初使用自底向上做的。对A数组遍历，找到最小的下标 `min_index`，并在下一轮中用此 `min_index` 替代上一次的 `end`，直至 `min_index` 为0，返回最小跳数 `jumps`。以下的实现有个 bug，细心的你能发现吗？

C++ greedy from bottom to top, bug version

```
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return -1;
        }

        const int N = A.size() - 1;
        int jumps = 0;
        int last_index = N;
        int min_index = N;

        for (int i = N - 1; i >= 0; --i) {
            if (i + A[i] >= last_index) {
                min_index = i;
            }

            if (0 == min_index) {
                return ++jumps;
            }

            if ((0 == i) && (min_index < last_index)) {
                ++jumps;
                last_index = min_index;
                i = last_index - 1;
            }
        }

        return jumps;
    }
};
```

源码分析

使用jumps记录最小跳数，last_index记录离终点最远的坐标，min_index记录此次遍历过程中找到的最小下标。

以上的bug在于当min_index为1时，i = 0, for循环中仍有--i，因此退出循环，无法进入 `if (0 == min_index)` 语句，因此返回的结果会小1个。

C++ greedy, from bottom to top

```

class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return 0;
        }

        const int N = A.size() - 1;
        int jumps = 0, end = N, min_index = N;

        while (end > 0) {
            for (int i = end - 1; i >= 0; --i) {
                if (i + A[i] >= end) {
                    min_index = i;
                }
            }

            if (min_index < end) {
                ++jumps;
                end = min_index;
            } else {
                // cannot jump to the end
                return -1;
            }
        }

        return jumps;
    }
};

```

源码分析

之前的 bug version 代码实在是太丑陋了，改写了个相对优雅的实现，加入了是否能到达终点的判断。在更新 `min_index` 的内循环中也可改为如下效率更高的方式：

```

        for (int i = 0; i != end; ++i) {
            if (i + A[i] >= end) {
                min_index = i;
                break;
            }
        }

```

题解(贪心法-自顶向下)

看过了自底向上的贪心法，我们再来瞅瞅自顶向下的实现。自顶向下使用 `farthest` 记录当前坐标出发能到达的最远坐标，遍历当前 `start` 与 `end` 之间的坐标，若 `i+A[i] > farthest` 时更新 `farthest` (寻找最小跳数)，当前循环遍历结束时递推 `end = farthest`。当 `end >= A.size() - 1` 时退出循环，返回最小跳数。

C++

```

/**
 * http://www.jiuzhang.com/solutions/jump-game-ii/
 */
class Solution {
public:
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    int jump(vector<int> A) {
        if (A.empty()) {
            return 0;
        }

        const int N = A.size() - 1;
        int start = 0, end = 0, jumps = 0;

        while (end < N) {
            int farthest = end;
            for (int i = start; i <= end; ++i) {
                if (i + A[i] >= farthest) {
                    farthest = i + A[i];
                }
            }

            if (end < farthest) {
                ++jumps;
                start = end + 1;
                end = farthest;
            } else {
                // cannot jump to the end
                return -1;
            }
        }

        return jumps;
    }
};

```

術語表

TLE

Time Limit Exceeded 的简称。你的程序在 OJ 上的运行时间太长了，超过了对应题目的时间限制。

[0. leetcode/lintcode题解/算法学习 笔记](#) [15.4.2. Jump Game](#) [15.3.1. Minimum Path Sum](#)
[7.2. Zero Sum Subarray](#) [10.12. Merge k Sorted Lists](#) [10.14. Reorder List](#) [6.4. Anagrams](#)