# Algorithm

http://algorithm.yuanbin.me

# 目錄


Preface 1.1
FAQ 1.2
- Guidelines for Contributing 1.2.1
- Contributors 1.2.2

Part I - Basics 1.3
Basics Data Structure 1.4
- String 1.4.1
- Linked List 1.4.2
- Binary Tree 1.4.3
- Huffman Compression 1.4.4
- Queue 1.4.5
- Heap 1.4.6
- Stack 1.4.7
- Set 1.4.8
- Map 1.4.9
- Graph 1.4.10

Basics Sorting 1.5
- Bubble Sort 1.5.1
- Selection Sort 1.5.2
- Insertion Sort 1.5.3
- Merge Sort 1.5.4
- Quick Sort 1.5.5
- Heap Sort 1.5.6
- Bucket Sort 1.5.7
- Counting Sort 1.5.8
- Radix Sort 1.5.9

Basics Algorithm 1.6
- Divide and Conquer 1.6.1
- Binary Search 1.6.2
- Math 1.6.3
  - Greatest Common Divisor 1.6.3.1
  - Prime 1.6.3.2
- Knapsack 1.6.4

Counting Problem 1.6.5
Probability 1.6.6
Shuffle 1.6.6.1
Bitmap 1.6.7
Basics Misc 1.7
Bit Manipulation 1.7.1
Part II - Coding 1.8
String 1.9
strStr 1.9.1
Two Strings Are Anagrams 1.9.2
Compare Strings 1.9.3
Anagrams 1.9.4
Longest Common Substring 1.9.5
Rotate String 1.9.6
Reverse Words in a String 1.9.7
Valid Palindrome 1.9.8
Longest Palindromic Substring 1.9.9
Space Replacement 1.9.10
Wildcard Matching 1.9.11
Length of Last Word 1.9.12
Count and Say 1.9.13
Integer Array 1.10
Remove Element 1.10.1
Zero Sum Subarray 1.10.2
Subarray Sum K 1.10.3
Subarray Sum Closest 1.10.4
Recover Rotated Sorted Array 1.10.5
Product of Array Exclude Itself 1.10.6
Partition Array 1.10.7
First Missing Positive 1.10.8
2 Sum 1.10.9
3 Sum 1.10.10
3 Sum Closest 1.10.11
Remove Duplicates from Sorted Array 1.10.12
Remove Duplicates from Sorted Array II 1.10.13
Merge Sorted Array 1.10.14

Merge Sorted Array II 1.10.15
Median 1.10.16
Partition Array by Odd and Even 1.10.17
Kth Largest Element 1.10.18
Binary Search 1.11
Binary Search 1.11.1
Search Insert Position 1.11.2
Search for a Range 1.11.3
First Bad Version 1.11.4
Search a 2D Matrix 1.11.5
Search a 2D Matrix II 1.11.6
Find Peak Element 1.11.7
Search in Rotated Sorted Array 1.11.8
Search in Rotated Sorted Array II 1.11.9
Find Minimum in Rotated Sorted Array 1.11.10
Find Minimum in Rotated Sorted Array II 1.11.11
Median of two Sorted Arrays 1.11.12
Sqrt x 1.11.13
Wood Cut 1.11.14
Math and Bit Manipulation 1.12
Single Number 1.12.1
Single Number II 1.12.2
Single Number III 1.12.3
O1 Check Power of 2 1.12.4
Convert Integer A to Integer B 1.12.5
Factorial Trailing Zeroes 1.12.6
Unique Binary Search Trees 1.12.7
Update Bits 1.12.8
Fast Power 1.12.9
Hash Function 1.12.10
Count 1 in Binary 1.12.11
Fibonacci 1.12.12
A plus B Problem 1.12.13
Print Numbers by Recursion 1.12.14
Majority Number 1.12.15
Majority Number II 1.12.16

Majority Number III 1.12.17
Digit Counts 1.12.18
Ugly Number 1.12.19
Plus One 1.12.20
Linked List 1.13
Remove Duplicates from Sorted List 1.13.1
Remove Duplicates from Sorted List II 1.13.2
Remove Duplicates from Unsorted List 1.13.3
Partition List 1.13.4
Add Two Numbers 1.13.5
Two Lists Sum Advanced 1.13.6
Remove Nth Node From End of List 1.13.7
Linked List Cycle 1.13.8
Linked List Cycle II 1.13.9
Reverse Linked List 1.13.10
Reverse Linked List II 1.13.11
Merge Two Sorted Lists 1.13.12
Merge k Sorted Lists 1.13.13
Reorder List 1.13.14
Copy List with Random Pointer 1.13.15
Sort List 1.13.16
Insertion Sort List 1.13.17
Palindrome Linked List 1.13.18
Delete Node in the Middle of Singly Linked List 1.13.19
LRU Cache 1.13.20
Rotate List 1.13.21
Swap Nodes in Pairs 1.13.22
Remove Linked List Elements 1.13.23
Binary Tree 1.14
Binary Tree Preorder Traversal 1.14.1
Binary Tree Inorder Traversal 1.14.2
Binary Tree Postorder Traversal 1.14.3
Binary Tree Level Order Traversal 1.14.4
Binary Tree Level Order Traversal II 1.14.5
Maximum Depth of Binary Tree 1.14.6
Balanced Binary Tree 1.14.7

Binary Tree Maximum Path Sum 1.14.8
Lowest Common Ancestor 1.14.9
Invert Binary Tree 1.14.10
Diameter of a Binary Tree 1.14.11
Construct Binary Tree from Preorder and Inorder Traversal 1.14.12
Construct Binary Tree from Inorder and Postorder Traversal 1.14.13
Subtree 1.14.14
Binary Tree Zigzag Level Order Traversal 1.14.15
Binary Tree Serialization 1.14.16
Binary Search Tree 1.15
Insert Node in a Binary Search Tree 1.15.1
Validate Binary Search Tree 1.15.2
Search Range in Binary Search Tree 1.15.3
Convert Sorted Array to Binary Search Tree 1.15.4
Convert Sorted List to Binary Search Tree 1.15.5
Binary Search Tree Iterator 1.15.6
Exhaustive Search 1.16
Subsets 1.16.1
Unique Subsets 1.16.2
Permutations 1.16.3
Unique Permutations 1.16.4
Next Permutation 1.16.5
Previous Permuation 1.16.6
Permutation Index 1.16.7
Permutation Index II 1.16.8
Permutation Sequence 1.16.9
Unique Binary Search Trees II 1.16.10
Palindrome Partitioning 1.16.11
Combinations 1.16.12
Combination Sum 1.16.13
Combination Sum II 1.16.14
Minimum Depth of Binary Tree 1.16.15
Word Search 1.16.16
Dynamic Programming 1.17
Triangle 1.17.1
Backpack 1.17.2

Backpack II 1.17.3
Minimum Path Sum 1.17.4
Unique Paths 1.17.5
Unique Paths II 1.17.6
Climbing Stairs 1.17.7
Jump Game 1.17.8
Word Break 1.17.9
Longest Increasing Subsequence 1.17.10
Palindrome Partitioning II 1.17.11
Longest Common Subsequence 1.17.12
Edit Distance 1.17.13
Jump Game II 1.17.14
Best Time to Buy and Sell Stock 1.17.15
Best Time to Buy and Sell Stock II 1.17.16
Best Time to Buy and Sell Stock III 1.17.17
Best Time to Buy and Sell Stock IV 1.17.18
Distinct Subsequences 1.17.19
Interleaving String 1.17.20
Maximum Subarray 1.17.21
Maximum Subarray II 1.17.22
Longest Increasing Continuous subsequence 1.17.23
Longest Increasing Continuous subsequence II 1.17.24
Egg Dropping Puzzle 1.17.25
Maximal Square 1.17.26
Graph 1.18
Find the Connected Component in the Undirected Graph 1.18.1
Route Between Two Nodes in Graph 1.18.2
Topological Sorting 1.18.3
Word Ladder 1.18.4
Bipartial Graph Part I 1.18.5
Data Structure 1.19
Implement Queue by Two Stacks 1.19.1
Min Stack 1.19.2
Sliding Window Maximum 1.19.3
Longest Words 1.19.4
Heapify 1.19.5

Kth Smallest Number in Sorted Matrix 1.19.6
Problem Misc 1.20
Nuts and Bolts Problem 1.20.1
String to Integer 1.20.2
Insert Interval 1.20.3
Merge Intervals 1.20.4
Minimum Subarray 1.20.5
Matrix Zigzag Traversal 1.20.6
Valid Sudoku 1.20.7
Add Binary 1.20.8
Reverse Integer 1.20.9
Gray Code 1.20.10
Find the Missing Number 1.20.11
N Queens 1.20.12
N Queens II 1.20.13
Minimum Window Substring 1.20.14
Continuous Subarray Sum 1.20.15
Continuous Subarray Sum II 1.20.16
Longest Consecutive Sequence 1.20.17
Part III - Contest 1.21
Google APAC 1.22
APAC 2015 Round B 1.22.1
Problem A. Password Attacker 1.22.1.1
APAC 2016 Round D 1.22.2
Problem A. Dynamic Grid 1.22.2.1
Microsoft 1.23
Microsoft 2015 April 1.23.1
Problem A. Magic Box 1.23.1.1
Problem B. Professor Q's Software 1.23.1.2
Problem C. Islands Travel 1.23.1.3
Problem D. Recruitment 1.23.1.4
Microsoft 2015 April 2 1.23.2
Problem A. Lucky Substrings 1.23.2.1
Problem B. Numeric Keypad 1.23.2.2
Problem C. Spring Outing 1.23.2.3
Microsoft 2015 September 2 1.23.3

Problem A. Farthest Point 1.23.3.1
Appendix I Interview and Resume 1.24
Interview 1.24.1
Resume 1.24.2
Appendix II System Design 1.25
The System Design Process 1.25.1
Statistics 1.25.2
System Architecture 1.25.3
Scalability 1.25.4
Tags 1.26

# 資料結構與演算法/leetcode/lintcode題解

build passing slack 4/350 chat on slack

- English via Data Structure and Algorithm notes
- 简体中文请戳 数据结构与算法/leetcode/lintcode题解
- 繁體中文請瀏覽 資料結構與演算法/leetcode/lintcode題解

## 簡介

本文檔為資料結構和演算法學習筆記，全文大致分為以下三大部分：

1. Part I為資料結構和演算法基礎，介紹一些基礎的排序/鏈表/基礎演算法
2. Part II為 OJ 上的程式設計題目實戰，按題目的內容分章節編寫，主要來源為 https://leetcode.com/ 和 http://www.lintcode.com/.
3. Part III 為附錄部分，包含如何寫履歷和其他附加資料

本文參考了很多教材和部落格，凡參考過的幾乎都給出明確超連結，如果不小心忘記了，請不要吝惜你的評論和issue :)

本項目保管在 https://github.com/billryan/algorithm-exercise 由 Gitbook 渲染生成 HTML 頁面。你可以在 GitHub(不是 Gitbook) 中 star 該項目查看更新，也可以訂閱 https://ds-algo.slack.com/messages/github_commit/ 中的 `#github_commit` channel 在郵件中查看更新細節。~~RSS 種子功能正在開發中。~~

Slack 的自助邀請註冊功能已啟用，訪問 http://slackin4ds-algo.herokuapp.com 即刻開啟~

你可以線上或者離線查看/搜索本文檔，以下方式任君選擇~

- 線上閱讀(由 Gitbook 渲染) http://algorithm.yuanbin.me
- 離線閱讀: 推送到GitHub後會觸發 travis-ci 的編譯，相應的部分編譯輸出提供七牛的靜態文件加速下載。
    1. EPUB. GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - 適合在 iPhone/iPad/MAC 上離線查看，實測效果極好。
    2. PDF. GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - 推薦下載適合電子屏閱讀的版本，Gitbook 官方使用的中文字體有點問題。
    3. MOBI. GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - Kindle 專用. 未測試，感覺不適合在 Kindle 上看此類書籍，儘管 Kindle 的屏幕對眼睛很好...
- Google 站內搜索: `keywords site:algorithm.yuanbin.me`
- Swiftype 站內搜索: 可使用網頁右下方的 `Search this site` 進行站內搜索

## 授權條款

本作品採用 **創用CC 姓名標示-相同方式分享 4.0 國際許可協議** 進行許可。**傳播此文檔時請注意遵循以上許可協議。** 關於本授權的更多詳情可參考 http://creativecommons.org/licenses/by-sa/4.0/

本著獨樂樂不如眾樂樂的開源精神，我將自己的演算法學習筆記公開和小夥伴們討論，希望高手們不吝賜教。

## 多國文字

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

## 如何貢獻

如果你發現任何有錯誤的地方或是想更新/翻譯本文檔，請毫不猶豫地猛點擊 貢獻指南.

## 如何練習演算法

雖說練習演算法偏向於演算法本身，但是好的程式碼風格還是很有必要的。粗略可分為以下幾點：

- 程式碼可為三大塊：異常處理（空串和邊界處理），主體，返回
- 程式碼風格(**可參考Google的程式設計語言規範**)
    1. 變量名的命名(有意義的變數名)
    2. 縮排(語句塊)
    3. 空格(運算子兩邊)
    4. 程式碼可讀性(即使if語句只有一句也要加花括號)
- 《Code Complete》中給出的參考

而對於實戰演算法的過程中，我們可以採取如下策略：

1. 總結歸類相似題目
2. 找出適合同一類題目的模板程序
3. 對基礎題熟練掌握

以下整理了一些最近練習演算法的網站資源，和大家共享之。

## 線上OJ及部分題解

- LeetCode Online Judge - 找工作方面非常出名的一個OJ，每道題都有 discuss 頁面，可以看別人分享的程式碼和討論，很有參考價值，相應的題解非常多。不過線上程式碼編輯框不太好用，寫著寫著框就拉下來了，最近評測速度比 lintcode 快很多，而且做完後可以看自己程式碼的運行時間分布，首推此 OJ 刷面試相關的題。
- LintCode | Coding interview questions online training system - 和leetcode類似的在線OJ，但是篩選和寫程式碼時比較方便，左邊為題目，右邊為程式碼框。還可以在 `source` 處選擇 CC150 或者其他來源的題。會根據系統locale選擇中文或者英文，可以拿此 OJ 輔助 leetcode 進行練習。

- LeetCode題解 - GitBook - 題解部分十分詳細，比較容易理解，但部分題目不全。
- FreeTymeKiyan/LeetCode-Sol-Res - Clean, Understandable Solutions and Resources on LeetCode Online Judge Algorithms Problems.
- soulmachine/leetcode - 含C++和Java兩個版本的題解。
- Woodstock Blog - IT，演算法及面試。有知識點及類型題總結，特別贊。
- ITint5 | 專注於IT面試 - 文章品質很高，也有部分公司面試題評測。
- Acm之家,專業的ACM學習網站 - 各類題解
- 牛客網-專業IT筆試面試備考平台,最全求職題庫,全面提升IT程式設計能力 - 中國一個IT求職方面的綜合性網站，比較適合想在中國求職的看看。感謝某位美女的推薦 :)

## 其他資源

- 九章算法 - 程式碼品質不錯，整理得也很好。
- 七月算法 - julyedu.com - july大神主導的在線演算法輔導。
- 刷題 | 一畝三分地論壇 - 時不時就會有驚喜放出。
- VisuAlgo - visualising data structures and algorithms through animation - 相當猛的資料結構和演算法可視化。
- Data Structure Visualization - 同上，非常好的動畫示例！！涵蓋了常用的各種資料結構/排序/演算法。
- 結構之法 算法之道 - 不得不服！
- julycoding/The-Art-Of-Programming-By-July - 程序員面試藝術的電子版
- 程序員面試、算法研究、程式設計藝術、紅黑樹、數據挖掘5大系列集錦
- 專欄：算法筆記——《算法設計與分析》 - CSDN上對《算法設計與分析》一書的學習筆記。
- 我的算法學習之路 - Lucida - Google 工程師的演算法學習經驗分享。

## 書籍推薦

本節後三項參考自九章微信分享，謝過。

- Algorithm Design (豆瓣)
- The Algorithm Design Manual, 作者還放出了自己上課的影片和slides - Skiena's Audio Lectures，The Algorithm Design Manual (豆瓣)
- 大部頭有 *Introduction to Algorithm* 和 TAOCP
- *Cracking The Coding Interview*. 著名的CTCI(又稱CC150)，Google, Mircosoft, LinkedIn 前HR離職之後寫的書，從很全面的角度剖析了面試的各個環節和題目。除了演算法資料結構等題以外，還包含OO Design, Database, System Design, Brain Teaser等類型的題目。準備北美面試的同學一定要看。
- 劍指*Offer*。適合中國找工作的同學看看，英文版叫Coding Interviews. 作者是何海濤(Harry He)。Amazon.cn上可以買到。有大概50多題，題目的分析比較全面，會從面試官的角度給出很多的建議和show各種坑。
- 進軍矽谷 -- 程序員面試揭秘。有差不多150題。

## 學習資源推薦(繁體中文譯者)

## 入門

- Data Structures and Algorithms in C++ -by Michael T. Goodrich, Roberto Tamassia and David M. Mount

  台大資工系的**資料結構與演算法**上課用書，內容好懂易讀，習題量大且深度廣度兼具，程式碼風格俐落而不失功能完整性，對C++背景的同學來說是良好的資料結構入門書。

- Data Structures • 數據結構(MOOC)

  北京清華大學的鄧俊輝老師開設的中文MOOC，以C++為主要的程式語言，對於一上來就看書覺得枯燥的同學是一帖入門良藥，講解深入淺出，投影片視覺化做得極好，程式作業禁用了部分STL如vector、list、set等，要求學生必須自己實現需要用的資料結構，程式作業使用清華自建的OJ平台，可以同時跟其他線上學習的同學競爭，作業表現優良的同學還可以加入清華內部的討論群組與清華的學生切磋，相當受用。

## 進階

# FAQ - Frequently Asked Question

Some guidelines for contributing and other questions are listed here.

## How to Contribute?

- Access Guidelines for Contributing for details.

# Guidelines for Contributing

- Access English via Guidelines for Contributing
- 繁體中文請移步 貢獻指南
- 简体中文请移步 贡献指南

除去 FAQ 中提到的兩種輕量級貢獻方法外，你還可以採用 git 這種分佈式協作工具一起改進這個文檔。

如果你不確定自己是否會貢獻比較多的內容，那麼在 GitHub 上 fork 後發 Pull Reqeust 就好了。如果你想成爲 Collaborators 貢獻大量內容, 那麼請大膽發郵件到(yuanbin2014(at)gmail.com)，大歡迎~

總結一下 git 的工作流程就是：

1. 從遠端更新 - `git pull origin master`
2. commit 本機更改 - `git commit -a -m 'xxx'`
3. 推送回遠端 - `git push origin master`

有些時候在 commit 之前可能會忘記 pull, 那麼此時 pull 將會產生一個 merge commit, 這顯然是不太優雅的，建議使用 `git rebase -i` 解決。

git 的簡明教學可參考 b哥的 Git Manual, 小清新極簡教程可參考 git - the simple guide - no deep shit!, rebase 的使用可參考 1, 2, 3

既然涉及到文檔合作，那麼最好是能有個像樣的文檔規範之類的東西方便大家更好的合(ㄐㄧㄠ)作(ㄐㄧ`)，目前想到的有如下幾點。

## 更新/翻譯特定語言

Gitbook 支持多語言書寫，具體通過根目錄下的 `LANGs.md` 目錄指定，目前根目錄下有 `en` , `zh-hans` , `zh-tw` 三個子文件夾分別用於三種語言的書寫，每個子文件夾相當於一個單獨的 Gitbook, 與其他語言的文檔是獨立的，所以更新時只需在各自語言的目錄下工作就好了。各語言的 SUMMARY.md 文件內容保持一致，且均使用英文。

## 目錄生成

Gitbook 中使用 `SUMMARY.md` 這個文件控制生成目錄，添加新內容時最好使用 Gitbook 家自帶的編輯器添加，這樣省事一點點。

## 文檔格式及編輯工具 - GFM && kramdown Markdown

使用markdown編寫，只使用 Gitbook 支援的 markdown 語法。Gitbook 底層的 markdown renderer 爲改動的 kramdown，並增加了GFM支援, 支援的擴充 markdown 語法算是非常多了，具體特性詳見 GitbookIO/kramed

推薦的 markdown 編輯器爲 Gitbook 自家的 editor, 目前新版的 bug 太多，而且是自動 commit 的，不便於版本控制，希望他們後續能改進。所以目前推薦老版，老版的見 editor-lagecy, 支持 Windows/Linux/MAC 三大平臺，業界良心！但是實測在Arch Linux/OSX 下可能會出現佔用記憶體/CPU 過高的情況... 編輯界面如下圖所示，最左邊爲章節預覽，中間爲 markdown 編輯框，右邊爲實時render 頁面，可選擇使用全屏模式。

使用其他如 Mou/Vim/Emacs/Sublime Text也不錯，但是在新增Chapter/Section時就比較麻煩了，嗯，你也可以新建 Section 後再使用其他編輯器編輯。

對 Gitbook 不熟的建議看看 Gitbook Documentation，有助於瞭解 http://algorithm.yuanbin.me 網頁上的文字及各章節等是如何編輯及render的。

# 章節名及編號

章節等文件名全部採用英文，子章節最多到三級，章節編號無需操心，這種瑣事交給 Gitbook 去做就好，如果一定要手動調整，修改 `SUMMARY.md` 文件，注意其中的縮排關係，Gitbook就依靠這個自動給章節編號了。

舉個例子，我現在想新增「動態規劃」及其子章節。首先在 Gitbook 頂部menu欄「Book」中找到「Add Chapter」，填入「Dynamic Programming」。好了，在Gitbook左側章節欄中就能看到新生成的「10. Dynamic Programming」了，左鍵單擊，Gitbook 就會生成「dynamic_programming」目錄及本章的說明文件「dynamic_programming/README.md」。如果想在「10. Dynamic Programming」下新增子章節，右鍵單擊，「Add Section」即可，同上，子章節文件名仍然使用英文名，網頁顯示的標題可以通過 rename 更改再加入中文。

嗯，以上步驟均可直接新建文件夾及操作 `SUMMARY.md` 文件完成。

# 正文書寫風格

1. 中英文混排貫穿全文，優雅美觀起見，儘可能在英文單詞前後加空格，這個使用能在輸入法中英文間加入空格功能就好了。
2. 程式碼的函數名或短的程式碼建議使用 `code`
3. 使用空行進行分段，嗯，markdown通用

Part II爲leetcode/lintcode題解，這部分的風格相對容易統一，感覺還不錯的風格 - Distinct Subsequences

大致遵循如下風格：

1. 給出題目鏈接及原文，引用的原文部分簡單起見我對題目使用了blockquote ，具體可參考我的那些markdown文本。
2. 給出自己的題解，儘可能清晰易懂。
3. 給出能AC的code, 如遇TLE或者錯誤的看情況給出錯誤的實現。使用blockquote, 給出語言類別以便highlight。具體可參看原markdown文件。
4. 題解中的核心部分對應的程式碼，程式碼中不能明顯看出來的邏輯和一些程式上常用的技巧。
5. 程式碼順序：Python => C++ => Java 因爲 Python 的程式碼一般最爲簡潔...
6. 如參考了其他資源，儘可能給出有用的參考鏈接，附簡單的說明。

通過github合作時，添加/修改內容時給出能看懂的commit就好了。暫時就想到這麼多，其實沒那麼多講究啦，感覺看著清楚就好，其他想到的再補充。:-)

## 數學公式

其實程式碼裡是用不著寫數學公式的，但是偶爾分析演算法可能會用著，用過 LaTeX 的都知道她生成的數學公式有多優雅，以至於不用她來寫數學公式都有點不舒服...

這個文檔裡對於較複雜的數學公式建議使用 LaTeX, 因爲託管在gitbook上，所以就用了輕量級的katex插件，沒有用重量級的 MathJax。行內(inline)和行間公式都是 兩個$, 區別在於行間公式寫到下一行行首，而行內公式不能寫在行首(廢話...)。katex非常脆弱，對一些高級的 LaTeX 語法不支援，否則無法編譯輸出到網站和pdf，儘量用簡單的 LaTeX 語法或者不用。

## 附件及圖片引用

圖片統一存放在 `images` 目錄下，其他附件存放在 `docs` 目錄下。引用圖片鏈接一般可以通過 `![Caption](../../shared-files/images/xxx.png)` 聲明。

圖片體積太大不利於頁面載入，建議先壓縮後再放入，如果是png圖片可考慮使用 TinyPNG – Compress PNG images while preserving transparency

# Part I - Basics

第一節主要總結一些基礎知識，如基本的資料結構和基礎演算法。

本節主要由以下章節構成。

## Reference

- VisuAlgo - visualising data structures and algorithms through animation - 相當厲害的資料結構和演算法可視化。
- Data Structure Visualization - 非常好的動畫示例！！涵蓋了常用的各種資料結構/排序/演算法。

# Data Structure - 資料結構

本章主要介紹一些基本的資料結構和演算法。

# String 字串

String 相關的題很常出現在面試題中，實際開發也經常用到，這裡總結一下 C++, Java, Python 中字串常用的方法。

## Python

```
s1 = str()
# in python `'''` and `"""` are the same
s2 = "shaunwei" # 'shaunwei'
s2len = len(s2)
# last 3 chars
s2[-3:] # wei
s2[5:8] # wei
s3 = s2[:5] # shaun
s3 += 'wei' # return 'shaunwei'
# list in python is same as ArrayList in java
s2list = list(s3)
# string at index 4
s2[4] # 'n'
# find index at first
s2.index('w')  # return 5, if not found, throw ValueError
s2.find('w') # return 5, if not found, return -1
```

在Python裡面，沒有StringBuffer 或者 StringBuilder。 但是在Python 裡面處理String本身就比較 cheap。

## Java

```
String s1 = new String();
String s2 = "billryan";
int s2Len = s2.length();
s2.substring(4, 8); // return "ryan"
StringBuilder s3 = new StringBuilder(s2.substring(4, 8));
s3.append("bill");
String s2New = s3.toString(); // return "ryanbill"
// convert String to char array
char[] s2Char = s2.toCharArray();
// char at index 4
char ch = s2.charAt(4); // return 'r'
// find index at first
int index = s2.indexOf('r'); // return 4. if not found, return -1
```

StringBuffer 與 StringBuilder, 前者保證執行緒安全(Thread Safety)，後者不是，但單執行緒下效率高一些，一般使用 StringBuilder.

# Linked List - 鏈表

鏈表是線性表(linear list)的一種。線性表是最基本、最簡單、也是最常用的一種資料結構。線性表中數據元素之間的關係是一對一的關系，即除了第一個和最後一個數據元素之外，其它數據元素都是首尾相接的。線性表有兩種儲存方式，一種是順序儲存結構，另一種是鏈式儲存結構。我們常用的陣列(array)就是一種典型的順序儲存結構。

相反，鏈式儲存結構就是兩個相鄰的元素在記憶體中可能不是物理相鄰的，每一個元素都有一個指標，指標一般是儲存著到下一個元素的指標。這種儲存方式的**優點**是已知插入位置時，定點插入和定點刪除的時間複雜度為 O(1)，不會浪費太多記憶體，添加元素的時候才會申請記憶體空間，刪除元素會釋放記憶體空間。缺點是訪問的時間複雜度最壞為 O(n)。

順序表的特性是隨機讀取，也就是循下標訪問(call-by-index)一個元素的時間複雜度是O(1)，鏈式表的特性是插入和刪除的時間複雜度為O(1)。

鏈表就是鏈式儲存的線性表。根據指標域的不同，鏈表分為單向鏈表、雙向鏈表、循環鏈表等等。

## 程式碼實現

### Python

```python
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
```

### C++

```cpp
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int val, ListNode *next=NULL):val(val),next(next){}
};
```

### Java

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

```
}
```

# 鏈表的基本操作

## 反轉單向鏈表(singly linked list)

鏈表的基本形式是：`1 -> 2 -> 3 -> null`，反轉需要變為 `3 -> 2 -> 1 -> null`。這裡要注意：

- 訪問某個節點 curt.next 時，要檢驗 curt 是否為 null。
- 要把反轉後的最後一個節點（即反轉前的第一個節點）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;
}
```

### 雙向鏈表

和單向鏈表的區別在於：雙向鏈表的反轉核心在於 `next` 和 `prev` 域的交換，還需要注意的是目前節點和上一個節點的遞推。

## Python

```
class DListNode:
    def __init__(self, val):
        self.val = val
        self.prev = self.next = null

    def reverse(self, head):
        curt = None
        while head:
            curt = head
            head = curt.next
            curt.next = curt.prev
            curt.prev = head
        return curt
```

## Java

```
class DListNode {
    int val;
    DListNode prev, next;
    DListNode(int val) {
        this.val = val;
        this.prev = this.next = null;
    }
}

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

### 刪除鏈表中的某個節點

刪除鏈表中的某個節點一定需要知道這個點的前繼節點，所以需要一直有指標指向前繼節點。

然後只需要把 `prev -> next = prev -> next -> next` 即可。但是由於鏈表表頭可能在這個過程中產生變化，導致我們需要一些特別的技巧去處理這種情況。就是下面提到的 Dummy Node。

## 鏈表指標的強健性(robustness)

綜合上面討論的兩種基本操作，鏈表操作時的強健性問題主要包含兩個情況：

- 當訪問鏈表中某個節點 curt.next 時，一定要先判斷 curt 是否為 null。
- 全部操作結束後，判斷是否有環；若有環，則置其中一端為 null。

## Dummy Node

Dummy node 是鏈表問題中一個重要的技巧，中文翻譯叫「啞節點」或者「假人頭結點」。

Dummy node 是一個虛擬節點，也可以認為是標竿節點。Dummy node 就是在鏈表表頭 head 前加一個節點指向 head，即 dummy -> head。Dummy node 的使用多針對單向鏈表沒有前向指標的問題，保證鏈表的 head 不會在刪除操作中遺失。除此之外，還有一種用法比較少見，就是使用 dummy node 來進行head的刪除操作，比如 Remove Duplicates From Sorted List II，一般的方法current = current.next 是無法刪除 head 元素的，所以這個時候如果有一個dummy node在head的前面。

所以，當鏈表的 head 有可能變化（被修改或者被刪除）時，使用 dummy node 可以簡化程式碼及很多邊界情況的處理，最終返回 dummy.next 即新的鏈表。

# 快慢指標(fast/slow pointer)

快慢指標也是一個可以用於很多問題的技巧。所謂快慢指標中的快慢指的是指標向前移動的步長，每次移動的步長較大即為快，步長較小即為慢，常用的快慢指標一般是在單向鏈表中讓快指標每次向前移動2，慢指標則每次向前移動1。快慢兩個指標都從鏈表頭開始遍曆，於是快指標到達鏈表末尾的時候慢指標剛好到達中間位置，於是可以得到中間元素的值。快慢指標在鏈表相關問題中主要有兩個應用：

- 快速找出未知長度單向鏈表的中間節點 設置兩個指標 `*fast` 、 `*slow` 都指向單向鏈表的頭節點，其中 `*fast` 的移動速度是 `*slow` 的2倍，當 `*fast` 指向末尾節點的時候， `slow` 正好就在中間了。此方法可以有效避免多次遍歷鏈表
- 判斷單向鏈表是否有環 利用快慢指標的原理，同樣設置兩個指標 `*fast` 、 `*slow` 都指向單向鏈表的頭節點，其中 `*fast` 的移動速度是 `*slow` 的2倍。如果 `*fast = NULL` ，說明該單向鏈表 以 `NULL` 結尾，不是循環鏈表；如果 `*fast = *slow` ，則快指標追上慢指標，說明該鏈表是循環鏈表。

## Python

```
class NodeCircle:
    def __init__(self, val):
        self.val = val
        self.next = None

    def has_circle(self, head):
        slow = head
        fast = head
        while (slow and fast):
            fast = fast.next
            slow = slow.next
            if fast:
                fast = fast.next
            if fast == slow:
                break
        if fast and slow and (fast == slow):
            return True
        else:
            return False
```

# Binary Tree - 二元樹

二元樹是每個節點最多有兩個子樹的樹結構，子樹有左右之分，二元樹常被用於實現**二元搜尋樹(binary search tree)**和**二元堆(binary heap)**。

二元樹的第i層(根結點為第1層，往下遞增)至多有 $2^{i-1}$ 個結點；深度為k的二元樹至多有 $2^k-1$ 個結點；對任何一棵二元樹T，如果其終端結點數為 $n_0$, 度為2的結點數為 $n_2$, 則 $n_0=n_2+1$。

一棵深度為 $k$, 且有 $2^k-1$ 個節點稱之為**滿二元樹**；深度為 $k$，有 $n$ 個節點的二元樹，若且唯若其每一個節點都與深度為 $k$ 的滿二元樹中，序號為 $1$ 至 $n$ 的節點對應時，稱之為**完全二元樹**。完全二元樹中重在節點標號對應。

## 程式實現

### Python

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

### C++

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

### Java

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

# Tree traversal 樹的遍歷

從二元樹的根節點出發，節點的遍歷分為三個主要步驟：對當前節點進行操作（稱為「訪問」節點，或者根節點）、遍歷左邊子節點、遍歷右邊子節點。訪問節點順序的不同也就形成了不同的遍歷方式。需要注意的是樹的遍歷通常使用遞迴的方法進行理解和實現，在訪問元素時也需要使用遞迴的思想去理解。

按照訪問根元素(當前元素)的前後順序，遍歷方式可劃分為如下幾種：

- 深度優先(depth-first)：先訪問子節點，再訪問父節點，最後訪問第二個子節點。根據根節點相對於左右子節點的訪問先後順序又可細分為以下三種方式。
  1. 前序(pre-order)：先根後左再右
  2. 中序(in-order)：先左後根再右
  3. 後序(post-order)：先左後右再根
- 廣度優先(breadth-first)：先訪問根節點，沿著樹的寬度遍歷子節點，直到所有節點均被訪問為止，又稱為層次(level-order)遍歷。

如下圖所示，遍歷順序在右側框中，紅色A為根節點。使用遞迴和整體的思想去分析遍歷順序較為清晰。

二元樹的廣度優先遍歷和樹的前序/中序/後序遍歷不太一樣，前/中/後序遍歷使用遞迴，也就是用堆疊(stack)的思想對二元樹進行遍歷，廣度優先一般使用隊列(queue)的思想對二元樹進行遍歷。

如果已知中序遍歷和前序遍歷或者後序遍歷，那麼就可以完全恢復出原二元樹結構。其中最爲關鍵的是前序遍歷中第一個一定是根，而後序遍歷最後一個一定是根，中序遍歷在得知根節點後又可進一步遞歸得知左右子樹的根節點。但是這種方法也是有適用範圍的：元素不能重複！否則無法完成定位。

## Python

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

class Traversal(object):
    def __init__(self):
```

```
        self.traverse_path = list()

    def preorder(self, root):
        if root:
            self.traverse_path.append(root.val)
            self.preorder(root.left)
            self.preorder(root.right)

    def inorder(self,root):
        if root:
            self.inorder(root.left)
            self.traverse_path.append(root.val)
            self.inorder(root.right)

    def postorder(self,root):
        if root:
            self.postorder(root.left)
            self.postorder(root.right)
            self.traverse_path.append(root.val)
```

這裡只給出簡單的 python 遞迴版實現，C++ 和 Java 的程式碼以及非遞迴版本的實現，敬請期待後續章節。

## 樹類題的複雜度分析

對樹相關的題進行複雜度分析時可統計對每個節點被訪問的次數，進而求得總的時間複雜度。

## Binary Search Tree - 二元搜尋樹

一顆**二元搜尋樹(BST)**是一顆二元樹，其中每個節點都含有一個可進行比較的鍵及相應的值，且每個節點的鍵都**大於等於左子樹中的任意節點的鍵**，而**小於右子樹中的任意節點的鍵**。

使用中序遍歷可得到有序數列，這是二元搜尋樹的又一個重要特徵。

二元搜尋樹使用的每個節點含有**兩個**鏈接，它是將鏈表插入的靈活性和有序陣列查找的高效性結合起來的高效符號表實現。

# Huffman Compression - 霍夫曼壓縮

主要思想：放棄文本文件的普通保存方式：不再使用7位或8位二進制數表示每一個字符，而是**用較少的比特表示出現頻率最高的字符，用較多的比特表示出現頻率低的字符**。

使用變動長度編碼(variable-length code)來表示字串，勢必會導致編解碼時碼字的唯一性問題，因此需要一種編解碼方式唯一的前綴碼(prefix code)，而表示前綴碼的一種簡單方式就是使用單詞搜尋樹，其中最優前綴碼即為Huffman首創。

以符號F, O, R, G, E, T為例，其出現的頻次如以下表格所示。

| Symbol | F | O | R | G | E | T |
|---|---|---|---|---|---|---|
| Frequence | 2 | 3 | 4 | 4 | 5 | 7 |
| Code | 000 | 001 | 100 | 101 | 01 | 11 |

則對各符號進行霍夫曼編碼的動態示例如下圖所示。基本步驟是將出現頻率由小到大排列，組成子樹後頻率相加作為整體再和其他未加入二元樹中的節點頻率比較。加權路徑長為節點的頻率乘以樹的深度。

有關霍夫曼編碼的具體步驟可參考 Huffman 編碼壓縮算法 | 酷 殼 - CoolShell.cn 和 霍夫曼編碼 - 維基百科，自由的百科全書，清晰易懂。

# Queue - 隊列

Queue 是一個 FIFO（First-in First-out, 先進先出）的資料結構，併發(concurrent)中經常使用，可以安全地將對象從一個任務傳給另一個任務。

# 程式碼實現

## Python

Queue 和 Stack 在 Python 中都是用 `list` , `[]` 實現的。 在python 中list是一個dynamic array, 可以通過 `append` 在list的尾部添加元素， 通過 `pop()` 在list的尾部彈出元素實現 `Stack` 的 `FILO` ， 如果是 `pop(0)` 則彈出頭部的元素實現 `Queue` 的 `FIFO` 。

```
queue = []  # same as list()
size = len(queue)
queue.append(1)
queue.append(2)
queue.pop(0) # return 1
queue[0] # return 2 examine the first element
```

### Methods

| \ | methods |
|---|---|
| Insert | queue.append(e) |
| Remove | queue.pop(0) |
| Examine | queue[0] |

## Java

Queue 在 Java 中是 Interface, 一種實現是 LinkedList, LinkedList 向上轉型為 Queue, Queue 通常不能存儲 `null` 元素，否則與 `poll()` 等方法的返回值混淆。

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

### Methods

| 0:0 | Throws exception | Returns special value |
|---|---|---|
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| | | |

| | | |
|---|---|---|
| Examine | element() | peek() |

優先考慮右側方法，右側元素不存在時返回 `null` . 判斷非空時使用 `isEmpty()` 方法，繼承自 Collection.

# Priority Queue - 優先隊列

應用程式常常需要處理帶有優先級的業務，優先級最高的業務首先得到服務。因此優先隊列這種資料結構應運而生。優先隊列中的每個元素都有各自的優先級，優先級最高的元素最先得到服務；優先級相同的元素按照其在優先隊列中的順序得到服務。

優先隊列可以使用陣列或鏈表實現，從時間和空間覆雜度來說，往往用二叉堆(Binary heap)來實現。

## Python

Python 中提供 `heapq` 的lib來實現 priority queue. 提供 `push` 和 `pop` 兩個基本操作和 `heapify` 初始化操作.

| \ | methods | time complexity |
|---|---|---|
| enqueue | heapq.push(queue, e) | $O(\log n)$ |
| dequeue | heapq.pop(queue) | $O(\log n)$ |
| init | heapq.heapify(queue) | $O(n \log n)$ |
| peek | queue[0] | $O(1)$ |

## Java

Java 中提供 `PriorityQueue` 類，該類是 Interface Queue 的另外一種實現，和 `LinkedList` 的區別主要在於排序行為而不是性能，基於 priority heap 實現，非 `synchronized` ，故多執行緒(Multi-thread)下應使用 `PriorityBlockingQueue` . 預設為自然序（小根堆），需要其他排序方式可自行實現 `Comparator` 接口，選用合適的構造器初始化。使用疊代器遍歷時不保證有序，有序訪問時需要使用 `Arrays.sort(pq.toArray())` .

不同方法的時間覆雜度：

- enqueuing and dequeuing: `offer` , `poll` , `remove()` and `add` - $O(\log n)$
- Object: `remove(Object)` and `contains(Object)` - $O(n)$
- retrieval: `peek` , `element` , and `size` - $O(1)$.

# Deque - 雙端隊列

雙端隊列（deque，全名double-ended queue）可以讓你在任何一端添加或者移除元素，因此它是一種具有隊列和堆疊性質的資料結構。

## Python

Python 的 `list` 就可以執行類似於 `deque` 的操作， 但是效率會過於慢。 為了提升數據的處理效率，一些高效的資料結構放在了 `collections` 中。 在 `collections` 中提供了 `deque` 的類， 如果需要多次對 `list` 執行頭尾元素的操作， 請使用 `deque` 。

```
dq = collections.deque();
```

### Methods

| \ | methods | time complexity |
|---|---|---|
| enqueue left | dq.appendleft(e) | $O(1)$ |
| enqueue right | dq.append(e) | $O(1)$ |
| dequeue left | dq.popleft() | $O(1)$ |
| dequeue right | dq.pop() | $O(1)$ |
| peek left | dq[0] | $O(1)$ |
| peek right | dq[-1] | $O(1)$ |

## Java

Java 在1.6之後提供了 Deque 介面，既可使用 `ArrayDeque` （陣列）來實現，也可以使用 `LinkedList` （鏈表）來實現。前者是一個數組外加首尾索引，後者是雙向鏈表。

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

### Methods

| | First Element (Head) | | Last Element (Tail) | |
|---|---|---|---|---|
| | Throws exception | Special value | Throws exception | Special value |
| Insert | $addFirst(e)$ | $offerFirst(e)$ | $addL*(e)$ | $offerL*(e)$ |
| Remove | $removeFirst()$ | $pollFirst()$ | $removeL*()$ | $pollL*()$ |
| Examine | $\geq tFirst()$ | $peekFirst()$ | $\geq tL*()$ | $peekL*()$ |

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

## Reference

- 優先隊列 - 維基百科，自由的百科全書
- 雙端隊列 - 維基百科，自由的百科全書

# Heap - 堆

一般情況下，堆通常指的是**二叉堆**，**二叉堆**是一個近似**完全二元樹**的數據結構，**即披著二元樹羊皮的陣列，**故使用陣列來實現較為便利。子結點的鍵值(key)或索引總是小於（或者大於）它的父節點，且每個節點的左右子樹又是一個**二叉堆**(大根堆(Max Heap)或者小根堆(Min Heap))。根節點最大的堆叫做最大堆或大根堆，根節點最小的堆叫做最小堆或小根堆。**常被用作實現優先隊列(Priority Queue)。**

## 特點

1. **以陣列表示，但是以完全二元樹的方式理解**。
2. 唯一能夠同時最優地利用空間和時間的方法——最壞情況下也能保證使用 $2N\log N$ 次比較和恒定的額外空間。
3. 在索引從0開始的陣列中：
   - 父節點 `i` 的左子節點在位置 `(2*i+1)`
   - 父節點 `i` 的右子節點在位置 `(2*i+2)`
   - 子節點 `i` 的父節點在位置 `floor((i-1)/2)`

## 堆的基本操作

以大根堆為例，堆的常用操作如下。

1. 最大堆調整（Max_Heapify）：將堆的末端子節點作調整，使得子節點永遠小於父節點
2. 創建最大堆（Build_Max_Heap）：將堆所有數據重新排序
3. 堆排序（HeapSort）：移除位在第一個數據的根節點，並做最大堆調整的遞迴運算

其中步驟1是給步驟2和3用的。

6 5 3 1 8 7 2 4

## Python

```python
class MaxHeap:
    def __init__(self, array=None):
        if array:
            self.heap = self._max_heapify(array)
        else:
            self.heap = []

    def _sink(self, array, i):
        # move node down the tree
        left, right = 2 * i + 1, 2 * i + 2
        max_index = i
        if left < len(array) and array[left] > array[max_index]:
            max_index = left
        if right < len(array) and array[right] > array[max_index]:
            max_index = right
        if max_index != i:
            array[i], array[max_index] = array[max_index], array[i]
            self._sink(array, max_index)

    def _swim(self, array, i):
        # move node up the tree
        if i == 0:
            return
        father = (i - 1) / 2
        if array[father] < array[i]:
            array[father], array[i] = array[i], array[father]
            self._swim(array, father)

    def _max_heapify(self, array):
        for i in xrange(len(array) / 2, -1, -1):
            self._sink(array, i)
        return array

    def push(self, item):
        self.heap.append(item)
        self._swim(self.heap, len(self.heap) - 1)

    def pop(self):
        self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]
        item = self.heap.pop()
        self._sink(self.heap, 0)
        return item
```

# Stack - 堆疊

堆疊是一種 LIFO(Last In First Out) 的資料結構，常用方法有添加元素，讀Stack頂元素，彈出(pop) Stack頂元素，判斷堆疊是否為空。

# 程式碼實現

## Python

```
stack = []
len(stack) # size of stack

# more efficient stack
import collections
stack = collections.deque()
```

`list` 作為最基本的 `python` 資料結構之一， 可以很輕鬆地實現 `stack` 。 如果需要更高效的 `stack` ， 建議使用 `deque` 。

### Methods

- `len(stack) != 0` - 判斷 `stack` 是否為空
- `stack[-1]` - 取堆疊頂元素，不移除
- `pop()` - 移除堆疊頂元素並返回該元素
- `append(item)` - 向堆疊頂添加元素

## Java

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

JDK doc 中建議使用 `Deque` 代替 `Stack` 實現堆疊，因為 `Stack` 繼承自 `Vector` ，需要 `synchronized` ，性能略低。

### Methods

- `boolean isEmpty()` - 判斷堆疊是否為空，若使用 Stack 類構造則為 empty()
- `E peek()` - 取堆疊頂元素，不移除
- `E pop()` - 移除堆疊頂元素並返回該元素
- `E push(E item)` - 向堆疊頂添加元素

# Set

Set 是一種用於保存不重複元素的資料結構。常被用作測試歸屬性，故其查找的性能十分重要。

# 程式實現

## Python

`Set` 是 `python` 自帶的基本資料結構， 有多種初始化方式。 `Python` 的 `set` 跟 `dict` 的 Implementation方式類似， 可以認爲 `set` 是只有 `key` 的 `dict` .

```
s = set()
s1 = {1, 2, 3}
s.add('shaunwei')
'shaun' in s  # return true
s.remove('shaunwei')
```

## C++

STL提供的資料結構有 Set 以及 Multiset ，分別提供不重複與重複元素的版本，自C++11以後，STL提供兩種 Set 的實現方式，一個是基於紅-黑樹的 `set` 與 `multiset` ，包含在 `<set>` 標頭檔之中，有序。另一個則是基於湊雜函數的 `unordered_set` 及 `unordered_multiset` ，包含在標頭檔 `<unordered_set>` ，無序。基本的 Set 使用如下所示

```
set<string> s;
s.insert("crossluna");
s.insert("billryan");
auto it = s.find("lucifer");
if(it != s.end()) {
    // "lucifer" found
}
```

另外可以藉由在建構時傳遞自訂的 Functor 、 Hash Function 以達成更彈性的使用，詳細用法及更多的介面請參考 STL 使用文檔。

## Java

Set 與 Collection 具有安全一樣的接口，通常有 `HashSet` , `TreeSet` 或 `LinkedHashSet` 三種實現。 `HashSet` 基於湊雜函數實現，無序，查詢速度最快； `TreeSet` 基於紅-黑樹實現，有序。

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

在不允許重複元素時可當做哈希表來用。

# Map - 關聯容器

Map 是一種關聯數組的資料結構，也常被稱爲字典(dictionary)或鍵值對(key-value pair)。

## 程式實現

### Python

在 Python 中 `dict` (Map) 是一種基本的資料結構。

```
# map 在 python 中是一個keyword
hash_map = {} # or dict()
hash_map['shaun'] = 98
hash_map['wei'] = 99
exist = 'wei' in hash_map  # check existence
point = hash_map['shaun']  # get value by key
point = hash_map.pop('shaun') # remove by key, return value
keys = hash_map.keys()  # return key list
# iterate dictionary(map)
for key, value in hash_map.items():
    # do something with k, v
    pass
```

### C++

與 Set 類似，STL提供了 Map 與 Multimap 兩種，提供同一鍵(key)對應單個或多個值(value)，自C++11以後，一樣提供兩種實現方式，基於紅-黑樹的 `map` 與 `multimap` ，包含在 `<map>` 標頭檔之中，鍵有序。另一個則是基於湊雜函數的 `unordered_map` 及 `unordered_multimap` 包含在標頭檔 `<unordered_map>` ，鍵無序。基本的 Map 使用如下所示

```
map<string, int> mp;
mp ["billryan"] = 69;
mp ["crossluna"] = 159;
auto it = mp.find("billryan");
if(it != mp.end()) {
    // "billryan" found
    cout << mp["billryan"]; // output: 69
}
```

另外可以藉由在建構時傳遞自訂的 Functor 、 Hash Function 以達成更彈性的使用，詳細用法及更多的介面請參考 STL 使用文檔。

### Java

Java 的實現中 Map 是一種將物件與物件相關聯的設計。常用的實現有 `HashMap` 和 `TreeMap` , `HashMap` 被用來快速訪問，而 `TreeMap` 則保證『鍵』始終有序。Map 可以返回鍵的 Set, 值的 Collection, 鍵值對的 Set.

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

# Graph - 圖

圖的表示通常使用**鄰接矩陣和鄰接表**，前者易實現但是對於稀疏矩陣會浪費較多空間，後者使用鏈表的方式存儲資訊但是對於圖搜索時間複雜度較高。

## 程式實現

### 鄰接矩陣 (Adjacency Matrix)

設頂點個數爲 V, 那麼鄰接矩陣可以使用 V × V 的二維陣列來表示。 `g[i][j]` 表示頂點 `i` 和頂點 `j` 的關係，對於無向圖(undirected graph)可以使用0/1表示是否有連接，對於帶有權重的圖則需要使用 `INF` 來區分。有重邊時保存邊數或者權值最大/小的邊即可。

#### Python

```
g = [[0 for _ in range(V)] for _ in range(V)]
```

#### Java

```
/* Java Definition */
int[][] g = new int[V][V];
```

#### C++

```
vector<vector<int>> g (V, vector<int>(V, 0));
```

### 鄰接表 (Adjacency List)

鄰接表通過表示從頂點 `i` 出發到其他所有可能能到的邊。

## 有向圖

#### Python

```
class DirectedGraphNode:
    def __init__(self, x):
        self.label = x
        self.neighbors = []
```

#### Java

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

### C++

```
struct DirectedGraphNode {
    int label;
    vector<DirectedGraphNode*> neighbors;

    DirectedGraphNode(int x): label(x) { }
};
```

## 無向圖同上，只不過在建圖時雙向同時加。

### Python

```
class UndirectedGraphNode:
    def __init__(self, x):
        self.label = x
        self.neighbors = []
```

### Java

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

### C++

```
struct UndirectedGraphNode {
    int label;
    vector<UndirectedGraphNode*> neighbors;
```

```
    UndirectedGraphNode(int x): label(x) { }
};
```

# Basics Sorting - 基礎排序演算法

# 演算法複習——排序

## 演算法分析

1. 時間複雜度-執行時間(比較和交換次數)
2. 空間複雜度-所消耗的額外記憶體空間
    - 使用小堆疊、隊列或表
    - 使用鏈表或指針、數組索引來代表數據
    - 排序數據的副本

在OJ上做題時，一些經驗法則(rule of thumb)以及封底估算(back-of-the-envelop calculation)可以幫助選擇適合的演算法，一個簡單的經驗法則是

```
10^9 operations per second
```

舉例來說，如果今天遇到一個題目，時間限制是1s，但僅有$10^3$筆輸入數據，此時即使使用$O(n^2)$的演算法也沒問題，但若有$10^5$筆輸入，則$O(n^2)$的演算法則非常可能超時，在實作前就要先思考是不是有$O(n\log n)$或更快的演算法。

對具有重鍵的數據(同一組數按不同鍵多次排序)進行排序時，需要考慮排序方法的穩定性，在非穩定性排序演算法中需要穩定性時可考慮加入小索引。

穩定性：如果排序後文件中擁有相同鍵的項的相對位置不變，這種排序方式是穩定的。

常見的排序演算法根據是否需要比較可以分為如下幾類：

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

從穩定性角度考慮可分為如下兩類：

- 穩定
- 非穩定

# Reference

- Sorting algorithm - Wikipedia, the free encyclopedia - 各類排序演算法的「平均、最好、最壞時間複雜度」總結。
- Big-O cheatsheet - 更清晰的總結
- 經典排序演算法總結與實現 | Jark's Blog - 基於 Python 的較為清晰的總結。
- 【面經】矽谷前沿Startup面試經驗-排序演算法總結及快速排序演算法代碼_九章演算法 - 總結了一些常用常問的排序演算法。
- 雷克雅維克大學的程式競賽課程 第一講的slide中提供了演算法分析的經驗法則

# Bubble Sort - 氣泡排序

核心：**氣泡**，持續比較相鄰元素，大的挪到後面，因此大的會逐步往後挪，故稱之為氣泡。

6 5 3 1 8 7 2 4

## Implementation

### Python

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


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

    return alist

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

### Java

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

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

## C++

```
void bubbleSort(vector<int> & arr){
    for(int i = 0; i < arr.size(); i++){
        for(int j = 1; j < arr.size() - i; j++){
            if(arr[j - 1] > arr[j])){
                std::swap(arr[j-1], arr[j]);
            }
        }
    }
    return arr;
}
```

## 複雜度分析

平均情況與最壞情況均為 $O(n^2)$, 使用了 temp 作為臨時交換變量，空間複雜度為 $O(1)$. 可以做適當程度的優化，當某一次外迴圈中發現陣列已經有序，就跳出迴圈不再執行，但這僅對於部分的輸入有效，平均及最壞時間複雜度仍為$O(n^2)$

```
void bubbleSort(vector<int> & arr){
    bool unsorted = true;
    for(int i = 0; i < arr.size() && unsorted; i++){
        unsorted = false;
        for(int j = 1; j < arr.size() - i; j++){
            if(arr[j - 1] > arr[j])){
                std::swap(arr[j-1], arr[j]);
                unsorted = true;
            }
        }
```

```
    }
    return arr;
}
```

## Reference

- 氣泡排序 - 維基百科，自由的百科全書

# Selection Sort - 選擇排序

核心：不斷地選擇剩餘元素中的最小者。

1. 找到陣列中最小元素並將其和陣列第一個元素交換位置。
2. 在剩下的元素中找到最小元素並將其與陣列第二個元素交換，直至整個陣列排序。

性質：

- 比較次數=(N-1)+(N-2)+(N-3)+...+2+1~N^2/2
- 交換次數=N
- 運行時間與輸入無關
- 數據移動最少

下圖來源為 File:Selection-Sort-Animation.gif - IB Computer Science

## Implementation

### Python

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


def selectionSort(alist):
    for i in xrange(len(alist)):
        print(alist)
        min_index = i
```

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

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

## Java

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

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

## C++

```
void selectionSort(vector<int> & arr){
    int min_idx = 0;
    for(int i = 0; i < arr.size(); i++){
        min_idx = i;
```

```
        for(int j = i + 1; j < arr.size(); j++){
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        std::swap(arr[i], arr[min_idx]);
    }
}
```

## Reference

- 選擇排序 - 維基百科，自由的百科全書
- The Selection Sort — Problem Solving with Algorithms and Data Structures

# Insertion Sort - 插入排序

核心：通過構建有序序列，對於未排序序列，從後向前掃描(對於單向鏈表則只能從前往後遍歷)，找到相應位置並插入。實現上通常使用in-place排序(需用到O(1)的額外空間)

1. 從第一個元素開始，該元素可認為已排序
2. 取下一個元素，對已排序陣列從後往前掃描
3. 若從排序陣列中取出的元素大於新元素，則移至下一位置
4. 重複步驟3，直至找到已排序元素小於或等於新元素的位置
5. 插入新元素至該位置
6. 重複2~5

性質：

- 交換操作和陣列中導致的數量相同
- 比較次數>=倒置數量，<=倒置的數量加上陣列的大小減一
- 每次交換都改變了兩個順序顛倒的元素的位置，即減少了一對倒置，倒置數量為0時即完成排序。
- 每次交換對應著一次比較，且1到N-1之間的每個i都可能需要一次額外的記錄(a[i]未到達陣列左端時)
- 最壞情況下需要~$N^2/2$次比較和 $N^2/2$次交換，最好情況下需要$N-1$次比較和0次交換。
- 平均情況下需要~$N^2/4$次比較和~$N^2/4$次交換

## Implementation

### Python

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

```python
def insertionSort(alist):
    for i, item_i in enumerate(alist):
        print alist
        index = i
        while index > 0 and alist[index - 1] > item_i:
            alist[index] = alist[index - 1]
            index -= 1

        alist[index] = item_i

    return alist

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

## Java

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

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

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

實現(C++)：

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

## 希爾排序 Shell sort

核心：基於插入排序，使陣列中任意間隔為h的元素都是有序的，即將全部元素分為h個區域使用插入排序。其實現可類似於插入排序但使用不同增量。更高效的原因是它權衡了子陣列的規模和有序性。

實現(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;
        }
}
```

希爾排序只描述了分為多個h做插入排序，並沒有規定h的值，事實上有很多研究就是在探討不同的h值對於複雜度的影響，在英文版的wiki百科的希爾排序條目中，給出了多種不同的h序列及分析，事實上可以看到Sedgewick給出的序列已經可以達到最差$\Theta(N^{4/3})$的複雜度。在實際應用上，若不是排序非常大的序列，這個複雜度已經可以接受，另外希爾排序的實現簡單，尤其是在硬體上，因此可以用應用在嵌入式系統之中。

## Reference


- 插入排序 - 維基百科，自由的百科全書
- 希爾排序 - 維基百科，自由的百科全書
- The Insertion Sort — Problem Solving with Algorithms and Data Structures

# Merge Sort - 合併排序

核心：將兩個有序對數組合併成一個更大的有序數組。通常做法為遞歸排序，並將兩個不同的有序數組合併到第三個數組中。

先來看看動圖，合併排序是一種典型的分治(divide and conquer)應用。

6 5 3 1 8 7 2 4

## Python

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


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

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

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

```
        sortedArray += B[r:]

        return sortedArray

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
merge_sort = Sort()
print(merge_sort.mergeSort(unsortedArray))
```

# 原地(in-place)合併

## Java

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

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

```
            }
        }
    }

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

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

## C++

```
void merge (vector<int>& arr, int low, int mid, int high){
    vector<int> helper(arr.size());
    for(int k = low; k <= high; k++){
        helper[k] = arr[k];
    }
    int i = low, j = mid+1;
    for(int k = low; k <= high; k++){
        if(i > mid){
            arr[k] = helper[j];
            j++;
        }
        else if(j > high){
            arr[k] = helper[i];
            i++;
        }
        else if(helper[j] > helper[i]){
            arr[k] = helper[j];
            j++;
        }
        else{
            arr[k] = helper[i];
            i++;
        }
    }
}
```

```
void mergeSort(vector<int>& arr, int low, int high){
    int mid = low + (high - low)/2;
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);
    merge(arr, low, mid, high);
}
```

時間複雜度為 $O(N \log N)$, 使用了等長的輔助陣列，空間複雜度為 $O(N)$。

## Reference

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

# Heap Sort - 堆排序

堆排序通常基於**二元堆** 實現，以大根堆(根結點為最大值)爲例，堆排序的實現過程分爲兩個子過程。第一步爲取出大根堆的根節點(當前堆的最大值), 由於取走了一個節點，故需要對餘下的元素重新建堆。重新建堆後繼續取根節點，循環直至取完所有節點，此時數組已經有序。基本思想就是這樣，不過實現上還是有些小技巧的。

## 堆的操作

以大根堆爲例，堆的常用操作如下。

1. 最大堆調整（Max_Heapify）：將堆的末端子節點作調整，使得子節點永遠小於父節點
2. 創建最大堆（Build_Max_Heap）：將堆所有數據重新排序
3. 堆排序（HeapSort）：移除位在第一個數據的根節點，並做最大堆調整的遞歸運算

其中步驟1是給步驟2和3用的。

6 5 3 1 8 7 2 4

建堆時可以自頂向下，也可以採取自底向上，以下先採用自底向上的思路分析。我們可以將數組的後半部分節點想象爲堆的最下面的那些節點，由於是單個節點，故顯然滿足二叉堆的定義，於是乎我們就可以從中間節點向上逐步構建二叉堆，每前進一步都保證其後的節點都是二叉堆，這樣一來前進到第一個節點時整個數組就是一個二叉堆了。下面用 C++/Java 實現一個堆的類。C++/Java 中推薦使用 PriorityQueue 來使用堆。

堆排在空間比較小(嵌入式設備和手機)時特別有用，但是因爲現代系統往往有較多的快取，堆排序無法有效利用快取，數組元素很少和相鄰的其他元素比較，故快取未命中的機率遠大於其他在相鄰元素間比較的算法。但是在大數據的排序下又重新發揮了重要作用，因爲它在插入操作和刪除最大元素的混合動態場景中能保證對數級別的運行時間。

### C++

```
#include <iostream>
```

```
#include <vector>

using namespace std;

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

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

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

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

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

    // adjust the heap to max-heap
    void max_heapify(vector<int> &nums, int k, int len) {
        // int len = nums.size();
        while (k < len) {
            int max_index = k;
            // left leaf node search
            int l = left(k);
            if (l < len && nums[l] > nums[max_index]) {
                max_index = l;
            }
            // right leaf node search
            int r = right(k);
            if (r < len && nums[r] > nums[max_index]) {
                max_index = r;
            }
```

```
            // node after k are max-heap already
            if (k == max_index) {
                break;
            }
            // keep the root node the largest
            int temp = nums[k];
            nums[k] = nums[max_index];
            nums[max_index] = temp;
            // adjust not only just current index
            k = max_index;
        }
    }

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

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

```
            nums[0] = nums[i];
            nums[i] = temp;
            // reconstruct heap
            build_max_heapify(nums, i);
        }
        print_heap(nums, len);
    }

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

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

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

    return 0;
}
```

## Java

```
import java.util.*;

public class HeapSort {
    // sign = 1 ==> min-heap, sign = -1 ==> max-heap
    private void siftDown(int[] nums, int k, int size, int sign) {
        int half = (size >>> 1);
        while (k < half) {
            int index = k;
            // left leaf node search
            int l = (k << 1) + 1;
            if (l < size && (sign * nums[l]) < (sign * nums[index])) {
                index = l;
            }
            // right leaf node search
            int r = l + 1;
```

```
            if (r < size && (sign * nums[r]) < (sign * nums[index])) {
                index = r;
            }
            // already heapify
            if (k == index) break;
            // keep the root node the smallest/largest
            int temp = nums[k];
            nums[k] = nums[index];
            nums[index] = temp;
            // adjust next index
            k = index;
        }
    }

    private void heapify(int[] nums, int size, int sign) {
        for (int i = size / 2; i >= 0; i--) {
            siftDown(nums, i, size, sign);
        }
    }

    private void minHeap(int[] nums, int size) {
        heapify(nums, size, 1);
    }

    private void maxHeap(int[] nums, int size) {
        heapify(nums, size, -1);
    }

    public void sort(int[] nums, boolean ascending) {
        if (ascending) {
            // build max heap
            maxHeap(nums, nums.length);
            // heap sort
            for (int i = nums.length - 1; i >= 0; i--) {
                int temp = nums[0];
                nums[0] = nums[i];
                nums[i] = temp;
                // reconstruct max heap
                maxHeap(nums, i);
            }
        } else {
            // build min heap
            minHeap(nums, nums.length);
            // heap sort
            for (int i = nums.length - 1; i >= 0; i--) {
                int temp = nums[0];
                nums[0] = nums[i];
                nums[i] = temp;
                // reconstruct min heap
                minHeap(nums, i);
            }
```

```
        }
    }

    public static void main(String[] args) {
        int[] A = new int[]{19, 1, 10, 14, 16, 4, 4, 7, 9, 3, 2, 8, 5, 11};
        HeapSort heapsort = new HeapSort();
        heapsort.sort(A, true);
        for (int i : A) {
            System.out.println(i);
        }
    }
}
```

## 複雜度分析

從程式碼中可以發現堆排最費時間的地方在於構建二叉堆的過程。

上述構建大根堆和小根堆都是自底向上的方法，建堆過程時間複雜度爲 $O(2N)$, 堆化過程(可結合圖形分析，最多需要調整的層數爲最大深度)時間複雜度爲 $\log i$, 故堆排過程中總的時間複雜度爲 $O(N \log N)$.

先看看建堆的過程，畫圖分析(比如以8個節點爲例)可知在最壞情況下，每次都需要調整之前已經成爲堆的節點，那麼就意味着有二分之一的節點向下比較了一次，四分之一的節點向下比較了兩次，八分之一的節點比較了三次... 等差等比數列求和，具體過程可參考下面的連結。

## Reference


- 堆排序 - 維基百科，自由的百科全書
- Priority Queues - Robert Sedgewick 的大作，詳解了關於堆的操作。
- 經典排序算法總結與實現 | Jark's Blog - 堆排序講的很好。
- *Algorithm* - Robert Sedgewick
- 堆排序中建堆過程時間複雜度O(n)怎麼來的？
- 《大話數據結構》第9章 排序 9.7 堆排序（上） - 伍迷 - 博客園
- 《大話數據結構》第9章 排序 9.7 堆排序（下） - 伍迷 - 博客園

# Bucket Sort

桶排序和合併排序有那麼點點類似，也使用了合併的思想。大致步驟如下：

1. 設置一個定量的數組當作空桶。
2. Divide - 從待排序數組中取出元素，將元素按照一定的規則塞進對應的桶子去。
3. 對每個非空桶進行排序，通常可在塞元素入桶時進行插入排序。
4. Conquer - 從非空桶把元素再放回原來的數組中。

## Reference

- Bucket Sort Visualization - 動態示例。
- 桶排序 - 維基百科，自由的百科全書

# Counting Sort

計數排序，顧名思義，就是對待排序陣列按元素進行計數。使用前提是需要先知道待排序陣列的元素範圍，將這些一定範圍的元素置於新陣列中，新陣列的大小爲待排序陣列中最大元素與最小元素的差值。

維基上總結的四個步驟如下：

1. 定新陣列大小——找出待排序的陣列中最大和最小的元素
2. 統計次數——統計陣列中每個值爲i的元素出現的次數，存入新陣列C的第i項
3. 對統計次數逐個累加——對所有的計數累加（從C中的第一個元素開始，每一項和前一項相加）
4. 反向填充目標陣列——將每個元素i放在新陣列的第C(i)項，每放一個元素就將C(i)減去1

其中反向填充主要是爲了避免重復元素落入新陣列的同一索引處。

## Reference

- 計數排序 - 維基百科，自由的百科全書 - 中文版的維基感覺比英文版的好理解些。
- Counting Sort Visualization - 動畫真心不錯~ 結合著看一遍就理解了。

# Radix Sort

經典排序演算法 - 基數排序Radix sort

原理類似桶排序 Bucket Sort,這裡總是需要10個桶,多次使用

首先以個位數的值進行裝桶,即個位數爲1則放入1號桶,爲9則放入9號桶,暫時忽視十位數

例如

取一個簡單的待排序陣列[62,14,59,88,16]

分配10個桶,桶編號爲0-9,以個位數數字爲桶編號依次放入桶,變成下邊這樣

| 0 | 0 | 62 | 0 | 14 | 0 | 16 | 0 | 88 | 59 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |桶編號

將桶裏的數字順序取出來,

輸出結果:[62,14,16,88,59]

再次入桶,不過這次以十位數的數字爲準,進入相應的桶,變成下邊這樣:

由於前邊做了個位數的排序,所以當十位數相等時,個位數字是由小到大的順序入桶的,就是說,入完桶還是有序

| 0 | 14,16 | 0 | 0 | 0 | 59 | 62 | 0 | 88 | 0 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |桶編號

因爲沒有大過100的數字,沒有百位數,所以到這排序完畢,順序取出即可

最後輸出結果:[14,16,59,62,88]

文章引用自 http://www.cnblogs.com/kkun/archive/2011/11/23/2260275.html

# Basics Algorithm

本章主要介紹一些常用的基本演算法，後序章節介紹一些高級演算法。

# Divide and Conquer - 分治法

在計算機科學中，分治法是一種很重要的演算法。分治法即『分而治之』，把一個複雜的問題分成兩個或更多的相同或相似的子問題，再把子問題分成更小的子問題……直到最後子問題可以簡單的直接求解，原問題的解即子問題的解的合併。這個思想是很多高效演算法的基礎，如排序演算法（快速排序，合併排序）等。

## 分治法思想

分治法所能解決的問題一般具有以下幾個特徵：

1. 問題的規模縮小到一定的程度就可以容易地解決。
2. 問題可以分解爲若干個規模較小的相同問題，即該問題具有**最優子結構**性質。
3. 利用該問題分解出的子問題的解可以合併爲該問題的解。
4. 該問題所分解出的各個子問題是相互獨立的，即子問題之間不包含公共的子問題。

分治法的三個步驟是：

1. 分解（Divide）：將原問題分解爲若乾子問題，這些子問題都是原問題規模較小的實例。
2. 解決（Conquer）：遞歸地求解各子問題。如果子問題規模足夠小，則直接求解。
3. 合併（Combine）：將所有子問題的解合併爲原問題的解。

分治法的經典題目：

1. 二分搜索
2. 大整數乘法
3. Strassen矩陣乘法
4. 棋盤覆蓋
5. 歸並排序
6. 快速排序
7. 循環賽日程表
8. 河內塔

# Binary Search - 二分搜索

二分搜索是一種在有序陣列中尋找目標值的經典方法，也就是說使用前提是『有序陣列』。非常簡單的題中『有序』特徵非常明顯，但更多時候可能需要我們自己去構造『有序陣列』。下面我們從最基本的二分搜索開始逐步深入。

## 模板一 - lower/upper bound

定義 lower bound 爲在給定升序陣列中大於等於目標值的最小索引，upper bound 則爲小於等於目標值的最大索引，下面給出程式碼和測試用例。

### Java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        int[] nums = new int[]{1,2,2,3,4,6,6,6,13,18};
        System.out.println(lowerBound(nums, 6)); // 5
        System.out.println(upperBound(nums, 6)); // 7
        System.out.println(lowerBound(nums, 7)); // 8
        System.out.println(upperBound(nums, 7)); // 7
    }

    /*
    * nums[index] >= target, min(index)
    */
    public static int lowerBound(int[] nums, int target) {
        if (nums == null || nums.length == 0) return -1;
        int lb = -1, ub = nums.length;
        while (lb + 1 < ub) {
            int mid = lb + (ub - lb) / 2;
            if (nums[mid] < target) {
                lb = mid;
            } else {
                ub = mid;
            }
        }

        return lb + 1;
    }

    /*
    * nums[index] <= target, max(index)
    */
    public static int upperBound(int[] nums, int target) {
```

```
        if (nums == null || nums.length == 0) return -1;
        int lb = -1, ub = nums.length;
        while (lb + 1 < ub) {
            int mid = lb + (ub - lb) / 2;
            if (nums[mid] > target) {
                ub = mid;
            } else {
                lb = mid;
            }
        }

        return ub - 1;
    }
}
```

## 源碼分析

以 `lowerBound` 的實現爲例，以上二分搜索的模板有幾個非常優雅的實現：

1. `while` 循環中 `lb + 1 < ub` , 而不是等號，因爲取等號可能會引起死循環。初始化 `lb < ub` 時，最後循環退出時一定有 `lb + 1 == ub` .
2. `mid = lb + (ub - lb) / 2` , 可有效防止兩數相加後溢出。
3. `lb` 和 `ub` 的初始化，初始化爲陣列的兩端以外，這種初始化方式比起 `0` 和 `nums.length - 1` 有不少優點，詳述如下。

如果遇到有問插入索引的位置時，可以分三種典型情況：

1. 目標值在陣列範圍之內，最後返回值一定是 `lb + 1`
2. 目標值比陣列最小值還小，此時 `lb` 一直爲 `-1` , 故最後返回 `lb + 1` 也沒錯，也可以將 `-1` 理解爲陣列前一個更小的值
3. 目標值大於等於陣列最後一個值，由於循環退出條件爲 `lb + 1 == ub` , 那麼循環退出時一定有 `lb = A.length - 1` , 應該返回 `lb + 1`

綜上所述，返回 `lb + 1` 是非常優雅的實現。其實以上三種情況都可以統一爲一種方式來理解，即索引 `-1` 對應於陣列前方一個非常小的數，索引 `ub` 即對應陣列後方一個非常大的數，那麼要插入的數就一定在 `lb` 和 `ub` 之間了。

**有時複雜的邊界條件處理可以通過『補項』這種優雅的方式巧妙處理。**

關於lb 和 ub 的初始化，由於 `mid = lb + (ub - lb) / 2` , 且有 `lb + 1 < ub` ，故 mid 還是有可能爲 `ub - 1` 或者 `lb + 1` 的，在需要訪問 `mid + 1` 或者 `mid - 1` 處索引的元素時可能會越界。這時候就需要將初始化方式改爲 `lb = 0, ub = A.length - 1` 了，最後再加一個關於 `lb, ub` 處索引元素的判斷即可。如 Search for a Range 和 Find Peak Element. 尤其是 Find Peak Element 中 lb 和 ub 的初始值如果初始化爲-1和陣列長度會帶來一些麻煩。

# 模板二 - 最優解

除了在有序陣列中尋找目標值這種非常直接的二分搜索外，我們還可以利用二分搜索求最優解（最大值/最小值），通常這種題中只是隱含了『有序陣列』，需要我們自己構造。

用數學語言來描述就是『求滿足某條件 $C(x)$ 的最小/大的 $x$』，以求最小值爲例，對於任意滿足條件的 $x$, 如果所有的 $x \le x' \le UB$ 對於 $C(x')$ 都爲真（其中 `UB` 可能爲無窮大，也可能爲滿足條件的最大的解，如果不滿足此條件就不能保證二分搜索的正確性），那麼我們就能使用二分搜索進行求解，其中初始化時下界 `lb` 初始化爲不滿足條件的值 `LB` , 上界初始化爲滿足條件的上界 `UB` . 隨後在 `while` 循環內部每次取中，滿足條件就取 `ub = mid` , 否則 `lb = mid` , 那麼最後 `ub` 就是要求的最小值。求最大值時類似，只不過處理的是 `lb` .

以 POJ No.1064 爲例。

## Problem Statement

有 $N$ 條繩子，它們的長度分別爲 $L_i$. 如果從它們中切割出 $K$ 條長度相同的繩子的話，這 $K$ 條繩子每條最長能有多長？答案保留到小數點後兩位。

### 輸入

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

### 輸出

2.00

## 題解

這道題看似是一個最優化問題，我們來嘗試下使用模板二的思想求解，**令 $C(x)$ 爲『可以得到 $K$ 條長度爲 $x$ 的繩子』**。根據題意，我們可以將上述條件進一步細化爲：

$$C(x) = \sum_i (floor(L_i/x)) \ge K$$

我們現在來分析下可行解的上下界。由於答案保留小數點後兩位，顯然繩子長度一定大於0，大於0的小數點後保留兩位的最小值爲 `0.01` , 顯然如果問題最後有解， `0.01` 一定是可行解中最小的，且這個解可以分割出的繩子條數是最多的。一般在 OJ 上不同變量都是會給出範圍限制，那麼我們將上界初始化爲 `最大範圍 + 0.01` , 它一定在可行解之外（也可以遍歷一遍陣列取陣列最大值，但其實二分後複雜度相差不大）。使用二分搜索後最後返回 `lb` 即可。

## Java

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
```

```
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int k = in.nextInt();
        double[] nums = new double[n];
        for (int i = 0; i < n; i++) {
            nums[i] = in.nextDouble();
        }
        System.out.printf("%.2f\n", Math.floor(solve(nums, k) * 100) / 100);
    }

    public static double solve(double[] nums, int K) {
        double lb = 0.00, ub = 10e5 + 0.01;
        // while (lb + 0.001 < ub) {
    for (int i = 0; i < 100; i++) {
            double mid = lb + (ub - lb) / 2;
            if (C(nums, mid, K)) {
                lb = mid;
            } else {
                ub = mid;
            }
        }
        return lb;
    }

    public static boolean C(double[] nums, double seg, int k) {
        int count = 0;
        for (double num : nums) {
            count += Math.floor(num / seg);
        }
        return count >= k;
    }
}
```

## 源碼分析

方法 `C` 只做一件事，給定陣列 `nums`，判斷是否能切割出 `K` 條長度均爲 `seg` 的繩子。 `while` 循環中使用 `lb + 0.001 < ub`，不能使用 `0.01`，因爲計算 `mid` 時有均值的計算，對於 `double` 型數值否則會有較大誤差。

## 模板三 - 二分搜索的 `while` 結束條件判定

對於整型我們通常使用 `lb + 1 < ub`，但對於 `double` 型數據來說會有些精度上的丟失，使得結束條件不是那麼好確定。像上題中採用的方法是題目中使用的精度除10。但有時候這種精度可能還是不夠，如果結束條件 `lb + EPS < ub` 中使用的 EPS 過小時 double 型數據精度有可能不夠從而導致死循環的產生！這時候我們將 `while` 循環體替換爲 `for (int i = 0; i < 100; i++)`，100 次循環後可以達到 $10^{-30}$ 精度範圍，一般都沒問題。

# 模板四 – （九章算法）模版

這個模版跟第一個模版類似， 但是相對更容易上手。這個模版的核心是， `將binary search 問題轉化成：尋找第一個或者最後一個，該target元素出現的位置的問題` ， `Find the any/first/last position of target in nums` . 詳解請見下面的例題。這個模版有四個要素。

1. start + 1 < end 表示， 當指針指到兩個元素，相鄰或者相交的時候， 循環停止。 這樣的話在最終分情況討論的時候，只用考慮 `1～2` 個元素。
2. start + (end - start) / 2 寫C++ 和 Java的同學要考慮到int overflow的問題， 所以需要考慮邊界情況。 寫Python的同學就不用考慮了， 因爲python這個語言本身已經非常努力的保證了number不會overflow。
3. A[mid] ==, >, < 在循環中， 分三種情況討論邊界。 要注意， 在移動 `start` 和 `end` 的時候， 只要單純的把指針指向 `mid` 的位置， 不要 `+1` 或者 `-1` 。 因爲只移動邊界到 `mid` 的位置， 不會誤刪除target。在工程中，儘量在程序最後的時候統一寫 `return` , 這樣可以增強可讀性。
4. A[start], A[end]? target 在循環結束時，因爲只有1～2個元素需要討論，所以結果非常容易解釋清楚。 只存在的2種情況爲， 1. `start + 1 == end` 邊界指向相鄰的兩個元素， 這時只需要分情況討論 `start` 和 `end` 與target的關係，就可以得出結果。 2. `start == end` 邊界指向同一元素，其實這個情況還是可以按照1的方法，分成 `start``end` 討論，只不過討論結果一樣而已。

## Python

```
class Solution:
    def binary_search(self, array, target):
        if not array:
            return -1

        start, end = 0, len(array) - 1
        while start + 1 < end:
            mid = (start + end) / 2
            if array[mid] == target:
                start = mid
            elif array[mid] < target:
                start = mid
            else:
                end = mid

        if array[start] == target:
            return start
        if array[end] == target:
            return end
        return -1
```

## Java

```
class Solution {
    public int binarySearch(int[] array, int target) {
```

```
        if (array == null || array.length == 0) {
            return -1;
        }

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

## Problem Statement

Search for a Range

### 樣例

給出[5, 7, 7, 8, 8, 10]和目標值target=8,

返回[3, 4]

## Python

```
class Solution:
    def search_range(self, array, target):
        ret = [-1, -1]
        if not array:
            return ret
        # search first position of target
        st, ed = 0, len(array) - 1
        while st + 1 < ed:
            mid = (st + ed) / 2
            if array[mid] == target:
                ed = mid
            elif array[mid] < target:
```

```
                st = mid
            else:
                ed = mid
        if array[st] == target:
            ret[0] = st
        elif array[ed] == target:
            ret[0] = ed

        # search last position of target
        st, ed = 0, len(array) - 1
        while st + 1 < ed:
            mid = (st + ed) / 2
            if array[mid] == target:
                st = mid
            elif array[mid] < target:
                st = mid
            else:
                ed = mid
        if array[ed] == target:
            ret[1] = ed
        elif array[st] == target:
            ret[1] = st

        return ret
```

## 源碼分析

search range的問題可以理解爲， 尋找第一次target出現的位置和最後一次target出現的位置。 當尋找第一次target出現位置的循環中， `array[mid] == target` 表示， target可以出現在mid或者mid更前的位置， 所以將ed移動到mid。當循環跳出時， st的位置在ed之前，所以先判斷在st位置上是否是target， 再判斷ed位置。當尋找最後一次target出現位置的循環中， `array[mid] == target` 表示， target可以出現在mid或者mid之後的位置， 所以將st移動到mid。 當循環結束時，ed的位置比st的位置更靠後， 所以先判斷ed的位置是否爲target， 再判斷st位置。 最後返回ret。

## Reference


- 《挑戰程序設計競賽》

# Math

本小節總結一些與數學（尤其是數論部分）有關的基礎，主要總結了《挑戰程序設計競賽》(原文為《プログラミングコンテストチャレンジブック》)第二章。主要包含以下內容：

1. Greatest Common Divisor(最大公因數)
2. Prime(質數基礎理論)
3. Modulus(取模運算)
4. Fast Power(快速冪運算)

## Modulus - 取模運算

有時計算結果可能會溢位，此時往往需要對結果取餘。如果有 `a % m = c % m` 和 `b % m = d % m`, 那麼有以下模運算成立。

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

需要注意的是沒有除法運算，另外由於最終結果可能溢位，故需要使用更大範圍的類型來保存取模之前的結果。另外若 `a` 是負數時往往需要改寫爲 `a % m + m`, 這樣就保證結果在 `[0, m - 1]` 範圍內了。

## Fast Power - 快速冪運算

快速冪運算的核心思想爲反覆平方法，將冪指數表示爲2的冪次的和，等價於二進制進行移位計算（不斷取冪的最低位），比如 $x^{22} = x^{16}x^4x^2$.

### C++

```
long long fastModPow(lont long x, int n, long long mod) {
    long long res = 1 % mod;
    while(n > 0) {
        //if lowest bit is 1, fast judge of even number
        if((n & 1) != 0)
            res = res * x % mod;
        x = x * x % mod;
        n >>= 1;
    }
    return res;
}
```

### Java

```java
import java.util.*;

public class FastPow {
    public static long fastModPow(long x, long n, long mod) {
        long res = 1 % mod;
        while (n > 0) {
            // if lowest bit is 1
            if ((n & 1) != 0) res = res * x % mod;
            x = x * x % mod;
            n >>= 1;
        }
        return res;
    }

    public static void main(String[] args) {
        if (args.length != 2 && args.length != 3) return;

        long x = Long.parseLong(args[0]);
        long n = Long.parseLong(args[1]);
        long mod = Long.MAX_VALUE;
        if (args.length == 3) {
            mod = Long.parseLong(args[2]);
        }
        System.out.println(fastModPow(x, n, mod));
    }
}
```

# Math

本小節總結一些與數學（尤其是數論部分）有關的基礎，主要總結了《挑戰程序設計競賽》第二章。

## 最大公因數(GCD, Greatest Common Divisor)

常用的方法爲輾轉相除法，也稱爲歐幾里得算法。不妨設函數 `gcd(a, b)` 是自然是 `a` , `b` 的最大公因數，不妨設 `a > b` , 則有 $a = b \times p + q$, 那麼對於 `gcd(b, q)` 則是 `b` 和 `q` 的最大公因數，也就是說 `gcd(b, q)` 既能整除 `b` , 又能整除 `a` (因爲 $a = b \times p + q$, `p` 是整數)，如此反覆最後得到 `gcd(a, b) = gcd(c, 0)` , 第二個數爲0時直接返回 `c` . 如果最開始 `a < b` , 那麼 `gcd(b, a % b) = gcd(b, a) = gcd(a, b % a)` .

關於時間複雜度的證明：可以分 `a > b/2` 和 `a < b/2` 證明，對數級別的時間複雜度，過程略。

與最大公因數相關的還有最小公倍數(LCM, Lowest Common Multiple), 它們兩者之間的關係爲 $lcm(a, b) \times gcd(a, b) = |ab|$.

### Java

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

### Problem

給定平面上兩個座標 P1=(x1, y1), P2=(x2,y2), 問線段 P1P2 上除 P1, P2以外還有幾個整數座標點？

#### Solution

問的是線段 P1P2, 故除 P1,P2以外的座標需在 x1,x2,y1,y2範圍之內，且不包含端點。在兩端點不重合的前提下有：

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$

那麼若得知 $M = gcd(x_2 - x_1, y_2 - y_1)$, 則有 $x - x_1$ 必爲 $x_2 - x_1/M$ 的整數倍大小，又因爲 $x_1 < x < x_2$, 故最多有 $M - 1$個整數座標點。

## 擴展歐幾里得算法

求解整係數 $x$ 和 $y$ 滿足 $d = gcd(a, b) = ax + by$, 仿照歐幾里得算法，應該要尋找

$gcd(b, a\%b) = bx' + (a\%b)y'$.

### Java

```
public class Solution {
    public static int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    public static int[] gcdExt(int a, int b) {
        if (b == 0) {
            return new int[] {a, 1, 0};
        } else {
            int[] vals = gcdExt(b, a % b);
            int d = vals[0];
            int x = vals[2];
            int y = vals[1];
            y -= (a / b) * x;
            return new int[] {d, x, y};
        }
    }

    public static void main(String[] args) {
        int a = 4, b = 11;
        int[] result = gcdExt(a, b);
        System.out.printf("d = %d, x = %d, y = %d.\n", result[0], result[1], result[
2]);
    }
}
```

## Problem

求整數 $x$ 和 $y$ 使得 $ax + by = 1$.

### Solution

不妨設 `gcd(a, b) = M` , 那麼有 $M(a'x + b'y) = 1$ ==> $a'x + b'y = 1/M$ 如果 M 大於1，由於等式左邊爲整數，故等式不成立，所以要想題中等式有解，必有 `gcd(a, b) = 1` .

**擴展提：題中等式右邊爲1，假如爲2又會怎樣？**

提示：此時$c = k \cdot gcd(a, b), x' = k \cdot x$ ==> $c \% gcd(a, b) == 0$, c 爲等式右邊的正整數值。詳細推導見 How to find solutions of linear Diophantine ax + by = c?

# Prime

質數：恰好有兩個因數的整數，一個是1，另一個則是它自己，比如整數3和5就是質數。質數的基本算法有**素性測試、埃氏篩法和整數分解。**

## 質數測試

如果 `d` 是 `n` 的因數，則易知 $n = d \cdot \frac{n}{d}$, 因此 `n/d` 也是 `n` 的因數，且這兩個因數中的較小者 $\min(d, n/d) <= \sqrt{n}$. 因此我們只需要對前 $\sqrt{n}$ 個數進行處理。

## 埃氏篩法 Sieve of Eratosthenes

質數測試針對的是單個整數，如果需要枚舉整數 `n` 以內的質數就需要埃氏篩法了。核心思想是枚舉從小到大的質數並將質數的整數倍依次從原整數數組中刪除，餘下的即爲全部質數。

## 區間篩法

求區間 `[a, b)` 內有多少質數？

埃氏篩法得到的是 `[1, n)` 內的質數，求區間質數時不太容易直接求解，我們採取以退爲進的思路先用埃氏篩法求得 `[1, b)` 內的質數，然後截取爲 `[a, b)` 即可。

## Implementation

### Java

```
import java.util.*;

public class Prime {
    // test if n is prime
    public static boolean isPrime(int n) {
        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) return false;
        }
        return n != 1; // 1 is not prime
    }

    // enumerate all the divisor for n
    public static List<Integer> getDivisor(int n) {
        List<Integer> result = new ArrayList<Integer>();
        for (int i = 1; i * i <= n; i++) {
            if (n % i == 0) {
                result.add(i);
                // i * i <= n ==> i <= n / i
```

```
                if (i != n / i) result.add(n / i);
            }
        }
        Collections.sort(result);
        return result;
    }

    // 12 = 2 * 2 * 3, the number of prime factor, small to big
    public static Map<Integer, Integer> getPrimeFactor(int n) {
        Map<Integer, Integer> result = new HashMap<Integer, Integer>();
        for (int i = 2; i * i <= n; i++) {
            // if i is a factor of n, repeatedly divide it out
            while (n % i == 0) {
                if (result.containsKey(i)) {
                    result.put(i, result.get(i) + 1);
                } else {
                    result.put(i, 1);
                }
                n = n / i;
            }
        }
        // if n is not 1 at last
        if (n != 1) result.put(n, 1);
        return result;
    }

    // sieve all the prime factor less equal than n
    public static List<Integer> sieve(int n) {
        List<Integer> prime = new ArrayList<Integer>();
        // flag if i is prime
        boolean[] isPrime = new boolean[n + 1];
        Arrays.fill(isPrime, true);
        isPrime[0] = false;
        isPrime[1] = false;
        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) {
                prime.add(i);
                for (int j = 2 * i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }
        return prime;
    }

    // sieve between [a, b)
    public static List<Integer> sieveSegment(int a, int b) {
        List<Integer> prime = new ArrayList<Integer>();
        boolean[] isPrime = new boolean[b];
        Arrays.fill(isPrime, true);
        isPrime[0] = false;
```

```
        isPrime[1] = false;
        for (int i = 2; i < b; i++) {
            if (isPrime(i)) {
                for (int j = 2 * i; j < b; j += i) isPrime[j] = false;
                if (i >= a) prime.add(i);
            }
        }
        return prime;
    }

    public static void main(String[] args) {
        if (args.length == 1) {
            int n = Integer.parseInt(args[0]);
            if (isPrime(n)) {
                System.out.println("Integer " + n + " is prime.");
            } else {
                System.out.println("Integer " + n + " is not prime.");
            }
            System.out.println();

            List<Integer> divisor = getDivisor(n);
            System.out.print("Divisor of integer " + n + ":");
            for (int d : divisor) System.out.print(" " + d);
            System.out.println();
            System.out.println();

            Map<Integer, Integer> primeFactor = getPrimeFactor(n);
            System.out.println("Prime factor of integer " + n + ":");
            for (Map.Entry<Integer, Integer> entry : primeFactor.entrySet()) {
                System.out.println("prime: " + entry.getKey() + ", times: " + entry
.getValue());
            }

            System.out.print("Sieve prime of integer " + n + ":");
            List<Integer> sievePrime = sieve(n);
            for (int i : sievePrime) System.out.print(" " + i);
            System.out.println();
        } else if (args.length == 2) {
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            List<Integer> primeSegment = sieveSegment(a, b);
            System.out.println("Prime of integer " + a + " to " + b + ":");
            for (int i : primeSegment) System.out.print(" " + i);
            System.out.println();
        }
    }
}
```

# Probability

# Shuffle and Sampling - 隨機抽樣和洗牌

## 洗牌算法

- Shuffle a given array - GeeksforGeeks

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

### 題解

這裡以 Fisher–Yates shuffle 演算法爲例，僞代碼如下：

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

轉化爲代碼爲：

```
    /*
     * shuffle cards
     */
    public static void shuffleCard(int[] cards) {
        if (cards == null || cards.length == 0) return;

        Random rand = new Random();
        for (int i = 0; i < cards.length; i++) {
            int k = rand.nextInt(i + 1); // 0~i (inclusive)
            int temp = cards[i];
            cards[i] = cards[k];
            cards[k] = temp;
        }
    }
```

看了算法和代碼後讓我們來使用歸納法簡單證明下這個洗牌算法的正確性。我們要證明的問題是：**數組中每個元素在每個索引處出現的機率均相等。**

對於單個元素來講，以上算法顯然正確，因爲交換後仍然只有一個元素。接下來我們不妨假設其遍歷到數組索引爲 `i-1` 時滿足隨機排列特性，那麼當遍歷到數組索引爲 `i` 時，隨機數 `k` 爲 `i` 的機率爲 `1/i` ,爲 `0~i-1` 的機率爲 `(i-1)/i` . 接下來與索引爲 `i` 的值交換，可以得知 `card[i]` 出現在索引 `i` 的位置的機率爲 `1/i` ,在其他索引位置的機率也爲 `1/i` ;而對於 `card[i]` 之前的元素，以索引 `j` 處的元素 `card[j]` 爲例進行分析可知其在位置 `j` 的機率爲 `1/(i-1) * (i-1)/i = 1/i` ,具體含義爲遍歷到索引 `i-1` 時 `card[j]` 位於索引 `j` 的機率( `1/(i-1)` )乘以遍歷到索引 `i` 時隨機數未選擇與索引 `j` 的數進行交換的機率( `(i-1)/i` ).

需要注意的是前面的 `j <= i-1` , 那麼 `card[j]` 位於索引 `i` 的機率又是多少呢？要位於索引 `i` ，則隨機數 `k` 須爲 `i` , 這種機率爲 `1/i` .

綜上，以上算法可以實現完美洗牌（等機率）。

## Random sampling - 隨機抽樣

隨機抽樣也稱爲水池抽樣，Randomly choosing a sample of k items from a list S containing n items. 大意是從大小爲 n 的數組中隨機選出 m 個整數，要求每個元素被選中的機率相同。

### 題解

比較簡潔的有算法 Algorithm R, 僞代碼如下：

```
/*
  S has items to sample, R will contain the result
*/
ReservoirSample(S[1..n], R[1..k])
  // fill the reservoir array
  for i = 1 to k
      R[i] := S[i]

  // replace elements with gradually decreasing probability
  for i = k+1 to n
    j := random(1, i)   // important: inclusive range
    if j <= k
        R[j] := S[i]
```

轉化爲代碼爲：

```
    /*
     * random sample
     */
    public static int[] randomSample(int[] nums, int m) {
        if (nums == null || nums.length == 0 || m <= 0) return new int[]{};

        int[] sample = new int[m];
        for (int i = 0; i < m; i++) {
            sample[i] = nums[i];
        }

        Random random = new Random();
        for (int i = m; i < nums.length; i++) {
            int k = random.nextInt(i + 1); // 0~i(inclusive)
            if (k < m) {
                sample[k] = nums[i];
            }
        }
```

```
        return sample;
    }
```

和洗牌算法類似，我們要證明的問題是：**數組中每個元素在最終採樣的數組中出現的機率均相等且爲 `m/n`**. 洗牌算法中是排列，而對於隨機抽樣則爲組合。

維基百科上的證明相對容易懂一些，這裏我稍微複述下。首先將數組前 m 個元素填充進新數組 `sample`, 然後從 `m` 開始遍歷直至數組最後一個索引。隨機數 `k` 的範圍爲 `0~i`, 如果 `k < m` ，新數組的索引爲 k 的元素則和原數組索引爲 `i` 的元素交換；如果 `k >= m`, 則不進行交換。 `i == m` 時，以原數組中第 `j` 個元素爲例，它被 `nums[m]` 替換的機率爲 `1/(m+1)`, 也就是說保留在 `sample` 數組中的機率爲 `m/(m+1)`. 對與第 `m+1` 個元素 `nums[m]` 來說，其位於 `sample` 數組中的機率則爲 `m*1/(m+1)` (可替換 m 個不同的元素).

接下來仍然使用數學歸納法證明，若 `i` 遍歷到 `r` 時，其之前的元素出現的機率爲 `m/(r-1)`, 那麼其之前的元素中任一元素 `nums[j]` 被替換的機率爲 `m/r * 1/m = 1/r`, 不被替換的機率則爲 `(r-1)/r`. 故元素 `nums[j]` 在 `i` 遍歷完 `r` 後仍然保留的機率爲 `m/(r-1) * (r-1)/r = m/r`. 而對於元素 `nums[r]` 來說，其要被替換至 `sample` 數組中的機率則爲 `m/r` (隨機數小於m 的個數爲 m).

綜上，以上算法在遍歷完長度爲 n 的數組後每個數出現在最終 `sample` 數組中的機率都爲 `m/n`.

## Implementation and Test case

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

## Java

```
import java.util.*;
import java.util.Random;

public class Probability {
    public static void main(String[] args) {
        int[] cards = new int[10];
        for (int i = 0; i < 10; i++) {
            cards[i] = i;
        }
        // 100000 times test
        final int times = 100000;
        final int m = 5;
        int[][] count = new int[cards.length][cards.length];
        int[][] count2 = new int[cards.length][m];
        for (int i = 0; i < times; i++) {
            shuffleCard(cards);
            shuffleTest(cards, count);
            int[] sample = randomSample(cards, m);
            shuffleTest(sample, count2);
        }
        System.out.println("Shuffle cards");
```

```
        shufflePrint(count);
        System.out.println();
        System.out.println("Random sample");
        shufflePrint(count2);
    }

    /*
     * shuffle cards
     */
    public static void shuffleCard(int[] cards) {
        if (cards == null || cards.length == 0) return;

        Random rand = new Random();
        for (int i = 0; i < cards.length; i++) {
            int k = rand.nextInt(i + 1);
            int temp = cards[i];
            cards[i] = cards[k];
            cards[k] = temp;
        }
    }

    /*
     * random sample
     */
    public static int[] randomSample(int[] nums, int m) {
        if (nums == null || nums.length == 0 || m <= 0) return new int[]{};

        m = Math.min(m, nums.length);
        int[] sample = new int[m];
        for (int i = 0; i < m; i++) {
            sample[i] = nums[i];
        }

        Random random = new Random();
        for (int i = m; i < nums.length; i++) {
            int k = random.nextInt(i + 1);
            if (k < m) {
                sample[k] = nums[i];
            }
        }

        return sample;
    }

    /*
     * nums[i] = j, num j appear in index i ==> count[j][i]
     */
    public static void shuffleTest(int[] nums, int[][] count) {
        if (nums == null || nums.length == 0) return;

        for (int i = 0; i < nums.length; i++) {
```

```
            count[nums[i]][i]++;
        }
    }

    /*
     * print shuffle test
     */
    public static void shufflePrint(int[][] count) {
        if (count == null || count.length == 0) return;

        // print index
        System.out.print("   ");
        for (int i = 0; i < count[0].length; i++) {
            System.out.printf("%-7d", i);
        }
        System.out.println();
        // print num appear in index i in total
        for (int i = 0; i < count.length; i++) {
            System.out.print(i + ": ");
            for (int j = 0; j < count[i].length; j++) {
                System.out.printf("%-7d", count[i][j]);
            }
            System.out.println();
        }
    }
}
```

以十萬次試驗爲例，左側是元素 `i`，列代表在相應索引位置出現的次數。可以看出分佈還是比較隨機的。

```
Shuffle cards
   0      1      2      3      4      5      6      7      8      9
0: 10033  9963   10043  9845   9932   10020  9964   10114  10043  10043
1: 9907   9951   9989   10071  10059  9966   10054  10023  10015  9965
2: 10042  10046  9893   10080  10050  9994   10024  9852   10098  9921
3: 10039  10023  10039  10024  9919   10057  10188  9916   9907   9888
4: 9944   9913   10196  10059  9838   10205  9899   9945   9850   10151
5: 10094  9971   10054  9958   10022  9922   10047  9978   9965   9989
6: 9995   10147  9824   10015  10023  9804   10050  10192  9939   10011
7: 9941   10131  9902   9920   10040  10121  10010  9928   9984   10023
8: 10010  9926   9883   10098  10083  10028  9801   9936   10200  10035
9: 9995   9929   10177  9930   10034  9883   9963   10116  9999   9974

Random sample
   0      1      2      3      4
0: 9966   10026  10078  9966   9891
1: 9958   9806   10066  10022  10039
2: 9923   9936   9964   10051  10083
3: 10165  10088  10184  9928   9916
4: 9998   9990   9973   9931   9832
```

```
5: 10026  9932   9873   10085  10035
6: 9942   9972   9990   10030  10026
7: 9903   10153  9997   10051  10044
8: 10082  10066  9804   9899   10147
9: 10037  10031  10071  10037  9987
```

## Reference

- Fisher–Yates shuffle - 洗牌算法的詳述，比較簡潔的演算法
- Reservoir sampling - 水池抽樣算法
- 如何測試洗牌程序 | 酷 殼 - CoolShell.cn - 借鑑了其中的一些測試方法
- 《計算機程序設計藝術》第二卷（半數值算法） - 3.4.2 隨機抽樣和洗牌
- 《編程珠璣》第十二章 - 抽樣問題

# Bitmap

最開始接觸 bitmap 是在 Jon Bentley 所著《Programming Pearls》 (無繁體中文版，簡體中文版書名為《編程珠璣》) 這本書上，書中所述的方法有點簡單粗暴，不過思想倒是不錯——從 Information Theory 的角度來解釋就是信息壓縮了。即將原來32位表示一個 int 變爲一位表示一個 int. 從空間的角度來說就是巨大的節省了(1/32)。可能的應用有**大數據排序/查找（非負整數）**。

C++ 中有 `bitset` 容器，其他語言可用類似方法實現。

## Implementation

### C

```
#include <stdio.h>
#include <stdlib.h>

/*
 * @param bits: uint array, i: num i of original array
 */
void setbit(unsigned int *bits, unsigned int i, int BIT_LEN)
{
        bits[i / BIT_LEN] |= 1 << (i % BIT_LEN);
}

/*
 * @param bits: uint array, i: num i of original array
 */
int testbit(unsigned int *bits, unsigned int i, int BIT_LEN)
{
        return bits[i / BIT_LEN] & (1 << (i % BIT_LEN));
}

int main(int argc, char *argv[])
{
        const int BIT_LEN = sizeof(int) * 8;
        const unsigned int N = 1 << (BIT_LEN - 1);
        unsigned int *bits = (unsigned int *)calloc(N, sizeof(int));
        for (unsigned int i = 0; i < N; i++) {
                if (i % 10000001 == 0) setbit(bits, i, BIT_LEN);
        }

        for (unsigned int i = 0; i < N; i++) {
                if (testbit(bits, i, BIT_LEN) != 0) printf("i = %u exists.\n", i);
        }
        free(bits);
        bits = NULL;
```

```
        return 0;
}
```

## 源碼分析

核心爲兩個函數方法的使用， `setbit` 用於將非負整數 `i` 置於指定的位。可用分區分位的方式來理解位圖排序的思想，即將非負整數 `i` 放到它應該在的位置。比如16，其可以位於第一個 int 型的第17位，具體實現即將第17位置一，細節見上面代碼。測試某個數是否存在於位圖中也可以採用類似方法。

# Basics Miscellaneous

雜項，涉及「位操作」等。

# Bit Manipulation

位操作有按位與(bitwise and)、或(bitwise or)、非(bitwise not)、左移n位和右移n位等操作。

## XOR - 異或(exclusive or)

> 異或：相同為0，不同為1。也可用「不進位加法」來理解。

異或操作的一些特點：

```
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(shift operation)

移位操作可近似為乘以/除以2的冪。 `0b0010 * 0b0110` 等價於 `0b0110 << 2` . 下面是一些常見的移位組合操作。

1. 將 `x` 最右邊的 `n` 位清零 - `x & (~0 << n)`
2. 獲取 `x` 的第 `n` 位值(0或者1) - `x & (1 << n)`
3. 獲取 `x` 的第 `n` 位的冪值 - `(x >> n) & 1`
4. 僅將第 `n` 位置為 `1` - `x | (1 << n)`
5. 僅將第 `n` 位置為 `0` - `x & (~(1 << n))`
6. 將 `x` 最高位至第 `n` 位(含)清零 - `x & ((1 << n) - 1)`
7. 將第 `n` 位至第0位(含)清零 - `x & (~((1 << (n + 1)) - 1))`
8. 僅更新第 `n` 位，寫入值為 `v` ; `v` 為1則更新為1，否則為0 - `mask = ~(1 << n); x = (x & mask) | (v << i)`

## 實際應用

### 位圖(Bitmap)

位圖一般用於替代flag array，節約空間。
一個int型的陣列用位圖替換後，占用的空間可以縮小到原來的1/32.(若int類型是32位元)
下面代碼定義了一個100萬大小的類圖，setbit和testbit函數

```
#define N 1000000 // 1 million
#define WORD_LENGTH sizeof(int) * 8 //sizeof返回字節數，乘以8，為int類型總位數

//bits為陣列，i控制具體哪位，即i為0~1000000
void setbit(unsigned int* bits, unsigned int i){
```

```
    bits[i / WORD_LENGTH] |= 1<<(i % WORD_LENGTH);
}

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

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

## Reference


- 位運算應用技巧（1） » NoAlGo博客
- 位運算應用技巧（2） » NoAlGo博客
- 位運算簡介及實用技巧（一）：基礎篇 | Matrix67: The Aha Moments
- *cc150* chapter 8.5 and chapter 9.5
- 《編程珠璣2》
- 《Elementary Algorithms》 Larry LIU Xinyu

# Part II - Coding

本節主要總結一些leetcode等題目的實戰經驗。

主要有以下章節構成。

# String - 字串

本章主要介紹字串相關題目。

> 處理字串操作相關問題時，常見的做法是從字串尾部開始編輯，從後往前逆向操作。這麼做的原因是因為字串的尾部往往有足夠空間，可以直接修改而不用擔心覆蓋字串前面的數據。

摘自《程序員面試金典》

# strStr

## Question

- 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.
```

## 題解

對於字串查找問題，可使用雙重for迴圈解決，效率更高的則為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;
    }
}
```

## 源碼分析

1. 邊界檢查： `source` 和 `target` 有可能是空串。
2. 邊界檢查之下標溢出：注意變量 `i` 的循環判斷條件，如果是單純的 `i < source.length()` 則在後面的 `source.charAt(i + j)` 時有可能溢出。
3. 代碼風格：（1）運算符 `==` 兩邊應加空格；（2）變量名不要起 `s1``s2` 這類，要有意義，如 `target``source` ；（3）即使if語句中只有一句話也要加大括號，即 `{return -1;}` ；（4）Java 代碼的大括號一般在同一行右邊，C++ 代碼的大括號一般另起一行；（5） `int i, j;` 聲明前有一行空格，是好的代碼風格。
4. 不要在for的條件中聲明 `i` , `j` ，容易在循環外再使用時造成編譯錯誤，錯誤代碼示例：

## 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

## Question

- CC150: (158) Two Strings Are Anagrams
- leetcode: Valid Anagram | LeetCode OJ

```
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
```

## 題解1 - hashmap 統計字頻

判斷兩個字串是否互為變位詞，若區分大小寫，考慮空白字符時，直接來理解可以認為兩個字串的擁有各不同字符的數量相同。對於比較字符數量的問題常用的方法為遍歷兩個字串，統計其中各字符出現的頻次，若不等則返回 `false` . 有很多簡單字串類面試題都是此題的變形題。

### C++

```cpp
class Solution {
public:
    /**
     * @param s: The first string
     * @param t: 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;
    }
};
```

## 源碼分析

1. 兩個字串長度不等時必不可能為變位詞(需要注意題目條件靈活處理)。
2. 初始化含有256個字符的計數器陣列。
3. 對字串 s 自增，字串 t 遞減，再次遍歷判斷 `letterCount` 陣列的值，小於0時返回 `false` .

在字串長度較長(大於所有可能的字符數)時，還可對第二個 `for` 循環做進一步優化，即 `t.size() > 256` 時，使用256替代 `t.size()` , 使用 `i` 替代 `t[i]` .

## 複雜度分析

兩次遍歷字串，時間複雜度最壞情況下為 $O(2n)$, 使用了額外的陣列，空間複雜度 $O(256)$.

# 題解2 - 排序字串

另一直接的解法是對字串先排序，若排序後的字串內容相同，則其互為變位詞。題解1中使用 hashmap 的方法對於比較兩個字串是否互為變位詞十分有效，但是在比較多個字串時，使用 hashmap 的方法複雜度則較高。

## 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;
        }
    }
};
```

## 源碼分析

對字串 s 和 t 分別排序，而後比較是否含相同內容。對字串排序時可以採用先統計字頻再組裝成排序後的字串，效率更高一點。

## 複雜度分析

C++的 STL 中 sort 的時間複雜度介於 $O(n)$ 和 $O(n^2)$之間，判斷 `s == t` 時間複雜度最壞為 $O(n)$.

## Reference

- *CC150 Chapter 9.1* 中文版 p109

# Compare Strings

## Question

- 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.
```

## 題解

題 Two Strings Are Anagrams | Data Structure and Algorithm 的變形題。題目意思是問B中的所有字元是否都在A中，而不是單個字元。比如B="AABC"包含兩個「A」，而A="ABCD"只包含一個「A」，故返回false. 做題時注意題意，必要時可向面試官確認。

既然不是類似 strstr 那樣的匹配，直接使用二重循環就不太合適了。題目中另外給的條件則是A和B都是全大寫單字，理解題意後容易想到的方案就是先遍歷 A 和 B 統計各字元出現的次數，然後比較次數大小即可。嗯，祭出萬能的哈希表。

### C++

```cpp
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;
    }
};
```

## 源碼解析

1. 異常處理，B 的長度大於 A 時必定返回 `false`，包含了空串的特殊情況。
2. 使用額外的輔助空間，統計各字元的頻次。

## 複雜度分析

遍歷一次 A 字串，遍歷一次 B 字串，時間複雜度最壞 $O(2n)$, 空間複雜度為 $O(26)$.

# Rotate String

## Question

- 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"

...
```

## 題解

常見的翻轉法應用題，仔細觀察規律可知翻轉的分割點在從數組末尾數起的offset位置。先翻轉前半部分，隨後翻轉後半部分，最後整體翻轉。

## Python

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

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

```
        return A
```

## C++

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

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

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

## Java

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

```
        }

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

        return A;
    }

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

## 源碼分析

1. 異常處理，A為空或者其長度為0
2. `offset` 可能超出A的大小，應對 `len` 取餘數後再用
3. 三步翻轉法

Python 雖沒有提供字符串的翻轉，但用 slice 非常容易實現，非常 Pythonic!

## 複雜度分析

翻轉一次時間複雜度近似為 $O(n)$, 原地交換，空間複雜度為 $O(1)$. 總共翻轉3次，總的時間複雜度為 $O(n)$, 空間複雜度為 $O(1)$.

## Reference

- Reverse a string in Python - Stack Overflow

# Valid Palindrome

- tags: [palindrome]

## Question

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

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

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

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

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

## 題解

字符串的回文判斷問題，由於字符串可隨機訪問，故逐個比較首尾字符是否相等最為便利，即常見的『兩根指針』技法。此題忽略大小寫，並只考慮字母和數字字符。鏈表的回文判斷總結見 Check if a singly linked list is palindrome.

## Python

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

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

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

```
                l += 1
                continue
            # find right alphanumeric character
            if not s[r].isalnum():
                r -= 1
                continue
            # case insensitive compare
            if s[l].lower() == s[r].lower():
                l += 1
                r -= 1
            else:
                return False
        #
        return True
```

## C++

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

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

        return true;
    }
```

```
};
```

## Java

```
public class Solution {
    /**
     * @param s A string
     * @return Whether the string is a valid palindrome
     */
    public boolean isPalindrome(String s) {
        if (s == null || s.isEmpty()) return true;

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

        return true;
    }
}
```

## 源碼分析

兩步走：

1. 找到最左邊和最右邊的第一個合法字元(字母或者字元)
2. 一致轉換為小寫進行比較

字元的判斷盡量使用語言提供的 API

## 複雜度分析

兩根指標遍歷一次，時間複雜度 $O(n)$, 空間複雜度 $O(1)$.

# Longest Palindromic Substring

- tags: [palindrome]

## Question

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

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

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

## 題解1 - 窮舉搜索(brute force)

最簡單的方案，窮舉所有可能的子串，判斷子串是否為回文，使用一變數記錄最大回文長度，若新的回文超過之前的最大回文長度則更新標記變數並記錄當前回文的起止索引，最後返回最長回文子串。

### Python

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

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

    def isPalindrome(self, s):
```

```
        if not s:
            return False
        # reverse compare
        return s == s[::-1]
```

## C++

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

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

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

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

## Java

```
public class Solution {
```

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

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

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

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

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

        return true;
    }
}
```

## 源碼分析

使用 `left` , `right` 作為子串的起止索引，用於最後構造返回結果，避免中間構造字串以減少開銷。

## 複雜度分析

窮舉所有的子串，$O(C_n^2) = O(n^2)$, 每次判斷字符串是否為回文，複雜度為 $O(n)$, 故總的時間複雜度為 $O(n^3)$. 故大數據集下可能 TLE. 使用了 `substr` 作為臨時子串，空間複雜度為 $O(n)$.

# 題解2 - 動態規劃(dynamic programming)

要改善效率，可以觀察哪邊有重複而冗餘的計算，例如已知"bab"為回文的情況下，若前後各加一個相同的字元，"cbabc"，當然也是回文。 因此可以使用動態規劃，將先前的結果儲存起來，假設字串 `s` 的長度為 `n` ，我們創建一個$(n \times n)$的bool值矩陣 `P` ， `P[i, j], i<= j` 表示由$[s_i, ..., s_j]$構成的子串是否為回文。就可以得到一個與子結構關係
`P[i, j] = P[i+1, j-1] AND s[i] == s[j]`

而基本狀態為
`P[i, i] = true`
與
`P[i, i+1] = (s[i] == s[i+1])`

因此可以整理成程式碼如下

```
string longestPalindrome(string s) {
    int n = s.length();
    int maxBegin = 0;
    int maxLen = 1;
    bool table[1000][1000] = {false};

    for (int i = 0; i < n; i++) {
        table[i][i] = true;
    }

    for (int i = 0; i < n-1; i++) {
            if (s[i] == s[i+1]) {
            table[i][i+1] = true;
            maxBegin = i;
            maxLen = 2;
        }
    }
    for (int len = 3; len <= n; len++) {
        for (int i = 0; i < n-len+1; i++) {
            int j = i+len-1;
            if (s[i] == s[j] && table[i+1][j-1]) {
                table[i][j] = true;
                maxBegin = i;
                maxLen = len;
            }
        }
    }
    return s.substr(maxBegin, maxLen);
}
```

## 複雜度分析

仍然是兩層迴圈，但每次迴圈內部只有常數次操作，因此時間複雜度是$O(n^2)$，另外空間複雜度是$O(n^2)$

# 題解 3 - Manacher's Algorithm

## Reference

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

# Space Replacement

## Question

- lintcode: (212) Space Replacement

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

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

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

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

Challenge
Do it in-place.
```

## 題解

根據題意，給定的輸入陣列長度足夠長，將空格替換為 `%20` 後也不會overflow。通常的思維為從前向後遍歷，遇到空格即將 `%20` 插入到新陣列中，這種方法在生成新陣列時很直觀，但要求原地替換時就不方便了，這時可聯想到插入排序的做法——從後往前遍歷，空格處標記下就好了。由於不知道新陣列的長度，故首先需要遍歷一次原陣列，字符串類題中常用方法。

需要注意的是這個題並未說明多個空格如何處理，如果多個連續空格也當做一個空格時稍有不同。

### Java

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

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

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

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

```
class Solution {
public:
    /**
     * @param string: An array of Char
     * @param length: The true length of the string
     * @return: The true length of new string
     */
    int replaceBlank(char string[], int length) {
        int space = 0;
        for (int i = 0; i < length; i++) {
            if (string[i] == ' ') space++;
        }

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

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

## 源碼分析

先遍歷一遍求得空格數，得到『新陣列』的實際長度，從後往前遍歷。

## 複雜度分析

遍歷兩次原陣列，時間複雜度近似為 $O(n)$, 使用了 `r` 作為標記，空間複雜度 $O(1)$.

# Count and Say

## Question

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

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

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

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

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

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

Given an integer n, generate the nth sequence.

Example
Given n = 5, return "111221".

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

## 題解

題目大意是找第 n 個數(字符串表示)，規則則是對於連續字符串，表示為重複次數+數本身。純粹是implementation的題目，小心處理實現細節就可以。

### Java

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

        String s = "1";
        for (int i = 1; i < n; i++) {
            int count = 1;
            StringBuilder sb = new StringBuilder();
            int sLen = s.length();
```

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

        return s;
    }
}
```

## C++

```
class Solution {
public:
    string countAndSay(int n) {
        string s = "1";
        if(n <= 1) return s;
        stringstream ss;
        while(n-- > 1){
            int count = 1;
            for(int j = 0; j < s.size(); j++){
                if(j < s.size()-1 and s[j] == s[j+1]){
                    count++;
                }
                else{
                    ss << count << s[j];
                    count = 1;
                }
            }
            s = ss.str();
            ss.str("");
        }
        return s;
    }
};
```

## 源碼分析

字符串是動態生成的，故使用 StringBuilder 更為合適。注意s 初始化為"1", 第一重 for循環中注意循環的次數為 n-1.

## 複雜度分析

略

## Reference

- [leetcode]Count and Say - 喵星人與汪星人

# Integer Array - 整數型陣列

本章主要總結與整數型陣列相關的題目。

# Remove Element

## Question

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

```
Given an array and a value, remove all occurrences of that value in place and retur
n the new length.

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

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]
```

## 題解1 - 使用容器

入門題，返回刪除指定元素後的陣列長度，使用容器操作非常簡單。以 lintcode 上給出的參數為例，遍歷容器內元素，若元素值與給定刪除值相等，刪除當前元素並往後繼續遍歷。 C++的vector已經支援了刪除操作，因此可以直接拿來使用。

### C++

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

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

## 源碼分析

注意在遍歷容器內元素和指定欲刪除值相等時，需要先自減 `--iter` ,因為 `for` 循環會對 `iter` 自增， `A.erase()` 刪除當前元素值並返回指向下一個元素的指針，一增一減正好平衡。如果改用 `while` 循環，則需注意訪問陣列時是否越界。

## 複雜度分析

由於vector每次erase的複雜度是$O(n)$，我們遍歷整個向量，最壞情況下，每個元素都與要刪除的目標元素相等，每次都要刪除元素的複雜度高達$O(n^2)$ 觀察此方法會如此低效的原因，是因為我們一次只刪除一個元素，導致很多沒必要的元素交換移動，如果能夠將要刪除的元素集中處理，則可以大幅增加效率，見題解2。

## 題解2 - 兩根指針

由於題中明確暗示元素的順序可變，且新長度後的元素不用理會。我們可以使用兩根指針分別往前往後遍歷，頭指針用於指示當前遍歷的元素位置，尾指針則用於在當前元素與欲刪除值相等時替換當前元素，兩根指針相遇時返回尾指針索引——即刪除元素後「新陣列」的長度。

### 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;
    }
};
```

## 源碼分析

遍歷當前陣列， `A[i] == elem` 時將陣列「尾部(以 n 為長度時的尾部)」元素賦給當前遍歷的元素。同時自減 `i` 和 `n` ，原因見題解1的分析。需要注意的是 `n` 在遍歷過程中可能會變化。

## 複雜度分析

此方法只遍歷一次陣列，且每個循環的操作至多也不過僅是常數次，因此時間複雜度是$O(n)$。

## Reference

- Remove Element | 九章算法

# First Missing Positive

## Question

- 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.
```

## 題解

容易想到的方案是先排序，然後遍歷求得缺的最小整數。排序算法中常用的基於比較的方法時間複雜度的理論下界為 $O(n \log n)$, 不符題目要求。常見的能達到線性時間複雜度的排序算法有 基數排序，計數排序 和 桶排序。

基數排序顯然不太適合這道題，計數排序對元素落在一定區間且重複值較多的情況十分有效，且需要額外的 $O(n)$ 空間，對這道題不太合適。最後就只剩下桶排序了，桶排序通常需要按照一定規則將值放入桶中，一般需要額外的 $O(n)$ 空間，乍看之下似乎不太適合在這道題中使用，但是若能設定一定的規則原地交換原數組的值呢？這道題的難點就在於這種規則的設定。

設想我們對給定數組使用桶排序的思想排序，第一個桶放1，第二個桶放2，如果找不到相應的數，則相應的桶的值不變(可能為負值，也可能為其他值)。

那麼怎麼才能做到原地排序呢？即若 $A[i] = x$, 則將 x 放到它該去的地方 - $A[x-1] = x$, 同時將原來 $A[x-1]$ 地方的值交換給 $A[i]$.

排好序後遍歷桶，如果不滿足 $f[i] = i + 1$, 那麼警察叔叔就是它了！如果都滿足條件怎麼辦？那就返回給定數組大小再加1唄。

### 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 (0 < A[i]  && 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;
    }
};
```

## 源碼分析

核心程式為那幾行交換，但是要正確處理各種邊界條件則要下一番功夫了，要能正常的交換，需滿足以下幾個條件：

1. `A[i]` 為正數，負數和零都無法在桶中找到生存空間...
2. `A[i] \leq size` 當前索引處的值不能比原陣列容量大，大了的話也沒用啊，一定不是缺的第一個正數。
3. `A[i] != i + 1` , 都滿足條件了就不用交換了。
4. `A[i] != A[A[i] - 1]` , 避免欲交換的值和自身相同，否則有重複值時會產生死循環。

如果滿足以上四個條件就可以愉快地交換彼此了，使用 `while` 循環處理，此時 `i` 並不自增，直到將所有滿足條件的索引處理完。

注意交換的寫法，若寫成

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

這又是滿滿的 bug :( 因為在第三行中 `A[i]` 已不再是之前的值，第二行賦值時已經改變，故源碼中的寫法比較安全。

最後遍歷桶排序後的數組，若在數組大小範圍內找到不滿足條件的解，直接返回，否則就意味著原數組給的元素都是從1開始的連續正整數，返回數組大小加1即可。

## 複雜度分析

「桶排序」需要遍歷一次原數組，考慮到 `while` 循環也需要一定次數的遍歷，故時間複雜度至少為 $O(n)$. 最後求索引值最多遍歷一次排序後數組，時間複雜度最高為 $O(n)$, 用到了 `temp` 作為中間交換，空間複雜度為 $O(1)$.

## Reference


- Find First Missing Positive | N00tc0d3r
- LeetCode: First Missing Positive 解題報告 - Yu's Garden - 博客園
- First Missing Positive | 九章算法

# 2 Sum

## Question

- 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 ta
rget number.

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
```

## 題解1 - 哈希表

找兩數之和是否為 `target` , 如果是找數組中一個值為 `target` 該多好啊！遍歷一次就知道了，我只想說，too naive... 難道要將數組中所有元素的兩兩組合都求出來與 `target` 比較嗎？時間複雜度顯然為 $O(n^2)$, 顯然不符題目要求。找一個數時直接遍歷即可，那麼可不可以將兩個數之和轉換為找一個數呢？我們先來看看兩數之和為 `target` 所對應的判斷條件—— $x_i + x_j = target$, 可進一步轉化為 $x_i = target - x_j$, 其中 $i$ 和 $j$ 為數組中的下標。一段神奇的數學推理就將找兩數之和轉化為了找一個數是否在數組中了！可見數學是多麼的重要...

基本思路有了，現在就來看看怎麼實現，顯然我們需要額外的空間(也就是哈希表)來保存已經處理過的 $x_j$, 如果不滿足等式條件，那麼我們就往後遍歷，並把之前的元素加入到哈希表中，如果 `target` 減去當前索引後的值在哈希表中找到了，那麼就將哈希表中相應的索引返回，大功告成！

### C++

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

```
        const int length = nums.size();
        if (0 == length) {
            return result;
        }

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

        return result;
    }
};
```

## 源碼分析

1. 異常處理。
2. 使用 C++ 11 中的哈希表實現 `unordered_map` 映射值和索引。
3. 找到滿足條件的解就返回，找不到就加入哈希表中。注意題中要求返回索引值的含義。

## 複雜度分析

哈希表用了和數組等長的空間，空間複雜度為 $O(n)$, 遍歷一次數組，時間複雜度為 $O(n)$.

## 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)
```

## 源碼分析

Python 中的 `dict` 就是天然的哈希表，使用 enumerate 可以同時返回索引和值，甚為方便。按題意似乎是要返回 list, 但個人感覺返回 tuple 更為合理。最後如果未找到符合題意的索引，返回 `(-1, -1)` .

# 題解2 - 排序後使用兩根指針

但凡可以用空間換時間的做法，往往也可以使用時間換空間。另外一個容易想到的思路就是先對數組排序，然後使用兩根指針分別指向首尾元素，逐步向中間靠攏，直至找到滿足條件的索引為止。

## 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;
    }
};
```

## 源碼分析

1. 異常處理。
2. 使用 `length` 保存數組的長度，避免反複調用 `nums.size()` 造成性能損失。
3. 使用 `pair` 組合排序前的值和索引，避免排序後找不到原有索引信息。
4. 使用標準庫函數排序。
5. 兩根指針指頭尾，逐步靠攏。

## 複雜度分析

遍歷一次原數組得到 `pair` 類型的新數組，時間複雜度為 $O(n)$, 空間複雜度也為 $O(n)$. 標准庫中的排序方法時間複雜度近似為 $O(n \log n)$, 兩根指針遍歷數組時間複雜度為 $O(n)$.

> lintcode 上的題要求時間複雜度在 $O(n \log n)$ 時，空間複雜度為 $O(1)$, 但問題是排序後索引會亂掉，如果要保存之前的索引，空間複雜度一定是 $O(n)$，所以個人認為不存在較為簡潔的 $O(1)$ 實現。如果一定要 $O(n)$ 的空間複雜度，那麼只能用暴搜了，此時的時間複雜度為 $O(n^2)$.

# 3 Sum

## Question

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

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

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

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

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

## 題解1 - 排序 + 哈希表 + 2 Sum

相比之前的 2 Sum, 3 Sum 又多加了一個數，按照之前 2 Sum 的分解為『1 Sum + 1 Sum』的思路，我們同樣可以將 3 Sum 分解為『1 Sum + 2 Sum』的問題，具體就是首先對原陣列排序，排序後選出第一個元素，隨後在剩下的元素中使用 2 Sum 的解法。

## Python

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

        numbers.sort()
        for i in xrange(length):
            target = 0 - numbers[i]
            # 2 Sum
            hashmap = {}
```

```
            for j in xrange(i + 1, length):
                item_j = numbers[j]
                if (target - item_j) in hashmap:
                    triplet = [numbers[i], target - item_j, item_j]
                    if triplet not in triplets:
                        triplets.append(triplet)
                else:
                    hashmap[item_j] = j

        return triplets
```

## 源碼分析

1. 異常處理，對長度小於3的直接返回。
2. 排序輸入陣列，有助於提高效率和返回有序列表。
3. 循環遍歷排序後陣列，先取出一個元素，隨後求得 2 Sum 中需要的目標數。
4. 由於本題中最後返回結果不能重複，在加入到最終返回值之前查重。

由於排序後的元素已經按照大小順序排列，且在2 Sum 中先遍歷的元素較小，所以無需對列表內元素再排序。

## 複雜度分析

排序時間複雜度 $O(n \log n)$, 兩重 `for` 循環，時間複雜度近似為 $O(n^2)$，使用哈希表(字典)實現，空間複雜度為 $O(n)$.

目前這段源碼為比較簡易的實現，leetcode 上的運行時間為500 + ms, 還有較大的優化空間，嗯，後續再進行優化。

## C++

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

        int ans = 0;

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

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

```
            int k = num.size() - 1;

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

                if (ans == 0)
                {
                    result.push_back({num[i], num[j], num[k]});
                    ++j;
                    while (j < num.size() && num[j] == num[j - 1])
                        ++j;
                    --k;
                    while (k >= 0 && num[k] == num[k + 1])
                        --k;
                }
                else if (ans > 0)
                    --k;
                else
                    ++j;
            }
        }

        return result;
    }
};
```

## Java

```
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        // Assumptions: array is not null, array.length >= 3
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            int left = i + 1;
            int right = nums.length - 1;
            while (left < right) {
                int tmp = nums[left] + nums[right];
                if (tmp + nums[i] == 0) {
                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    left++;
                    while (left < right && nums[left] == nums[left - 1]) {
                        left++;
                    }
                } else if (tmp + nums[i] < 0) {
```

```
                    left++;
                } else {
                    right--;
                }
            }
        }
        return result;
    }
}
```

## 源碼分析

同python解法不同，沒有使用hash map

```
S = {-1 0 1 2 -1 -4}
排序後：
S = {-4 -1 -1 0 1 2}
      ↑  ↑        ↑
      i  j        k
         →        ←
i每輪只走一步，j和k根據S[i]+S[j]+S[k]=ans和0的關係進行移動，且j只向後走（即S[j]只增大），k只向前走（即S[k]只減小）
如果ans>0說明S[k]過大，k向前移；如果ans<0說明S[j]過小，j向後移；ans==0即為所求。
至於如何取到所有解，看程式碼即可理解，不再贅述。
```

## 複雜度分析

外循環i走了n輪,每輪j和k一共走n-i步，所以時間複雜度為$O(n^2)$。 最終運行時間為52ms

## Reference

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

# 3 Sum Closest

## Question

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

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

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

## 題解1 - 排序 + 2 Sum + 兩根指標 + 優化過濾

和 3 Sum 的思路接近，首先對原陣列排序，隨後將3 Sum 的題拆解為『1 Sum + 2 Sum』的題，對於 Closest 的題使用兩根指標而不是哈希表的方法較為方便。對於有序陣列來說，在查找 Cloest 的值時其實是有較大的優化空間的。

### Python

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

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

```
                if larger_count > 1:
                    return result

        while (start < end):
            sum3 = numbers[start] + numbers[end] + item_i
            if abs(sum3 - target) < abs(result - target):
                result = sum3

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

    return result
```

## 源碼分析

1. leetcode 上不能自己導入 `sys` 包，保險起見就初始化了 `result` 為還算較大的數，作為異常的返回值。
2. 對陣列進行排序。
3. 依次遍歷排序後的陣列，取出一個元素 `item_i` 後即轉化為『2 Sum Cloest』問題。『2 Sum Cloest』的起始元素索引為 `i + 1` ，之前的元素不能參與其中。
4. 優化一——由於已經對原陣列排序，故遍歷原陣列時比較最小的三個元素和 `target` 值，若第二次大於 `target` 果斷就此罷休，後面的值肯定越來越大。
5. 兩根指標求『2 Sum Cloest』，比較 `sum3` 和 `result` 與 `target` 的差值的絕對值，更新 `result` 為較小的絕對值。
6. 再度對『2 Sum Cloest』進行優化，仍然利用有序陣列的特點，若處於『一大一小』的臨界值時就可以馬上退出了，後面的元素與 `target` 之差的絕對值只會越來越大。

## 複雜度分析

對原陣列排序，平均時間複雜度為 $O(n \log n)$, 兩重 `for` 循環，由於有兩處優化，故最壞的時間複雜度才是 $O(n^2)$, 使用了 `result` 作為臨時值保存最接近 `target` 的值，兩處優化各使用了一個輔助變量，空間複雜度 $O(1)$.

## C++

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

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

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

## 源碼分析

和前面3Sum解法相似，同理使用i,j,k三個指標進行循環。
區別在於3sum中的target為0，這裡新增一個變數用於比較哪組數據與target更為相近，並讓相對應的指標調整使之更近。

## 複雜度分析

時間複雜度同理為$O(n^2)$ 運行時間 16ms

## Reference

- 3Sum Closest | 九章算法

# Remove Duplicates from Sorted Array

## Question

- lintcode: (100) Remove Duplicates from Sorted Array

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

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

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

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

Example
```

## 題解

使用雙指標(下標)，一個指標(下標)遍歷vector數組，另一個指標(下標)只取不重複的數置於原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;
    }
};
```

## 源碼分析

注意最後需要返回的是 `++size` 或者 `size + 1`

# Remove Duplicates from Sorted Array II

## Question

- 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
```

## 題解

在上題基礎上加了限制條件元素最多可重複出現兩次。因此可以在原題的基礎上添加一變量跟蹤元素重複出現的次數，小於指定值時執行賦值操作。但是需要注意的是重複出現次數 `occurence` 的初始值(從1開始，而不是0)和reset的時機。

### C++

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

```
        }

        return ++size;
    }
};
```

## 源碼分析

1. 在數組元素小於3(即為2)時可直接返回vector數組大小。
2. 初始化 `occurence` 的值為1，而不是0. 理解起來也方便些。
3. 初始化下標值 `i` 從1開始
   - `nums[size] != nums[i]` 時遞增 `size` 並賦值，同時重置 `occurence` 的值為1
   - `(nums[size] == nums[i])` 時，首先判斷 `occurence` 的值是否小於2，小於2則先遞增 `size` ，隨後將 `nums[i]` 的值賦給 `nums[size]` 。這裡由於小標 `i` 從1開始，免去了對 `i` 為0的特殊情況考慮。
4. 最後返回 `size + 1` ，即為 `++size`

# Merge Sorted Array

## Question

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

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

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

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

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

## 題解

因為本題有 in-place 的限制，故必須從陣列末尾的兩個元素開始比較；否則就會產生挪動，一旦挪動就會是 $O(n^2)$ 的。 自尾部向首部逐個比較兩個陣列內的元素，取較大的置於陣列 A 中。由於 A 的容量較 B 大，故最後 `m == 0` 或者 `n == 0` 時僅需處理 B 中的元素，因為 A 中的元素已經在 A 中，無需處理。

## Python

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

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

```
            else:
                A[index] = B[n - 1]
                n -= 1
            index -= 1

        # B has elements left
        while n > 0:
            A[index] = B[n - 1]
            n -= 1
            index -= 1
```

## C++

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

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

## Java

```
class Solution {
    /**
```

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

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

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

## 源碼分析

第一個 while 只能用條件與(conditional AND)。

## 複雜度分析

最壞情況下需要遍歷兩個陣列中所有元素，時間複雜度為 $O(n)$. 空間複雜度 $O(1)$.

# Merge Sorted Array II

## Question

- lintcode: (64) Merge Sorted Array II

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

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

B=[2,4,5,6]

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

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

## 題解

上題要求 in-place, 此題要求返回新陣列。由於可以生成新陣列，故使用常規思路按順序遍歷即可。

## Python

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

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

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

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

        return C
```

## C++

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

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

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

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

```
        return C;
    }
};
```

## Java

```
class Solution {
    /**
     * @param A and B: sorted integer array A and B.
     * @return: A new sorted integer array
     */
    public ArrayList<Integer> mergeSortedArray(ArrayList<Integer> A, ArrayList<Inte
ger> B) {
        if (A == null || A.isEmpty()) return B;
        if (B == null || B.isEmpty()) return A;

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

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

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

        return C;
    }
}
```

## 源碼分析

分三步走，後面分別單獨處理剩餘的元素。

## 複雜度分析

遍歷 A, B 陣列各一次，時間複雜度 $O(n)$, 空間複雜度 $O(1)$.

### Challenge

兩個倒排列表，一個特別大，一個特別小，如何 Merge？此時應該考慮用一個二分法插入小的，即記憶體拷貝。

# Search - 搜索

本章主要總結二分搜索相關的題目。

- 能使用二分搜索的前提是數組已排序。
- 二分搜索的使用場景：（1）可轉換為find the first/last position of...（2）時間複雜度至少為 $O(\log n)$。
- 遞迴和迭代的使用場景：能用迭代就用迭代，特別複雜時採用遞迴。

# Binary Search - 二分搜尋

## Question

- 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 inde
x of this number in O(log n) time complexity.

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

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

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

## 題解

對於已排序升序陣列，使用二分搜尋可滿足複雜度要求，注意陣列中可能有重複值。

## Java

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

        int start = 0;
        int end = nums.length - 1;
        int mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2; // avoid overflow when (end + start)
```

```
            if (target < nums[mid]) {
                end = mid;
            } else if (target > nums[mid]) {
                start = mid;
            } else {
                end = mid;
            }
        }

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

        return -1;
    }
}
```

## 源碼分析

1. 首先對輸入做異常處理，陣列為空或者長度為0。
2. 初始化 `start, end, mid` 三個變量，注意mid的求值方法，可以防止兩個整型值相加時溢出。
3. **使用迭代而不是遞迴**進行二分搜尋，因為工程中遞迴寫法存在潛在溢出的可能。
4. while終止條件應為 `start + 1 < end` 而不是 `start <= end` ， `start == end` 時可能出現死循環。**即循環終止條件是相鄰或相交元素時退出。**
5. 迭代終止時target應為start或者end中的一個——由上述循環終止條件有兩個，具體誰先誰後視題目是找 first position or last position 而定。
6. 賦值語句 `end = mid` 有兩個條件是相同的，可以選擇寫到一塊。
7. 配合while終止條件 `start + 1 < end` （相鄰即退出）的賦值語句mid永遠沒有 `+1` 或者 `-1` ，這樣不會死循環。

## C++

```
class Solution {
public:
    /**
     * @param nums: The integer array.
     * @param target: Target number to find.
     * @return: The first position of target. Position starts from 0.
     */
    int binarySearch(vector<int> &nums, int target) {
        if( nums.size() == 0 ) return -1;

        int lo = 0, hi = nums.size();
        while(lo < hi){
```

```
            int mi = lo + (hi - lo)/2;
            if(nums[mi] < target)
                lo = mi + 1;
            else
                hi = mi;
        }

        if(nums[lo] == target) return lo;
        return -1;
    }
};
```

## 源碼分析

遇到需要處理陣列範圍的問題，由於C/C++語言本身的特性，統一使用開閉區間表示index範圍將有許多好處，[lo, hi)表示包含lo但不包含hi的區間。比方說，如果要遍歷這個區間，迴圈的條件可以寫為 `for(i = lo; i < hi; i++)` 這類常用的方式，如果要求此段區間長度可用 `int length = hi - lo;` ，另外在很多邊界條件的判斷上也會比較簡潔。實際上在STL裡的iterator也是使用了用類似的概念，一個容器的end()表示的是一個已經超出指定範圍的iterator。以此題來說，可以看出C++的實現方法確實比較簡潔。

1. 終止條件簡單設定為 `lo < hi` ，事實上觀察調整 `lo` 與 `hi` 範圍的過程，終止的時候一定是 `lo == hi` 。
2. 觀察 `lo` 的更新條件，是當 `nums[mi]` 比目標值小時將 `lo` 更新為 `mi + 1` ，也就是說， `lo` 可以保證下界一定會不斷排除比 `target` 小的值，其餘狀況每次循環 `hi` 則減少範圍，因此等到循環終止之後， `lo` 就會指到**不小於 `target` 的最小元素**，我們再將這個元素與 `target` 比較，就知道是否有找到，沒有的話就返回-1

# Search Insert Position

## Question

- lintcode: (60) Search Insert Position

```
Given a sorted array and a target value, return the index if the target is found. I
f not, return the index where it would be if it were inserted in order.

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
```

## 題解

應該把二分法的問題拆解為 `find the first/last position of...` 的問題。由最原始的二分搜尋可找到不小於目標整數的最小下標。返回此下標即可。

### Java

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

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

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

```
            } else if (A[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }

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

## 源碼分析

要注意例子中的第三個, [1,3,5,6], 7 → 4，即找不到要找的數字的情況，此時應返回數組長度，即代碼中最後一個else的賦值語句 `return end + 1;`

## C++

```
class Solution {
    /**
     * param A : an integer sorted array
     * param target :  an integer to be inserted
     * return : an integer
     */
public:
    int searchInsert(vector<int> &A, int target) {
        int N = A.size();
        if (N == 0) return 0;
        if (A[N-1] < target) return N;
        int lo = 0, hi = N;
        while (lo < hi) {
            int mi = lo + (hi - lo)/2;
            if (A[mi] < target)
                lo = mi + 1;
            else
                hi = mi;
        }
        return lo;
    }
};
```

## 源碼分析

與lintcode - (14) Binary Search類似，在C++的解法裡我們也使用了[lo, hi)的表示方法，而題意是找出不小於 `target` 的最小位置，因此每次二分搜尋的循環裡如果發現 `A[m]` 已經小於 `target` ，就應該將下界 `lo` 往右推，其他狀況則將上界 `hi` 向左移動，然而必須注意的是如果 `target` 比陣列中所有元素都大，必須返回 `ho` 位置，然而此上下界的表示方法是不可能返回 `ho` 的，所以還有另外加一個判斷式，如果 `target` 已經大於陣列中最後一個元素，就直接返回其位置。

# Search for a Range

## Question

- lintcode: (61) Search for a Range

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

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].
```

## 題解

Search for a range 的題目可以拆解為找 first & last position 的題目，即要做兩次二分。由上題二分查找可找到滿足條件的左邊界，因此只需要再將右邊界找出即可。注意到在 `(target == nums[mid]` 時賦值語句為 `end = mid` ，將其改為 `start = mid` 即可找到右邊界，解畢。

## Java

```java
/**
 * 本代碼fork自九章算法。沒有版權歡迎轉發。
 * 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
    }
}
```

## 源碼分析

1. 首先對輸入做異常處理，數組為空或者長度為0
2. 初始化 `start, end, mid` 三個變量，注意mid的求值方法，可以防止兩個整型值相加時溢出
3. **使用迭代而不是遞歸**進行二分查找
4. while終止條件應為 `start + 1 < end` 而不是 `start <= end` ， `start == end` 時可能出現死循環
5. 先求左邊界，迭代終止時先判斷 `A.get(start) == target` ，再判斷 `A.get(end) == target` ，因為迭代終止時target必取start或end中的一個，而end又大於start，取左邊界即為start.
6. 再求右邊界，迭代終止時先判斷 `A.get(end) == target` ，再判斷 `A.get(start) == target`
7. 兩次二分查找除了終止條件不同，中間邏輯也不同，即當 `A.get(mid) == target` 如果是左邊界（first postion），中間邏輯是 `end = mid` ；若是右邊界（last position），中間邏輯是 `start = mid`
8. 兩次二分查找中間勿忘記重置 `start, end` 的變量值。

## C++

```
class Solution {
    /**
     *@param A : an integer sorted array
     *@param target :  an integer to be inserted
     *return : a list of length 2, [index1, index2]
     */
public:
    vector<int> searchRange(vector<int> &A, int target) {
        // good, fail are the result
        // When found, returns good, otherwise returns fail
        int N = A.size();
        vector<int> fail = {-1, -1};
        if(N == 0)
            return fail;
        vector<int> good;

        // search for starting position
        int lo = 0, hi = N;
        while(lo < hi){
            int m = lo + (hi- lo)/2;
            if(A[m] < target)
                lo = m + 1;
            else
                hi = m;
        }

        if(A[lo] != target)
            return fail;

        good.push_back(lo);

        // search for ending position
        lo = 0; hi = N;
```

```
        while(lo < hi){
            int m = lo + (hi - lo)/2;
            if(target < A[m])
                hi = m;
            else
                lo = m + 1;
        }
        good.push_back(lo - 1);

        return good;
    }
};
```

## 源碼分析

與前面題目類似，此題是將兩個子題組合起來，前半為找出"不小於target的最左元素"，後半是"不大於target的最右元素"，同樣的，使用開閉區間[lo, hi)仍然可以簡潔的處理各種邊界條件，僅須注意在解第二個子題"不大於target的最右元素"時，由於每次 `lo` 更新時都至少加1，最後會落在我們要求的位置的下一個，因此記得減1回來，若直覺難以理解，可以使用一個例子在紙上推一次每個步驟就可以體會。

# First Bad Version

## Question

- leetcode: First Bad Version
- 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 in the code case, so it caused itself and the following versions are
all failed in the unit tests.
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 different language. For example, Java is VersionControl.isBadVersion.

Example
Given n=5

Call isBadVersion(3), get false

Call isBadVersion(5), get true

Call isBadVersion(4), get true

return 4 is the first bad version

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

題 Search for a Range 的變形，找出左邊界即可。

## Java

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

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

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

## C++

```
/**
 * class VersionControl {
 *     public:
 *     static bool isBadVersion(int k);
 * }
 * you can use VersionControl::isBadVersion(k) to judge wether
```

```
 * the kth code version is bad or not.
*/
class Solution {
public:
    /**
     * @param n: An integers.
     * @return: An integer which is the first bad version.
     */
    int findFirstBadVersion(int n) {
        if (n < 1) {
            return -1;
        }

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

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

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

## 源碼分析

找左邊界和Search for a Range類似，但是最好要考慮到有可能end處也為good version，此部分異常也可放在開始的時候處理。

## Python

```
#class VersionControl:
#    @classmethod
#    def isBadVersion(cls, id)
#        # Run unit tests to check whether verison `id` is a bad version
#        # return true if unit tests passed else false.
# You can use VersionControl.isBadVersion(10) to check whether version 10 is a
```

```python
# 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
```

## Leetcode版題解

很明顯使用二分搜索，此題的測試只有Bad version必定出現的情況，會不會全部都是好的，可以向面試官詢問清楚，二分搜索的範圍，仍然使用下標範圍[0, n)控制邊界，要注意的是返回值是產品的編號，記得+1。另外直接使用產品編號[1, n+1)是行不通的，因為當n是INT_MAX時就會出現問題。

```cpp
// Forward declaration of isBadVersion API.
bool isBadVersion(int version);

class Solution {
public:
    int firstBadVersion(int n) {
        if(isBadVersion(1)) return 1;
        int lo = 0, hi = n;
        while(lo < hi){
            int m = lo + (hi - lo)/2;
            if(!isBadVersion(m) and isBadVersion(m+1))
                return m+1;
            else if(isBadVersion(m) and isBadVersion(m+1))
                hi = m;
            else
                lo = m + 1;
        }
    }
```

```
};
```

# Find Peak Element

## Question

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

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

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

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

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

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

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

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

Challenge
Time complexity O(logN)
```

## 題解1 - lintcode

由時間複雜度的暗示可知應使用二分搜索。首先分析若使用傳統的二分搜索，若 `A[mid] > A[mid - 1] && A[mid] < A[mid + 1]` ，則找到一個peak為A[mid]；若 `A[mid - 1] > A[mid]` ，則A[mid]左側必定存在一個peak，可用反證法證明：若左側不存在peak，則A[mid]左側元素必滿足 `A[0] > A[1] > ... > A[mid -1] > A[mid]` ，與已知 `A[0] < A[1]` 矛盾，證畢。同理可得若 `A[mid + 1] > A[mid]` ，則A[mid]右側必定存在一個peak。如此迭代即可得解。

備注：如果本題是找 first/last peak，就不能用二分法了。

## Python

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

```
        l, r = 0, len(A) - 1
        while l + 1 < r:
            mid = l + (r - l) / 2
            if A[mid] < A[mid - 1]:
                r = mid
            elif A[mid] < A[mid + 1]:
                l = mid
            else:
                return mid
        mid = l if A[l] > A[r] else r
        return mid
```

## C++

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

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

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

## Java

```
class Solution {
    /**
     * @param A: An integers array.
     * @return: return any of peek positions.
```

```
     */
    public int findPeak(int[] A) {
        if (A == null || A.length == 0) return -1;

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

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

## 題解2 - leetcode

leetcode 上的題和 lintcode 上有細微的變化，題目如下：

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

Given an input array where num[i] ≠ num[i+1],
find a peak element and return its index.

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

You may imagine that num[-1] = num[n] = -∞.

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

click to show spoilers.

Note:
Your solution should be in logarithmic complexity.
```

如果一開始做的是 leetcode 上的版本而不是 lintcode 上的話，這道題難度要大一些。有了以上的分析基礎再來刷 leetcode 上的這道題就是小 case 了，注意題中的關鍵提示 `num[-1] = num[n] = -∞`，雖然不像 lintcode 上那麼直接，但是稍微變通下也能想到。即 `num[-1] < num[0] && num[n-1] > num[n]`，

那麼問題來了，這樣一來就不能確定峰值一定存在了，因為給定數組為單調序列的話就咩有峰值了，但是實際情況是——題中有負無窮的假設，也就是說在單調序列的情況下，峰值為數組首部或者尾部元素，誰大就是誰了。

## C++

```
class Solution {
public:
    int findPeakElement(vector<int>& arr) {
        int N = arr.size();
        int lo = 0, hi = N;
        while(lo < hi) {
            int mi = lo + (hi - lo)/2;
            if( (mi == 0 || arr[mi-1] <= arr[mi] ) && (mi == N-1 || arr[mi] >= arr[
mi+1]) )
                return mi;
            else if((mi == 0 || arr[mi-1] <= arr[mi] ))
                lo = mi + 1;
            else
                hi = mi;
        }
        return -1;
    }
};
```

## Java

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

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

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

```
}
```

## 源碼分析

典型的二分法模板應用，需要注意的是需要考慮單調序列的特殊情況。當然也可使用緊湊一點的實現如改寫循環條件為 `l < r` ，這樣就不用考慮單調序列了，見實現2.

## 複雜度分析

二分法，時間複雜度 $O(\log n)$.

## Java - compact implementation[leetcode_discussion]

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

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

        return start;
    }
}
```

C++ 的代碼可參考 Java 或者 @xuewei4d 的實現。

> leetcode 和 lintcode 上給的方法名不一樣，leetcode 上的為 `findPeakElement` 而 lintcode 上為 `findPeak` ，弄混的話會編譯錯誤。

## Reference

- > leetcode_discussion. Java - Binary-Search Solution - Leetcode Discuss ↩

# Sqrt x

## Question

- leetcode: Sqrt(x) | LeetCode OJ
- lintcode: (141) Sqrt(x)

## 題解 - 二分搜索

由於只需要求整數部分，故對於任意正整數 $x$, 設其整數部分為 $k$, 顯然有 $1 \leq k \leq x$, 求解 $k$ 的值也就轉化為了在有序陣列中查找滿足某種約束條件的元素，顯然二分搜索是解決此類問題的良方。

### 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
```

## 源碼分析

1. 異常檢測，先處理小於等於0的值。
2. 使用二分搜索的經典模板，注意不能使用 `start < end` , 否則在給定值1時產生死循環。
3. 最後返回平方根的整數部分 `start` .

二分搜索過程很好理解，關鍵是最後的返回結果還需不需要判斷？比如是取 start, end, 還是 mid? 我們首先來分析下二分搜索的循環條件，由 `while` 循環條件 `start + 1 < end` 可知， `start` 和 `end` 只可能有兩種關系，一個是 `end == 1 || end ==2` 這一特殊情況，返回值均為1，另一個就是循環終止

時 `start` 恰好在 `end` 前一個元素。設值 x 的整數部分為 k, 那麼在執行二分搜索的過程中

$start \leq k \leq end$ 關系一直存在，也就是說在沒有找到 $mid^2 == x$ 時，循環退出時有 $start < k < end$, 取整的話顯然就是 `start` 了。

## C++

```
class Solution{
public:
    int mySqrt(int x) {
        if(x <= 1) return x;
        int lo = 2, hi = x;
        while(lo < hi){
            int m = lo + (hi - lo)/2;
            int q = x/m;
            if(q == m and x % m == 0)
                return m;
            else if(q < m)
                hi = m;
            else
                lo = m + 1;
        }

        return lo - 1;
    }
};
```

## 源碼分析

此題依然可以被翻譯成"找不大於target的$x^2$"，而所有待選的自然數當然是有序數列，因此同樣可以用二分搜索的思維解題，然而此題不會出現重複元素，因此可以增加一個相等就返回的條件，另外這邊我們同樣使用[lo, hi)的標示法來處理邊界條件，可以參照[Search for a range]，就不再贅述。另外特別注意，判斷找到的條件不是用 `m * m == x` 而是 `x / m == m` ，這是因為 `x * x` 可能會超出 `INT_MAX` 而溢位，因此用除法可以解決這個問題，再輔以餘數判斷是否整除以及下一步的走法。

## 複雜度分析

經典的二分搜索，時間複雜度為 $O(\log n)$, 使用了 `start` , `end` , `mid` 變量，空間複雜度為 $O(1)$.

除了使用二分法求平方根近似解之外，還可使用牛頓迭代法進一步提高運算效率，欲知後事如何，請猛戳 求平方根sqrt()函數的底層算法效率問題 -- 簡明現代魔法，不得不感歎演算法的魔力！

# Count 1 in Binary

## Question

- lintcode: (365) Count 1 in Binary

```
Count how many 1 in binary representation of a 32-bit integer.

Example
Given 32, return 1

Given 5, return 2

Given 1023, return 9

Challenge
If the integer is n bits with m 1 bits. Can you do it in O(m) time?
```

## 題解

題 O1 Check Power of 2 的進階版， `x & (x - 1)` 的含義爲去掉二進制數中1的最後一位，無論 x 是正數還是負數都成立。

### C++

```cpp
class Solution {
public:
    /**
     * @param num: an integer
     * @return: an integer, the number of ones in num
     */
    int countOnes(int num) {
        int count=0;
        while (num) {
            num &= num-1;
            count++;
        }
        return count;
    }
};
```

### Java

```
public class Solution {
    /**
     * @param num: an integer
     * @return: an integer, the number of ones in num
     */
    public int countOnes(int num) {
        int count = 0;
        while (num != 0) {
            num = num & (num - 1);
            count++;
        }

        return count;
    }
}
```

## 源碼分析

累加計數器即可。

## 複雜度分析

這種算法依賴於數中1的個數，時間複雜度爲 $O(m)$. 空間複雜度 $O(1)$.

## Reference

- Number of 1 bits | LeetCode - 評論中有關於不同演算法性能的討論

# A plus B Problem

## Question

- lintcode: (1) A + B Problem

```
Write a function that add two numbers A and B.
You should not use + or any arithmetic operators.


Example
Given a=1 and b=2 return 3

Note
There is no need to read data from standard input stream.
Both parameters are given in function aplusb,
you job is to calculate the sum and return it.
Challenge
Of course you can just return a + b to get accepted.
But Can you challenge not do it like that?
Clarification
Are a and b both 32-bit integers?
Yes.
Can I use bit operation?

Sure you can.
```

## 題解

不用加減法實現加法，類似數字電路中的全加器 (Full Adder)，XOR 求得部分和，OR 求得進位，最後將進位作爲加法器的輸入，典型的遞迴實現思路。

### Java

```
class Solution {
    /*
     * param a: The first integer
     * param b: The second integer
     * return: The sum of a and b
     */
    public int aplusb(int a, int b) {
        int result = a ^ b;
        int carry = a & b;
        carry <<= 1;
        if (carry != 0) {
```

```
            result = aplusb(result, carry);
        }

        return result;
    }
}
```

## 源碼分析

遞迴步爲進位是否爲0，爲0時返回。

## 複雜度分析

取決於進位，近似爲 $O(1)$. 使用了部分額外變量，空間複雜度爲 $O(1)$.

# Plus One

## Question

- leetcode: Plus One | LeetCode OJ
- lintcode: (407) Plus One

### Problem Statement

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

#### Example

Given [1,2,3] which represents 123, return [1,2,4].

Given [9,9,9] which represents 999, return [1,0,0,0].

## 題解

又是一道兩個整數按數位相加的題，自後往前累加，處理下進位即可。這道題中是加1，其實還可以擴展至加2，加3等。

### Java

```
public class Solution {
    /**
     * @param digits a number represented as an array of digits
     * @return the result
     */
    public int[] plusOne(int[] digits) {
        return plusDigit(digits, 1);
    }

    private int[] plusDigit(int[] digits, int digit) {
        if (digits == null || digits.length == 0) return null;

        // regard digit(0~9) as carry
        int carry = digit;
        int[] result = new int[digits.length];
        for (int i = digits.length - 1; i >= 0; i--) {
            result[i] = (digits[i] + carry) % 10;
            carry = (digits[i] + carry) / 10;
        }
```

```
        // carry == 1
        if (carry == 1) {
            int[] finalResult = new int[result.length + 1];
            finalResult[0] = 1;
            return finalResult;
        }

        return result;
    }
}
```

## C++

```
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        int carry = 1;
        for(int i = digits.size() - 1; i >= 0; i--){
            digits[i] += carry;
            carry = digits[i] / 10;
            digits[i] %= 10;
        }

        if(carry == 1){
            digits.insert(digits.begin(), 1);
        }
        return digits;
    }
};
```

## 源碼分析

源碼中單獨實現了加任何數(0~9)的私有方法，更為通用，對於末尾第一個數，可以將要加的數當做進位處理，這樣就不必單獨區分最後一位了，十分優雅！

## 複雜度分析

Java 中需要返回數組，而這個數組在處理之前是不知道大小的，故需要對最後一個進位單獨處理。時間複雜度 $O(n)$, 空間複雜度在最後一位有進位時惡化為 $O(n)$, 當然也可以通過兩次循環使得空間複雜度為 $O(1)$.

## Reference

- Soulmachine 的 leetcode 題解，將要加的數當做進位處理就是從這學到的。

# Linked List - 鏈表

本節包含鏈表的一些常用操作，如刪除、插入和合併等。

常見錯誤有 遍歷鏈表不向前遞推節點，遍歷鏈表前未保存頭節點，返回鏈表節點指標錯誤。

# Remove Duplicates from Sorted List

## Question

- leetcode: Remove Duplicates from Sorted List | LeetCode OJ
- lintcode: (112) Remove Duplicates from Sorted List

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

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

## 題解

遍歷之，遇到當前節點和下一節點的值相同時，刪除下一節點，並將當前節點 `next` 值指向下一個節點的 `next` , 當前節點首先保持不變，直到相鄰節點的值不等時才移動到下一節點。

## Python

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def deleteDuplicates(self, head):
        if head is None:
            return None

        node = head
        while node.next is not None:
            if node.val == node.next.val:
                node.next = node.next.next
            else:
                node = node.next

        return head
```

## C++

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

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

        return head;
    }
};
```

## Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
```

```
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

        ListNode node = head;
        while (node.next != null) {
            if (node.val == node.next.val) {
                node.next = node.next.next;
            } else {
                node = node.next;
            }
        }

        return head;
    }
}
```

## 源碼分析

1. 首先進行異常處理，判斷head是否為NULL
2. 遍歷鏈表， `node->val == node->next->val` 時，保存 `node->next` ，便於後面釋放記憶體(非C/C++無需手動管理記憶體)
3. 不相等時移動當前節點至下一節點，注意這個步驟必須包含在 `else` 中，否則邏輯較為複雜

`while` 循環處也可使用 `node != null && node->next != null` , 這樣就不用單獨判斷 `head` 是否為空了，但是這樣會降低遍歷的效率，因為需要判斷兩處。

## 複雜度分析

遍歷鏈表一次，時間複雜度為 $O(n)$, 使用了一個變數進行遍歷，空間複雜度為 $O(1)$.

## Reference

- Remove Duplicates from Sorted List 參考程序 | 九章

# Remove Duplicates from Sorted List II

## Question

- leetcode: Remove Duplicates from Sorted List II | LeetCode OJ
- lintcode: (113) Remove Duplicates from Sorted List II

```
Given a sorted linked list, delete all nodes that have duplicate numbers,
leaving only distinct numbers from the original list.

Example
Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.
```

## 題解

上題為保留重複值節點的一個，這題刪除全部重複節點，看似區別不大，但是考慮到鏈表頭不確定(可能被刪除，也可能保留)，因此若用傳統方式需要較多的if條件語句。這裏介紹一個**處理鏈表頭節點不確定的方法——引入dummy node.**

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

引入新的指標變數 `dummy` ，並將其next變數賦值為head，考慮到原來的鏈表頭節點可能被刪除，故應該從dummy處開始處理，這裏複用了head變數。考慮鏈表 `A->B->C` ，刪除B時，需要處理和考慮的是A和C，將A的next指向C。如果從空間使用效率考慮，可以使用head代替以上的node，含義一樣，node比較好理解點。

與上題不同的是，由於此題引入了新的節點 `dummy` ，不可再使用 `node->val == node->next->val` ，原因有二：

1. 此題需要將值相等的節點全部刪掉，而刪除鏈表的操作與節點前後兩個節點都有關系，故需要涉及三個鏈表節點。且刪除單向鏈表節點時不能刪除當前節點，只能改變當前節點的 `next` 指向的節點。
2. 在判斷val是否相等時需先確定 `node->next` 和 `node->next->next` 均不為空，否則不可對其進行取值。

說多了都是淚，先看看我的錯誤實現：

## 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;
    }
};
```

## 錯因分析

錯在什麼地方？

1. 節點dummy的初始化有問題，對class的初始化應該使用 `new`
2. 在else語句中 `node->next = node->next->next;` 改寫了 `dummy-next` 中的內容，返回的 `dummy-`

`next` 不再是隊首元素，而是隊尾元素。原因很微妙，應該使用 `node = node->next;` ，node代表節點指標變數，而node->next代表當前節點所指向的下一節點地址。具體分析可自行在紙上畫圖分析，可對指標和鏈表的理解又加深不少。

圖中上半部分為ListNode的記憶體示意圖，每個框底下為其內存地址。 `dummy` 指標本身的地址為ox7fff5d0d2500，其保存著指標值為0x7fbe7bc04c50. `head` 指標本身的地址為ox7fff5d0d2508，其保存著指標值為0x7fbe7bc04c00.

好了，接下來看看正確實現及解析。

## Python

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def deleteDuplicates(self, head):
        if head is None:
            return None
```

```
        dummy = ListNode(0)
        dummy.next = head
        node = dummy
        while node.next is not None and node.next.next is not None:
            if node.next.val == node.next.next.val:
                val_prev = node.next.val
                while node.next is not None and node.next.val == val_prev:
                    node.next = node.next.next
            else:
                node = node.next

        return dummy.next
```

## C++

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == NULL) return NULL;

        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode *node = dummy;
        while (node->next != NULL && node->next->next != NULL) {
            if (node->next->val == node->next->next->val) {
                int val_prev = node->next->val;
                // remove ListNode node->next
                while (node->next != NULL && val_prev == node->next->val) {
                    ListNode *temp = node->next;
                    node->next = node->next->next;
                    delete temp;
                }
            } else {
                node = node->next;
            }
        }

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

## Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode node = dummy;
        while(node.next != null && node.next.next != null) {
            if (node.next.val == node.next.next.val) {
                int val_prev = node.next.val;
                while (node.next != null && node.next.val == val_prev) {
                    node.next = node.next.next;
                }
            } else {
                node = node.next;
            }
        }

        return dummy.next;
    }
}
```

## 源碼分析

1. 首先考慮異常情況，head 為 NULL 時返回 NULL
2. new一個dummy變數， `dummy->next` 指向原鏈表頭。
3. 使用新變數node並設置其為dummy頭節點，遍歷用。
4. 當前節點和下一節點val相同時先保存當前值，便於while循環終止條件判斷和刪除節點。注意這一段代碼也比較精煉。
5. 最後返回 `dummy->next` ，即題目所要求的頭節點。

Python 中也可不使用 `is not None` 判斷，但是效率會低一點。

## 複雜度分析

兩個指標(node.next 和 node.next.next)遍歷，時間複雜度為 $O(2n)$. 使用了一個 dummy 和中間緩存變數，空間複雜度近似為 $O(1)$.

## Reference

- Remove Duplicates from Sorted List II | 九章

# Linked List Cycle

## Question

- 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?
```

## 題解 - 快慢指標

對於帶環鏈表的檢測，效率較高且易於實現的一種方式為使用快慢指標。快指標每次走兩步，慢指標每次走一步，如果快慢指標相遇(快慢指標所指內存為同一區域)則有環，否則快指標會一直走到 `NULL` 為止退出循環，返回 `false` .

快指標走到 `NULL` 退出循環即可確定此鏈表一定無環這個很好理解。那麼帶環的鏈表快慢指標一定會相遇嗎？先來看看下圖。

在有環的情況下，最終快慢指標一定都走在環內，加入第 `i` 次遍歷時快指標還需要 `k` 步才能追上慢指標，由於快指標比慢指標每次多走一步。那麼每遍歷一次快慢指標間的間距都會減少1，直至最終相遇。故快慢指標相遇一定能確定該鏈表有環。

### 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) {
        ListNode *fast = head, *slow = head;
        while(fast and fast->next){
            slow = slow -> next;
            fast = fast -> next -> next;
            if(slow == fast) return true;
        }
        return false;
    }
};
```

## Java

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }
```

```
        ListNode slow = head;
        ListNode fast = head;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                return true;
            }
        }
        return false;
    }
}
```

## 源碼分析

1. 異常處理，將 `head->next` 也考慮在內有助於簡化後面的代碼。
2. 慢指標初始化為 `head` , 快指標初始化為 `head` 的下一個節點，這是快慢指標初始化的一種方法，有時會簡化邊界處理，但有時會增加麻煩，比如該題的進階版。

## 複雜度分析

1. 在無環時，快指標每次走兩步走到尾部節點，遍歷的時間複雜度為 $O(n/2)$.
2. 有環時，最壞的時間複雜度近似為 $O(n)$. 最壞情況下鏈表的頭尾相接，此時快指標恰好在慢指標前一個節點，還需 n 次快慢指標相遇。最好情況和無環相同，尾節點出現環。

故總的時間複雜度可近似為 $O(n)$.

## Reference

- Linked List Cycle | 九章算法

# Reverse Linked List

## Question

- leetcode: Reverse Linked List | LeetCode OJ
- lintcode: (35) Reverse Linked List

```
Reverse a linked list.

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

Challenge
Reverse it in-place and in one-pass
```

## 題解1 - 非遞迴

聯想到同樣也可能需要翻轉的數組，在數組中由於可以利用下標隨機訪問，翻轉時使用下標即可完成。而在單向鏈表中，僅僅只知道頭節點，而且只能單向往前走，故需另尋出路。分析由 `1->2->3` 變為 `3->2->1` 的過程，由於是單向鏈表，故只能由1開始遍曆，1和2最開始的位置是 `1->2` ，最後變為 `2->1` ，故從這裡開始尋找突破口，探討如何交換1和2的節點。

```
temp = head->next;
head->next = prev;
prev = head;
head = temp;
```

要點在於維護兩個指針變量 `prev` 和 `head` , 翻轉相鄰兩個節點之前保存下一節點的值，分析如下圖所示：

1. 保存head下一節點
2. 將head所指向的下一節點改為prev
3. 將prev替換為head，波浪式前進
4. 將第一步保存的下一節點替換為head，用於下一次循環

## Python

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        prev = None
        curr = head
        while curr is not None:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        # fix head
        head = prev

        return head
```

## C++

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        ListNode *prev = NULL;
        ListNode *curr = head;
        while (curr != NULL) {
            ListNode *temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;
```

```
        return head;
    }
};
```

## Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode temp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
}
```

## 源碼分析

題解中基本分析完畢，代碼中的prev賦值操作精煉，值得借鑒。

## 複雜度分析

遍歷一次鏈表，時間複雜度為 $O(n)$, 使用了輔助變數，空間複雜度 $O(1)$.

# 題解2 - 遞迴

遞迴的終止步分三種情況討論：

1. 原鏈表為空，直接返回空鏈表即可。
2. 原鏈表僅有一個元素，返回該元素。
3. 原鏈表有兩個以上元素，由於是單向鏈表，故翻轉需要自尾部向首部逆推。

由尾部向首部逆推時大致步驟為先翻轉當前節點和下一節點，然後將當前節點指向的下一節點置空(否則會出現死循環和新生成的鏈表尾節點不指向空)，如此遞迴到頭節點為止。新鏈表的頭節點在整個遞迴過程中一直沒有變化，逐層向上返回。

## Python

```
"""
Definition of ListNode

class ListNode(object):

    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of the linked list.
    @return: You should return the head of the reversed linked list.
                  Reverse it in-place.
    """
    def reverse(self, head):
        # case1: empty list
        if head is None:
            return head
        # case2: only one element list
        if head.next is None:
            return head
        # case3: reverse from the rest after head
        newHead = self.reverse(head.next)
        # reverse between head and head->next
        head.next.next = head
        # unlink list from the rest
        head.next = None

        return newHead
```

## C++

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

```
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: The new head of reversed linked list.
     */
    ListNode *reverse(ListNode *head) {
        // case1: empty list
        if (head == NULL) return head;
        // case2: only one element list
        if (head->next == NULL) return head;
        // case3: reverse from the rest after head
        ListNode *newHead = reverse(head->next);
        // reverse between head and head->next
        head->next->next = head;
        // unlink list from the rest
        head->next = NULL;

        return newHead;
    }
};
```

## Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverse(ListNode head) {
        // case1: empty list
        if (head == null) return head;
        // case2: only one element list
        if (head.next == null) return head;
        // case3: reverse from the rest after head
        ListNode newHead = reverse(head.next);
        // reverse between head and head->next
        head.next.next = head;
        // unlink list from the rest
        head.next = null;
```

```
        return newHead;
    }
}
```

## 源碼分析

case1 和 case2 可以合在一起考慮，case3 返回的為新鏈表的頭節點，整個遞迴過程中保持不變。

## 複雜度分析

遞迴嵌套層數為 $O(n)$, 時間複雜度為 $O(n)$, 空間(不含函數堆疊空間)複雜度為 $O(1)$.

# Reference

- 全面分析再動手的習慣：鏈表的反轉問題（遞迴和非遞迴方式） - 木棉和木槿 - 博客園
- data structures - Reversing a linked list in Java, recursively - Stack Overflow
- 反轉單向鏈表的四種實現（遞迴與非遞迴，C++） | 寧心勉學，慎思篤行
- iteratively and recursively Java Solution - Leetcode Discuss

# Merge Two Sorted Lists

## Question

- 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
```

## 題解

此題為兩個鏈表的合併，合併後的表頭節點不一定，故應聯想到使用 `dummy` 節點。鏈表節點的插入主要涉及節點 `next` 指標值的改變，兩個鏈表的合併操作則涉及到兩個節點的 `next` 值變化，若每次合併一個節點都要改變兩個節點 `next` 的值且要對 `NULL` 指標做異常處理，勢必會異常麻煩。嗯，第一次做這題時我就是這麼想的... 下面看看相對較好的思路。

首先 `dummy` 節點還是必須要用到，除了 `dummy` 節點外還引入一個 `lastNode` 節點充當下一次合併時的頭節點。在 `l1` 或者 `l2` 的某一個節點為空指標 `NULL` 時，退出 `while` 循環，並將非空鏈表的頭部鏈接到 `lastNode->next` 中。

### C++

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

```
                l2 = l2->next;
            }

            lastNode = lastNode->next;
        }

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

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

## 源碼分析

1. 異常處理，包含在 `dummy->next` 中。
2. 引入 `dummy` 和 `lastNode` 節點，此時 `lastNode` 指向的節點為 `dummy`
3. 對非空l1,l2循環處理，將l1/l2的較小者鏈接到 `lastNode->next` ，往後遞推 `lastNode`
4. 最後處理l1/l2中某一鏈表為空退出while循環，將非空鏈表頭鏈接到 `lastNode->next`
5. 返回 `dummy->next` ，即最終的首指標

注意 `lastNode` 的遞推並不影響 `dummy->next` 的值，因為 `lastNode` 和 `dummy` 是兩個不同的指標變量。

> 鏈表的合併為常用操作，務必非常熟練，以上的模板非常精煉，有兩個地方需要記牢。1. 循環結束條件中為條件與操作；2. 最後處理 `lastNode->next` 指標的值。

## 複雜度分析

最好情況下，一個鏈表為空，時間複雜度為 $O(1)$. 最壞情況下， `lastNode` 遍曆兩個鏈表中的每一個節點，時間複雜度為 $O(l1 + l2)$. 空間複雜度近似為 $O(1)$.

## Reference

- Merge Two Sorted Lists | 九章算法

# Remove Linked List Elements

## Question

- leetcode: Remove Linked List Elements | LeetCode OJ
- lintcode: (452) Remove Linked List Elements

### Problem Statement

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

#### Example

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

## 題解

刪除鏈表中指定值，找到其前一個節點即可，將 next 指向下一個節點即可。

## Python

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def removeElements(self, head, val):
        """
        :type head: ListNode
        :type val: int
        :rtype: ListNode
        """
        dummy = ListNode(0)
        dummy.next = head
        curr = dummy
        while curr.next is not None:
            if curr.next.val == val:
                curr.next = curr.next.next
            else:
                curr = curr.next

        return dummy.next
```

## Java

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

        return dummy.next;
    }
}
```

## 源碼分析

while 循環中使用 `curr.next` 較爲方便，if 語句中比較時也使用 `curr.next.val` 也比較簡潔，如果使用 `curr` 會比較難處理。

## 複雜度分析

略

Maximum Depth of Binary Tree# Binary Tree - 二元樹

二元樹的基本概念在 Binary Tree | Algorithm 中有簡要的介紹，這裏就二元樹的一些應用做一些實戰演練。

二元樹的遍歷大致可分為前序、中序、後序三種方法。

# Binary Tree Preorder Traversal

## Question

- leetcode: Binary Tree Preorder Traversal | LeetCode OJ
- lintcode: (66) Binary Tree Preorder Traversal

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

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

   1
    \
     2
    /
   3

return [1,2,3].

Example
Challenge
Can you do it without recursion?
```

## 題解1 - 遞迴

**面試時不推薦遞迴這種做法。**

遞迴版很好理解，首先判斷當前節點(根節點)是否為 `null` ，是則返回空vector，否則先返回當前節點的值，然後對當前節點的左節點遞迴，最後對當前節點的右節點遞迴。遞迴時對返回結果的處理方式不同可進一步細分為遍歷和分治兩種方法。

譯註：也不是完全不能這麼做，不過以二元樹的遍歷來說，遞迴方法太容易實現，面試官很可能進一步要求迭代的方法，並且有可能會問遞迴的缺點(連續呼叫函數導致stack的overflow問題)，不過如果遍歷並不是題幹而只是解決方法的步驟，用簡單的迭代方式實現有時亦無不可且可以減少錯誤，因此務必要和面試官充分溝通，另即使迭代寫不出來只寫出遞迴版本也要好過完全寫不出東西。

## Python - Divide and Conquer

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

```
"""


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

## C++ - Divide and Conquer

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root != NULL) {
            // Divide (分)
            vector<int> left = preorderTraversal(root->left);
            vector<int> right = preorderTraversal(root->right);
            // Merge
            result.push_back(root->val);
            result.insert(result.end(), left.begin(), left.end());
            result.insert(result.end(), right.begin(), right.end());
        }

        return result;
    }
```

```
};
```

## C++ - Traversal

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        traverse(root, result);

        return result;
    }

private:
    void traverse(TreeNode *root, vector<int> &ret) {
        if (root != NULL) {
            ret.push_back(root->val);
            traverse(root->left, ret);
            traverse(root->right, ret);
        }
    }
};
```

## Java - Divide and Conquer

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
```

```
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root != null) {
            // Divide
            List<Integer> left = preorderTraversal(root.left);
            List<Integer> right = preorderTraversal(root.right);
            // Merge
            result.add(root.val);
            result.addAll(left);
            result.addAll(right);
        }

        return result;
    }
}
```

## 源碼分析

使用遍歷的方法保存遞迴返回結果需要使用輔助遞迴函數 `traverse` ，將結果作為參數傳入遞迴函數中，傳值時注意應使用 `vector` 的引用。 分治方法首先分開計算各結果，最後合並到最終結果中。 C++ 中由於是使用vector, 將新的vector插入另一vector不能再使用push_back, 而應該使用insert。 Java 中使用 `addAll` 方法.

## 複雜度分析

遍歷樹中節點，時間複雜度 $O(n)$, 未使用額外空間(不包括呼叫函數的stack開銷)。

# 題解2 - 迭代

迭代時需要利用堆疊來保存遍歷到的節點，紙上畫圖分析後發現應首先進行出堆疊拋出當前節點，保存當前節點的值，隨後將右、左節點分別進入堆疊(注意進入堆疊順序，先右後左)，迭代到其為葉子節點(NULL)為止。

## Python

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```python
class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def preorderTraversal(self, root):
        if root is None:
            return []

        result = []
        s = []
        s.append(root)
        while s:
            root = s.pop()
            result.append(root.val)
            if root.right is not None:
                s.append(root.right)
            if root.left is not None:
                s.append(root.left)

        return result
```

## C++

```cpp
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root == NULL) return result;

        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
```

```
            TreeNode *node = s.top();
            s.pop();
            result.push_back(node->val);
            if (node->right != NULL) {
                s.push(node->right);
            }
            if (node->left != NULL) {
                s.push(node->left);
            }
        }

        return result;
    }
};
```

## Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root == null) return result;

        Stack<TreeNode> s = new Stack<TreeNode>();
        s.push(root);
        while (!s.empty()) {
            TreeNode node = s.pop();
            result.add(node.val);
            if (node.right != null) s.push(node.right);
            if (node.left != null) s.push(node.left);
        }

        return result;
    }
}
```

## 源碼分析

1. 對root進行異常處理
2. 將root壓入堆疊

3. 循環終止條件為堆疊s為空，所有元素均已處理完
4. 訪問當前堆疊頂元素(首先取出堆疊頂元素，隨後pop掉堆疊頂元素)並存入最終結果
5. 將右、左節點分別壓入堆疊內，以便取元素時為先左後右。
6. 返回最終結果

其中步驟4,5,6為迭代的核心，對應前序遍歷「根左右」。

所以說到底，**使用迭代，只不過是另外一種形式的遞迴。**使用遞迴的思想去理解遍歷問題會容易理解許多。

## 複雜度分析

使用輔助堆疊，最壞情況下堆疊空間與節點數相等，空間複雜度近似為 $O(n)$, 對每個節點遍歷一次，時間複雜度近似為 $O(n)$.

# Binary Tree Inorder Traversal

## Question

- leetcode: Binary Tree Inorder Traversal | LeetCode OJ
- lintcode: (67) Binary Tree Inorder Traversal

### Problem Statement

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?

## 題解1 - 遞迴版

中序遍歷的訪問順序爲『先左再根後右』，遞迴版最好理解，遞迴調用時注意返回值和遞迴左右子樹的順序即可。

## Python

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


class Solution:
```

```
    """
    @param root: The root of binary tree.
    @return: Inorder in ArrayList which contains node values.
    """
    def inorderTraversal(self, root):
        if root is None:
            return []
        else:
            return [root.val] + self.inorderTraversal(root.left) \
                              + self.inorderTraversal(root.right)
```

## Python - with helper

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def inorderTraversal(self, root):
        result = []
        self.helper(root, result)
        return result

    def helper(self, root, ret):
        if root is not None:
            self.helper(root.left, ret)
            ret.append(root.val)
            self.helper(root.right, ret)
```

## C++

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
```

```
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        helper(root, result);
        return result;
    }

private:
    void helper(TreeNode *root, vector<int> &ret) {
        if (root != NULL) {
            helper(root->left, ret);
            ret.push_back(root->val);
            helper(root->right, ret);
        }
    }
};
```

## Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        helper(root, result);
        return result;
    }

    private void helper(TreeNode root, List<Integer> ret) {
        if (root != null) {
            helper(root.left, ret);
            ret.add(root.val);
            helper(root.right, ret);
        }
    }
}
```

## 源碼分析

Python 這種動態語言在寫遞迴時返回結果好處理點，無需聲明類型。通用的方法爲在遞迴函數入口參數中傳入返回結果， 也可使用分治的方法替代輔助函數。

## 複雜度分析

樹中每個節點都需要被訪問常數次，時間複雜度近似爲 $O(n)$. 未使用額外輔助空間。

# 題解2 - 迭代版

使用輔助 stack 改寫遞迴程序，中序遍歷沒有前序遍歷好寫，其中之一就在於出入 stack 的順序和限制規則。我們採用「左根右」的訪問順序可知主要由如下四步構成。

1. 首先需要一直對左子樹迭代並將非空節點壓入 stack
2. 節點指針爲空後不再壓入 stack
3. 當前節點爲空時進行出 stack 操作，並訪問 stack 頂節點
4. 將當前指針p用其右子節點替代

步驟2,3,4對應「左根右」的遍歷結構，只是此時的步驟2取的左值爲空。

## Python

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def inorderTraversal(self, root):
        result = []
        s = []
        while root is not None or s:
            if root is not None:
                s.append(root)
                root = root.left
            else:
                root = s.pop()
                result.append(root.val)
                root = root.right

        return result
```

## C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
```

```
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Inorder in vector which contains node values.
     */
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        stack<TreeNode *> s;

        while (!s.empty() || NULL != root) {
            if (root != NULL) {
                s.push(root);
                root = root->left;
            } else {
                root = s.top();
                s.pop();
                result.push_back(root->val);
                root = root->right;
            }
        }

        return result;
    }
};
```

## Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
```

```
        if (root == null) return result;

        Deque<TreeNode> stack = new ArrayDeque<TreeNode>();
        while (root != null || (!stack.isEmpty())) {
            if (root != null) {
                stack.push(root);
                root = root.left;
            } else {
                root = stack.pop();
                result.add(root.val);
                root = root.right;
            }
        }

        return result;
    }
}
```

## 源碼分析

使用 stack 的思想模擬遞迴，注意迭代的演進和邊界條件即可。

## 複雜度分析

最壞情況下 stack 保存所有節點，空間複雜度 $O(n)$, 時間複雜度 $O(n)$.

# Reference

# Binary Tree Postorder Traversal

## Question

- leetcode: Binary Tree Postorder Traversal | LeetCode OJ
- lintcode: (68) Binary Tree Postorder Traversal

### Problem Statement

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?

## 題解1 - 遞迴

首先使用遞迴便於理解。

## Python - Divide and Conquer

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def postorderTraversal(self, root):
```

```
        if root is None:
            return []
        else:
            return self.postorderTraversal(root.left) +\
                   self.postorderTraversal(root.right) + [root.val]
```

## C++ - Traversal

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in vector which contains node values.
     */
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;

        traverse(root, result);

        return result;
    }

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

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

## Java - Divide and Conquer

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root != null) {
            List<Integer> left = postorderTraversal(root.left);
            result.addAll(left);
            List<Integer> right = postorderTraversal(root.right);
            result.addAll(right);
            result.add(root.val);
        }

        return result;
    }
}
```

## 源碼分析

遞迴版的太簡單了，沒啥好說的，注意入 Stack 順序。

## 複雜度分析

時間複雜度近似爲 $O(n)$.

# 題解2 - 迭代

使用遞迴寫後序遍歷那是相當的簡單，我們來個不使用遞迴的迭代版。整體思路仍然爲「左右根」，那麼怎麼才能知道什麼時候該訪問根節點呢？問題即轉化爲如何保證左右子節點一定先被訪問到？由於入Stack之後左右節點已無法區分，因此需要區分左右子節點是否被訪問過(加入到最終返回結果中)。除了有左右節點的情況，根節點也可能沒有任何子節點，此時也可直接將其值加入到最終返回結果中。

## Python

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
```

```
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def postorderTraversal(self, root):
        result = []
        if root is None:
            return result
        s = []
        # previously traversed node
        prev = None
        s.append(root)
        while s:
            curr = s[-1]
            noChild = curr.left is None and curr.right is None
            childVisited = (prev is not None) and \
                           (curr.left == prev or curr.right == prev)
            if noChild or childVisited:
                result.append(curr.val)
                s.pop()
                prev = curr
            else:
                if curr.right is not None:
                    s.append(curr.right)
                if curr.left is not None:
                    s.append(curr.left)

        return result
```

## C++

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> result;
        if (root == NULL) return result;

        TreeNode *prev = NULL;
```

```
        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *curr = s.top();
            bool noChild = false;
            if (curr->left == NULL && curr->right == NULL) {
                noChild = true;
            }
            bool childVisited = false;
            if (prev != NULL && (curr->left == prev || curr->right == prev)) {
                childVisited = true;
            }

            // traverse
            if (noChild || childVisited) {
                result.push_back(curr->val);
                s.pop();
                prev = curr;
            } else {
                if (curr->right != NULL) s.push(curr->right);
                if (curr->left != NULL) s.push(curr->left);
            }
        }

        return result;
    }
};
```

## Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root == null) return result;

        Deque<TreeNode> stack = new ArrayDeque<TreeNode>();
        stack.push(root);
        TreeNode prev = null;
        while (!stack.isEmpty()) {
            TreeNode curr = stack.peek();
```

```
            boolean noChild = (curr.left == null && curr.right == null);
            boolean childVisited = false;
            if (prev != null && (curr.left == prev || curr.right == prev)) {
                childVisited = true;
            }

            if (noChild || childVisited) {
                result.add(curr.val);
                stack.pop();
                prev = curr;
            } else {
                if (curr.right != null) stack.push(curr.right);
                if (curr.left != null) stack.push(curr.left);
            }
        }

        return result;
    }
}
```

## 源碼分析

遍歷順序爲『左右根』，判斷根節點是否應該從Stack中剔除有兩種條件，一爲無子節點，二爲子節點已遍歷過。判斷子節點是否遍歷過需要排除 `prev == null` 的情況，因爲 prev 初始化爲 null.

**將遞迴寫成迭代的難點在於如何在迭代中體現遞迴本質及邊界條件的確立，可使用簡單示例和紙上畫出Stack調用圖輔助分析。**

## 複雜度分析

最壞情況下Stack內存儲所有節點，空間複雜度近似爲 $O(n)$, 每個節點遍歷兩次或以上，時間複雜度近似爲 $O(n)$.

# 題解3 - 反轉先序遍歷

要想得到『左右根』的後序遍歷結果，我們發現只需將『根右左』的結果轉置即可，而先序遍歷通常爲『根左右』，故改變『左右』的順序即可，所以如此一來後序遍歷的非遞迴實現起來就非常簡單了。

## C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
```

```
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in vector which contains node values.
     */
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root == NULL) return result;

        stack<TreeNode*> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();
            result.push_back(node->val);
            // root, right, left => left, right, root
            if (node->left != NULL) s.push(node->left);
            if (node->right != NULL) s.push(node->right);
        }
        // reverse
        std::reverse(result.begin(), result.end());
        return result;
    }
};
```

## Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in ArrayList which contains node values.
```

```
     */
    public ArrayList<Integer> postorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) return result;

        Deque<TreeNode> stack = new ArrayDeque<TreeNode>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            result.add(node.val);
            if (node.left != null) stack.push(node.left);
            if (node.right != null) stack.push(node.right);
        }
        Collections.reverse(result);

        return result;
    }
}
```

## 源碼分析

注意入Stack的順序和最後轉置即可。

## 複雜度分析

同先序遍歷。

## Reference

- [leetcode]Binary Tree Postorder Traversal @ Python - 南郭子綦 - 解釋清晰
- 更簡單的非遞迴遍歷二叉樹的方法 - 比較新穎和簡潔的實現

# Binary Tree Level Order Traversal

## Question

- leetcode: Binary Tree Level Order Traversal | LeetCode OJ
- lintcode: (69) Binary Tree Level Order Traversal

### Problem Statement

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

Challenge 1: Using only 1 queue to implement it.

Challenge 2: Use DFS algorithm to do it.

## 題解 - 使用隊列

此題爲廣度優先搜索(BFS)的基礎題，使用一個隊列保存每層的節點即可。出隊列和將子節點入隊列的實現使用 for 循環，將每一輪的節點輸出。

### C++

```
/**
 * Definition of TreeNode:
```

```
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int> > result;

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

        queue<TreeNode *> q;
        q.push(root);
        while (!q.empty()) {
            vector<int> list;
            int size = q.size(); // keep the queue size first
            for (int i = 0; i != size; ++i) {
                TreeNode * node = q.front();
                q.pop();
                list.push_back(node->val);
                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }
            result.push_back(list);
        }

        return result;
    }
};
```

## Java

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        if (root == null) return result;

        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            List<Integer> list = new ArrayList<Integer>();
            int qSize = q.size();
            for (int i = 0; i < qSize; i++) {
                TreeNode node = q.poll();
                list.add(node.val);
                // push child node into queue
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            result.add(new ArrayList<Integer>(list));
        }

        return result;
    }
}
```

## 源碼分析

1. 異常，還是異常
2. 使用STL的 `queue` 數據結構，將 `root` 添加進隊列
3. **遍歷當前層所有節點，注意需要先保存隊列大小，因爲在入隊出隊時隊列大小會變化**
4. `list` 保存每層節點的值，每次使用均要初始化

## 複雜度分析

使用輔助隊列，空間複雜度 $O(n)$, 時間複雜度 $O(n)$.

# Binary Tree Level Order Traversal II

## Question

- leetcode: Binary Tree Level Order Traversal II | LeetCode OJ
- lintcode: (70) Binary Tree Level Order Traversal II

```
Given a binary tree, return the bottom-up level order traversal of its nodes' value
s.
(ie, from left to right, level by level from leaf to root).

Example
Given binary tree {3,9,20,#,#,15,7},

    3
   / \
  9  20
    /  \
   15   7


return its bottom-up level order traversal as:
[
  [15,7],
  [9,20],
  [3]
]
```

## 題解

這題在普通的 BFS 基礎上增加了逆序輸出，簡單的實現可以使用 Stack 或者最後對結果逆序。

## Java - Stack

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
```

```java
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: buttom-up level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null) return result;

        Stack<ArrayList<Integer>> s = new Stack<ArrayList<Integer>>();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            int qLen = q.size();
            ArrayList<Integer> aList = new ArrayList<Integer>();
            for (int i = 0; i < qLen; i++) {
                TreeNode node = q.poll();
                aList.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            s.push(aList);
        }

        while (!s.empty()) {
            result.add(s.pop());
        }
        return result;
    }
}
```

## Java - Reverse

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */


public class Solution {
```

```java
    /**
     * @param root: The root of binary tree.
     * @return: buttom-up level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null) return result;

        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while (!q.isEmpty()) {
            int qLen = q.size();
            ArrayList<Integer> aList = new ArrayList<Integer>();
            for (int i = 0; i < qLen; i++) {
                TreeNode node = q.poll();
                aList.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            result.add(aList);
        }

        Collections.reverse(result);
        return result;
    }
}
```

## 源碼分析

Java 中 Queue 是接口，通常可用 LinkedList 實例化。

## 複雜度分析

時間複雜度爲 $O(n)$, 使用了 Queue 或者 Stack 作爲輔助空間，空間複雜度爲 $O(n)$.

# Maximum Depth of Binary Tree

## Question

- leetcode: Maximum Depth of Binary Tree | LeetCode OJ
- lintcode: (97) Maximum Depth of Binary Tree

### Problem Statement

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 farthest leaf node.

#### Example

Given a binary tree as follow:

```
  1
 / \
2   3
   / \
  4   5
```

The maximum depth is `3`.

## 題解 - 遞迴

樹遍歷的題目最方便的寫法自然是遞迴，不過遞迴調用的層數過多可能會導致 Stack 空間 overflow，因此需要適當考慮遞迴調用的層數。我們首先來看看使用遞迴如何解這道題，要求二叉樹的最大深度，直觀上來講使用深度優先搜索判斷左右子樹的深度孰大孰小即可，從根節點往下一層樹的深度即自增1，遇到 `NULL` 時即返回0。

由於對每個節點都會使用一次 `maxDepth` ，故時間複雜度爲 $O(n)$, 樹的深度最大爲 $n$, 最小爲 $\log_2 n$, 故空間複雜度介於 $O(\log n)$ 和 $O(n)$ 之間。

### C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
```

```
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);

        return max(left_depth, right_depth) + 1;
    }
};
```

## Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        // write your code here
        if (root == null) {
            return 0;
        }
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
```

```
}
```

## 題解 - 迭代(顯式使用 Stack)

使用遞迴可能會導致棧空間溢出，這裏使用顯式棧空間(使用堆內存)來代替之前的隱式 Stack 空間。從上節遞迴版的程式碼(先處理左子樹，後處理右子樹，最後返回其中的較大值)來看，是可以使用類似後序遍歷的迭代思想去實現的。

首先使用後序遍歷的模板，在每次迭代循環結束處比較棧當前的大小和當前最大值 `max_depth` 進行比較。

### C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
            return 0;
        }

        TreeNode *curr = NULL, *prev = NULL;
        stack<TreeNode *> s;
        s.push(root);

        int max_depth = 0;

        while(!s.empty()) {
            curr = s.top();
            if (!prev || prev->left == curr || prev->right == curr) {
                if (curr->left) {
                    s.push(curr->left);
                } else if (curr->right){
```

```
                    s.push(curr->right);
                }
            } else if (curr->left == prev) {
                if (curr->right) {
                    s.push(curr->right);
                }
            } else {
                s.pop();
            }

            prev = curr;

            if (s.size() > max_depth) {
                max_depth = s.size();
            }
        }

        return max_depth;
    }
};
```

## 題解3 - 迭代(隊列)

在使用了遞迴/後序遍歷求解樹最大深度之後，我們還可以直接從問題出發進行分析，樹的最大深度即爲廣度優先搜索中的層數，故可以直接使用廣度優先搜索求出最大深度。

### C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: An integer
     */
    int maxDepth(TreeNode *root) {
        if (NULL == root) {
```

```
            return 0;
        }

        queue<TreeNode *> q;
        q.push(root);

        int max_depth = 0;
        while(!q.empty()) {
            int size = q.size();
            for (int i = 0; i != size; ++i) {
                TreeNode *node = q.front();
                q.pop();

                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }

            ++max_depth;
        }

        return max_depth;
    }
};
```

## Java

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
```

```
        }

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

        return depth;
    }
}
```

## 源碼分析

廣度優先中隊列的使用中，`qLen` 需要在for 循環遍歷之前獲得，因爲它是一個變量。

## 複雜度分析

最壞情況下空間複雜度爲 $O(n)$, 遍歷每一個節點，時間複雜度爲 $O(n)$,

# Invert Binary Tree

## Question

- leetcode: Invert Binary Tree | LeetCode OJ
- lintcode: (175) Invert Binary Tree

```
Invert a binary tree.

Example
  1         1
 / \       / \
2   3  => 3   2
   /       \
  4         4
Challenge
Do it in recursion is acceptable, can you do it without recursion?
```

## 題解1 - Recursive

二元樹的題用遞迴的思想求解自然是最容易的，此題要求爲交換左右子節點，故遞迴交換即可。具體實現可分返回值爲 `NULL` 或者二元樹節點兩種情況，返回值爲節點的情況理解起來相對不那麼直觀一些。

### C++ - return void

```cpp
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * };
 */
class Solution {
public:
    /**
     * @param root: a TreeNode, the root of the binary tree
     * @return: nothing
     */
    void invertBinaryTree(TreeNode *root) {
```

```
        if (root == NULL) return;
        swap(root->left, root->right);
        invertBinaryTree(root->left);
        invertBinaryTree(root->right);
    }
};
```

## C++ - return TreeNode *

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return NULL;

        TreeNode *temp = root->left;
        root->left = invertTree(root->right);
        root->right = invertTree(temp);

        return root;
    }
};
```

## 源碼分析

分三塊實現，首先是節點爲空的情況，然後交換左右節點，最後遞迴調用，遞迴調用的正確性可通過畫圖理解。

## 複雜度分析

每個節點遍歷一次，時間複雜度爲 $O(n)$, 使用了臨時變數，空間複雜度爲 $O(1)$.

# 題解2 - Iterative

遞迴的實現非常簡單，那麼非遞迴的如何實現呢？如果將遞迴改寫成 stack 的實現，那麼簡單來講就需要兩個 stack 了，稍顯複雜。其實仔細觀察此題可發現使用 level-order 的遍歷次序也可實現。即從根節點開始進入隊列 queue，交換左右節點，並將非空的左右子節點進入隊列，從隊列中取出節點，交換之，直至隊列爲空。

## C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * };
 */
class Solution {
public:
    /**
     * @param root: a TreeNode, the root of the binary tree
     * @return: nothing
     */
    void invertBinaryTree(TreeNode *root) {
        if (root == NULL) return;

        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            // pop out the front node
            TreeNode *node = q.front();
            q.pop();
            // swap between left and right pointer
            swap(node->left, node->right);
            // push non-NULL node
            if (node->left != NULL) q.push(node->left);
            if (node->right != NULL) q.push(node->right);
        }
    }
};
```

## 源碼分析

交換左右指針後需要判斷子節點是否非空，僅入隊非空子節點。

## 複雜度分析

遍歷每一個節點，時間複雜度爲 $O(n)$, 使用了隊列，最多存儲最下一層子節點數目，最多只有總節點數的一半，故最壞情況下 $O(n)$.

## Reference

- 0ms C++ Recursive/Iterative Solutions with Explanations - Leetcode Discuss

# Binary Search Tree - 二元搜尋樹

二元搜尋樹的定義及簡介在 Binary Search Trees 中已經有所介紹。簡單來說就是每個節點的值大於等於左子結點的值，而小於右子節點的值。

# Insert Node in a Binary Search Tree

## Question

- 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 keep the tree still be a valid binary search tree.

Example
Given binary search tree as follow:

     2

  /    \

1        4

         /

       3

after Insert node 6, the tree should be:

     2

  /    \

1        4

         /   \

       3        6

Challenge
Do it without recursion
```

## 題解 - 遞迴

二元樹的題使用遞迴自然是最好理解的，程式碼也簡潔易懂，缺點就是遞迴調用時stack空間容易溢出，故實際實現中一般使用迭代替代遞迴，性能更佳嘛。不過迭代的缺點就是程式碼量稍(很)大，邏輯也可能不是那麼好懂。

既然確定使用遞迴，那麼接下來就應該考慮具體的實現問題了。在遞迴的具體實現中，主要考慮如下兩點：

1. 基本條件/終止條件 - 返回值需斟酌。
2. 遞迴步/條件遞迴 - 能使原始問題收斂。

首先來找找遞迴步，根據二叉查找樹的定義，若插入節點的值若大於當前節點的值，則繼續與當前節點的右子樹的值進行比較；反之則繼續與當前節點的左子樹的值進行比較。題目的要求是返回最終二元搜尋樹的根節點，從以上遞迴步的描述中似乎還難以對應到實際程式碼，這時不妨分析下終止條件。

有了遞迴步，終止條件也就水到渠成了，若當前節點爲空時，即返回結果。問題是——返回什麼結果？當前節點爲空時，說明應該將「插入節點」插入到上一個遍歷節點的左子節點或右子節點。對應到程序程式碼中即爲 `root->right = node` 或者 `root->left = node` . 也就是說遞迴步使用 `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;
    }
}
```

## 題解 - 迭代

看過了以上遞迴版的題解，對於這個題來說，將遞迴轉化爲迭代的思路也是非常清晰易懂的。迭代比較當前節點的值和插入節點的值，到了二元樹的最後一層時選擇是鏈接至左子結點還是右子節點。

### 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;
    }
};
```

## 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;
    }
}
```

## 源碼分析

在 `NULL == tempNode->right` 或者 `NULL == tempNode->left` 時需要在鏈接完 `node` 後立即返回 `root` ，避免死循環。

# Exhaustive Search - 窮竭搜索

窮竭搜索又稱暴力搜索，指代將所有可能性列出來，然後再在其中尋找滿足題目條件的解。常用求解方法和工具有：

1. 遞歸函數
2. 棧
3. 隊列
4. 深度優先搜索(DFS, Depth-First Search)，又常稱為回溯法
5. 廣度優先搜索(BFS, Breadth-First Search)

1, 2, 3 往往在深搜或者廣搜中體現。

## DFS

DFS 通常從某個狀態開始，根據特定的規則轉移狀態，直至無法轉移(節點為空)，然後回退到之前一步狀態，繼續按照指定規則轉移狀態，直至遍曆完所有狀態。

回溯法包含了多類問題，模板類似。

排列組合模板->搜索問題(是否要排序，哪些情況要跳過)

使用回溯法的一般步驟：

1. 確定所給問題的解空間：首先應明確定義問題的解空間，解空間中至少包含問題的一個解。
2. 確定結點的擴展搜索規則
3. 以深度優先方式搜索解空間，並在搜索過程中用剪枝函數避免無效搜索。

## BFS

BFS 從某個狀態開始，搜索**所有可以到達的狀態**，轉移順序為『初始狀態->只需一次轉移就可到達的所有狀態->只需兩次轉移就可到達的所有狀態->...』，所以對於同一個狀態，BFS 只搜索一次，故時間複雜度為 $O(states \times transfer_methods)$. BFS 通常配合隊列一起使用，搜索時先將狀態加入到隊列中，然後從隊列頂端不斷取出狀態，再把從該狀態可轉移到的狀態中尚未訪問過的部分加入隊列，知道隊列為空或已找到解。因此 BFS 適合用於『由近及遠』的搜索，比較適合用於求解最短路徑、最少操作之類的問題。

## Reference


- 《挑戰程序設計競賽》Chaper 2.1 p26 最基礎的「窮竭搜索」
- Steven Skiena: Lecture15 - Backtracking
- 全面解析回溯法：算法框架與問題求解 - 五嶽 - 博客園
- 五大常用算法之四：回溯法 - 紅臉書生 - 博客園
- 演算法筆記 - Backtracking

# Subsets - 子集

## Question

- leetcode: Subsets | LeetCode OJ
- lintcode: (17) Subsets

```
Given a set of distinct integers, return all possible subsets.

Note
Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

Example
If S = [1,2,3], a solution is:

[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

## 題解

子集類問題類似Combination，以輸入陣列 `[1, 2, 3]` 分析，根據題意，最終返回結果中子集類的元素應該按照升序排列，故首先需要對原陣列進行排序。題目的第二點要求是子集不能重複，至此原題即轉化為數學中的組合問題。我們首先嘗試使用 DFS 進行求解，大致步驟如下：

1. `[1] -> [1, 2] -> [1, 2, 3]`
2. `[2] -> [2, 3]`
3. `[3]`

將上述過程轉化為程式碼即為對陣列遍歷，每一輪都保存之前的結果並將其依次加入到最終返回結果中。

## Python

```python
class Solution:
    # @param {integer[]} nums
```

```
    # @return {integer[][]}
    def subsets(self, nums):
        if nums is None:
            return []

        result = []
        nums.sort()
        self.dfs(nums, 0, [], result)
        return result

    def dfs(self, nums, pos, list_temp, ret):
        # append new object with []
        ret.append([] + list_temp)

        for i in xrange(pos, len(nums)):
            list_temp.append(nums[i])
            self.dfs(nums, i + 1, list_temp, ret)
            list_temp.pop()
```

## C++

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int> > result;
        if (nums.empty()) return result;

        sort(nums.begin(), nums.end());
        vector<int> list;
        dfs(nums, 0, list, result);

        return result;
    }

private:
    void dfs(vector<int>& nums, int pos, vector<int> &list,
             vector<vector<int> > &ret) {

        ret.push_back(list);

        for (int i = pos; i < nums.size(); ++i) {
            list.push_back(nums[i]);
            dfs(nums, i + 1, list, ret);
            list.pop_back();
        }
    }
};
```

## Java

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
        if (nums == null || nums.length == 0) {
            return result;
        }

        Arrays.sort(nums);
        dfs(nums, 0, list, result);

        return result;
    }

    private void dfs(int[] nums, int pos, List<Integer> list,
                     List<List<Integer>> ret) {

        // add temp result first
        ret.add(new ArrayList<Integer>(list));

        for (int i = pos; i < nums.length; i++) {
            list.add(nums[i]);
            dfs(nums, i + 1, list, ret);
            list.remove(list.size() - 1);
        }
    }
}
```

## 源碼分析

Java 和 Python 的程式碼中在將臨時list 添加到最終結果時新生成了物件，(Python 使用 `[] +` ), 否則最終返回結果將隨著 `list` 的變化而變化。

**Notice: backTrack(num, i + 1, list, ret);中的『i + 1』不可誤寫為『pos + 1』，因為 `pos` 用於每次大的循環， `i` 用於內循環，第一次寫subsets的時候在這坑了很久... :(**

回溯法可用圖示和函數運行的堆棧圖來理解，強烈建議**使用圖形和遞迴的思想**分析，以陣列 `[1, 2, 3]` 進行分析。下圖所示為 `list` 及 `result` 動態變化的過程，箭頭向下表示 `list.add` 及 `result.add` 操作，箭頭向上表示 `list.remove` 操作。

## 複雜度分析

對原有陣列排序，時間複雜度近似為 $O(n \log n)$. 狀態數為所有可能的組合數 $O(2^n)$, 生成每個狀態所需的時間複雜度近似為 $O(1)$, 如 `[1] -> [1, 2]` , 故總的時間複雜度近似為 $O(2^n)$.

使用了臨時空間 `list` 保存中間結果， `list` 最大長度為陣列長度，故空間複雜度近似為 $O(n)$.

## Reference

- [NineChap 1.2] Permutation - Woodstock Blog
- 九章算法 - subsets模板
- LeetCode: Subsets 解題報告 - Yu's Garden - 博客園

# Next Permutation

## Question

- leetcode: Next Permutation | LeetCode OJ
- lintcode: (52) Next Permutation

### Problem Statement

Given a list of integers, which denote a permutation.

Find the next permutation in ascending order.

#### Example

For `[1,3,2,3]`, the next permutation is `[1,3,3,2]`

For `[4,3,2,1]`, the next permutation is `[1,2,3,4]`

#### Note

The list may contains duplicate integers.

## 題解

找下一個升序排列，C++ STL 源碼剖析一書中有提及，Permutations 一小節中也有詳細介紹，下面簡要介紹一下字典序算法：

1. 從後往前尋找索引滿足 `a[k] < a[k + 1]`，如果此條件不滿足，則說明已遍歷到最後一個。
2. 從後往前遍歷，找到第一個比 `a[k]` 大的數 `a[l]`，即 `a[k] < a[l]`.
3. 交換 `a[k]` 與 `a[l]`.
4. 反轉 `k + 1 ~ n` 之間的元素。

由於這道題中規定對於 `[4,3,2,1]`，輸出爲 `[1,2,3,4]`，故在第一步稍加處理即可。

## Python

```
class Solution:
    # @param num :  a list of integer
    # @return : a list of integer
    def nextPermutation(self, num):
        if num is None or len(num) <= 1:
            return num
        # step1: find nums[i] < nums[i + 1], Loop backwards
        i = 0
        for i in xrange(len(num) - 2, -1, -1):
```

```
            if num[i] < num[i + 1]:
                break
            elif i == 0:
                # reverse nums if reach maximum
                num = num[::-1]
                return num
        # step2: find nums[i] < nums[j], Loop backwards
        j = 0
        for j in xrange(len(num) - 1, i, -1):
            if num[i] < num[j]:
                break
        # step3: swap betwenn nums[i] and nums[j]
        num[i], num[j] = num[j], num[i]
        # step4: reverse between [i + 1, n - 1]
        num[i + 1:len(num)] = num[len(num) - 1:i:-1]

        return num
```

## C++

```
class Solution {
public:
    /**
     * @param nums: An array of integers
     * @return: An array of integers that's next permuation
     */
    vector<int> nextPermutation(vector<int> &nums) {
        if (nums.empty() || nums.size() <= 1) {
            return nums;
        }
        // step1: find nums[i] < nums[i + 1]
        int i = 0;
        for (i = nums.size() - 2; i >= 0; --i) {
            if (nums[i] < nums[i + 1]) {
                break;
            } else if (0 == i) {
                // reverse nums if reach maximum
                reverse(nums, 0, nums.size() - 1);
                return nums;
            }
        }
        // step2: find nums[i] < nums[j]
        int j = 0;
        for (j = nums.size() - 1; j > i; --j) {
            if (nums[i] < nums[j]) break;
        }
        // step3: swap betwenn nums[i] and nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
```

```
        nums[j] = temp;
        // step4: reverse between [i + 1, n - 1]
        reverse(nums, i + 1, nums.size() - 1);

        return nums;

    }

private:
    void reverse(vector<int>& nums, int start, int end) {
        for (int i = start, j = end; i < j; ++i, --j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
};
```

## Java

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: return nothing (void), do not return anything, modify nums in-place
 instead
     */
    public void nextPermutation(int[] nums) {
        if (nums == null || nums.length == 0) return;

        // step1: search the first nums[k] < nums[k+1] backward
        int k = -1;
        for (int i = nums.length - 2; i >= 0; i--) {
            if (nums[i] < nums[i + 1]) {
                k = i;
                break;
            }
        }
        // if current rank is the largest, reverse it to smallest, return
        if (k == -1) {
            reverse(nums, 0, nums.length - 1);
            return;
        }

        // step2: search the first nums[k] < nums[l] backward
        int l = nums.length - 1;
        while (l > k && nums[l] <= nums[k]) l--;

        // step3: swap nums[k] with nums[l]
        int temp = nums[k];
```

```
        nums[k] = nums[l];
        nums[l] = temp;

        // step4: reverse between k+1 and nums.length-1;
        reverse(nums, k + 1, nums.length - 1);
    }

    private void reverse(int[] nums, int lb, int ub) {
        for (int i = lb, j = ub; i < j; i++, j--) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
}
```

## 源碼分析

和 Permutation 一小節類似，這裏只需要注意在step 1中 `i == -1` 時需要反轉之以獲得最小的序列。對於有重復元素，只要在 step1和 step2中判斷元素大小時不取等號即可。Lintcode 上給的註釋要求（其實是 Leetcode 上的要求）和實際給出的輸出不一樣。

## 複雜度分析

最壞情況下，遍歷兩次原陣列，反轉一次陣列，時間複雜度爲 $O(n)$, 使用了 temp 臨時變量，空間複雜度可認爲是 $O(1)$.

# Minimum Depth of Binary Tree

## Question

- leetcode: Minimum Depth of Binary Tree | LeetCode OJ
- lintcode: (155) Minimum Depth of Binary Tree

```
Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path
from the root node down to the nearest leaf node.

Example
Given a binary tree as follow:

        1

     /     \

   2       3

          /    \

        4      5
The minimum depth is 2
```

## 題解

注意審題，題中的最小深度指的是從根節點到**最近的葉子節點（因為題中的最小深度是the number of nodes，故該葉子節點不能是空節點）**，所以需要單獨處理葉子節點為空的情況。此題使用 DFS 遞迴實現比較簡單。

### Java

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
```

```java
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int minDepth(TreeNode root) {
        if (root == null) return 0;

        int leftDepth = minDepth(root.left);
        int rightDepth = minDepth(root.right);

        // current node is not leaf node
        if (root.left == null) {
            return 1 + rightDepth;
        } else if (root.right == null) {
            return 1 + leftDepth;
        }

        return 1 + Math.min(leftDepth, rightDepth);
    }
}
```

## C++

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int depth(TreeNode* n){
        if(!n->left and !n->right) return 1;
        if(!n->left) return 1 + depth(n->right);
        if(!n->right) return 1 + depth(n->left);
        return 1 + min(depth(n->left), depth(n->right));
    }
    int minDepth(TreeNode* root) {
        if(!root) return 0;
        return depth(root);
    }
};
```

## 源碼分析

建立好遞迴模型即可，左右子節點為空時需要單獨處理。

## 複雜度分析

每個節點遍歷一次，時間複雜度 $O(n)$. 不計函數呼叫堆疊空間的話空間複雜度 $O(1)$.

# Dynamic Programming - 動態規劃

動態規劃是一種「分治」的思想，通俗一點來說就是「大事化小，小事化無」的藝術。在將大問題化解爲小問題的「分治」過程中，保存對這些小問題已經處理好的結果，並供後面處理更大規模的問題時直接使用這些結果。嗯，感覺講了和沒講一樣，還是不會使用動規的思想解題...

下面看看知乎上的熊大大對動規比較「正經」的描述。

> 動態規劃是通過拆分問題，定義問題狀態和狀態之間的關係，使得問題能夠以遞推（或者說分治）的方式去解決。

以上定義言簡意賅，可直接用於實戰指導，不愧是參加過NOI的。

動規的思想雖然好理解，但是要真正活用起來就需要下點功夫了。建議看看下面知乎上的回答。

動態規劃最重要的兩個要點：

1. 狀態(狀態不太好找，可先從轉化方程入手分析)
2. 狀態間的轉化方程(從題目的隱含條件出發尋找遞推關係)

其他的要點則是如初始化狀態的確定(由狀態和轉化方程得知)，需要的結果(狀態轉移的終點)

動態規劃問題中一般從以下四個角度考慮：

1. 狀態(State)
2. 狀態間的轉移方程(Function)
3. 狀態的初始化(Initialization)
4. 返回結果(Answer)

動規適用的情形：

1. 最大值/最小值
2. 有無可行解
3. 求方案個數(如果需要列出所有方案，則一定不是動規，因爲全部方案爲指數級別複雜度，所有方案需要列出時往往用遞歸)
4. 給出的數據不可隨便調整位置

## 單序列(DP_Sequence)

單序列動態規劃的狀態通常定義爲：陣列前 i 個位置, 數字, 字母 或者 以第i個爲... 返回結果通常爲陣列的最後一個元素。

按照動態規劃的四要素，此類題可從以下四個角度分析。

1. State: f[i] 前i個位置/數字/字母...
2. Function: f[i] = f[i-1]... 找遞推關係
3. Initialization: 根據題意進行必要的初始化
4. Answer: f[n-1]

# 雙序列(DP_Two_Sequence)

一般有兩個陣列或者兩個字符串，計算其匹配關係。雙序列中常用二維陣列表示狀態轉移關係，但往往可以使用滾動陣列的方式對空間複雜度進行優化。舉個例子，以題 Distinct Subsequences 爲例，狀態轉移方程如下：

```
f[i+1][j+1] = f[i][j+1] + f[i][j] (if S[i] == T[j])
f[i+1][j+1] = f[i][j+1] (if S[i] != T[j])
```

從以上轉移方程可以看出 `f[i+1][*]` 只與其前一個狀態 `f[i][*]` 有關，而對於 `f[*][j]` 來說則基於當前索引又與前一個索引有關，故我們以遞推的方式省略第一維的空間，並以第一維作爲外循環，內循環爲 j, 由遞推關係可知在使用滾動陣列時，若內循環 j 仍然從小到大遍歷，那麼對於 `f[j+1]` 來說它得到的 `f[j]` 則是當前一輪( `f[i+1][j]` )的值，並不是需要的 `f[i][j]` 的值。所以若想得到上一輪的結果，必須在內循環使用逆推的方式進行。文字表述比較模糊，可以自行畫一個二維矩陣的轉移矩陣來分析，認識到這一點非常重要。

小結一下，使用滾動陣列的核心在於：

1. 狀態轉移矩陣中只能取 `f[i+1][*]` 和 `f[i][*]` , 這是使用滾動陣列的前提。
2. 外循環使用 i, 內循環使用 j 並同時使用逆推，這是滾動陣列使用的具體實踐。

程式碼如下：

```java
public class Solution {
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        if (S == null || T == null) return 0;
        if (S.length() < T.length()) return 0;
        if (T.length() == 0) return 1;

        int[] f = new int[T.length() + 1];
        f[0] = 1;
        for (int i = 0; i < S.length(); i++) {
            for (int j = T.length() - 1; j >= 0; j--) {
                if (S.charAt(i) == T.charAt(j)) {
                        f[j + 1] += f[j];
                }
            }
        }

        return f[T.length()];
    }
}
```

紙上得來終覺淺，絕知此事要躬行。光說不練假把戲，下面就來幾道DP的題練練手。

## Reference

1. 什麼是動態規劃？動態規劃的意義是什麼？ - 知乎 - 熊大大和王勐的回答值得細看，適合作爲動態規劃的科普和入門。維基百科上對動態規劃的描述感覺太過學術。
2. 動態規劃：從新手到專家 - Topcoder上的一篇譯作。

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

## Question

- lintcode: (109) Triangle

```
Given a triangle, find the minimum path sum from top to bottom. Each step you may m
ove to adjacent numbers on the row below.

Note
Bonus point if you are able to do this using only O(n) extra space, where n is the
total number of rows in the triangle.

Example
For example, given the following triangle

[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).
```

## 題解

題中要求最短路徑和，每次只能訪問下行的相鄰元素，將triangle視爲二維座標。此題方法較多，下面分小節詳述。

### Method 1 - Traverse without hashmap

首先考慮最容易想到的方法——遞歸遍歷，逐個累加所有自上而下的路徑長度，最後返回這些不同的路徑長度的最小值。由於每個點往下都有2條路徑，使用此方法的時間複雜度約爲 $O(2^n)$, 顯然是不可接受的解，不過我們還是先看看其實現思路。

### 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);
    }
};
```

## 源碼分析

`dfs()` 的循環終止條件爲 `x == n` ，而不是 `x == n - 1` ，主要是方便在遞歸時sum均可使用 `sum + triangle[x][y]` ，而不必根據不同的y和y+1改變，代碼實現相對優雅一些。理解方式則變爲從第x行走到第x+1行時的最短路徑和，也就是說在此之前並不將第x行的元素值計算在內。

這種遍歷的方法時間複雜度如此之高的主要原因是因爲在n較大時遞歸計算了之前已經得到的結果，而這些結果計算一次後即不再變化，可再次利用。因此我們可以使用hashmap記憶已經計算得到的結果從而對其進行優化。

## Method 2 - Divide and Conquer without hashmap

既然可以使用遞歸遍歷，當然也可以使用「分治」的方法來解。「分治」與之前的遍歷區別在於「分治」需要返回每次「分治」後的計算結果，下面看代碼實現。

## 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))  + triangl
e[x][y];
    }
};
```

使用「分治」的方法代碼相對簡潔一點，接下來我們使用hashmap保存triangle中不同座標的點計算得到的路徑和。

## Method 3 - Divide and Conquer with hashmap

新建一份大小和triangle一樣大小的hashmap，並對每個元素賦以 `INT_MIN` 以做標記區分。

## 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> > &has
hmap) {
        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];
    }
};
```

由於已經計算出的最短路徑值不再重複計算，計算複雜度由之前的 $O(2^n)$，變爲 $O(n^2)$, 每個座標的元素僅計算一次，故共計算的次數爲 $1+2+...+n \approx O(n^2)$.

## Method 4 - Dynamic Programming

從主章節中對動態規劃的簡介我們可以知道使用動態規劃的難點和核心在於**狀態的定義及轉化方程的建立**。那麼問題來了，到底如何去找適合這個問題的狀態及轉化方程呢？

我們仔細分析題中可能的狀態和轉化關係，發現從 `triangle` 中座標爲 $triangle[x][y]$ 的元素出發，其路徑只可能爲 $triangle[x][y]->triangle[x+1][y]$ 或者 $triangle[x][y]->triangle[x+1][y+1]$. 以點 $(x,y)$ 作爲參考，那麼可能的狀態 $f(x,y)$ 就可以是：

1. 從 $(x,y)$ 出發走到最後一行的最短路徑和
2. 從 $(0,0)$ 走到 $(x,y)$的最短路徑和

如果選擇1作爲狀態，則相應的狀態轉移方程爲：

$f_1(x,y) = min\{f_1(x+1,y), f_1(x+1,y+1)\} + triangle[x][y]$

如果選擇2作爲狀態，則相應的狀態轉移方程爲：

$f_2(x,y) = min\{f_2(x-1,y), f_2(x-1,y-1)\} + triangle[x][y]$

兩個狀態所對應的初始狀態分別爲 $f_1(n-1,y), 0 \le y \le n-1$ 和 $f_2(0,0)$. 在代碼中應注意考慮邊界條件。下面分別就這種不同的狀態進行動態規劃。

## 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]) + tri
angle[i][j];
            }
        }

        return hashmap[0][0];
    }
};
```

## 源碼分析

1. 異常處理
2. 使用hashmap保存結果
3. 初始化 `hashmap[N-1][i]` ,由於是自底向上，故初始化時保存最後一行元素
4. 使用自底向上的方式處理循環
5. 最後返回結果hashmap[0][0]

從空間利用角度考慮也可直接使用triangle替代hashmap，但是此舉會改變triangle的值，不推薦。

## 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;
    }
};
```

### 源碼解析

自頂向下的實現略微有點複雜，在尋路時需要考慮最左邊和最右邊的邊界，還需要在最後返回結果時比較最小值。

## Java From Top to Bottom

```
public class Solution {
    /**
```

```
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    public int minimumTotal(int[][] triangle) {
        // write your code here
        if (triangle == null || triangle.length == 0) return 0;
        int[] last = new int[triangle.length];
        int[] current = new int[triangle.length];
        last[0] = triangle[0][0];
        current[0] = last[0];
        for (int i = 1; i < triangle.length; i++) {
            for (int j = 0; j < i + 1; j++) {
                int sum = Integer.MAX_VALUE;
                if (j != 0) {
                    sum = triangle[i][j] + last[j - 1];
                }
                if (j != i) {
                    sum = Math.min(sum, triangle[i][j] + last[j]);
                }
                current[j] = sum;
            }
            for (int k = 0; k < i + 1; k++) last[k] = current[k];
        }
        int min = Integer.MAX_VALUE;
        for (int n : current) {
            min = Math.min(n, min);
        }
        return min;
    }
}
```

## 源碼解析

思路基本和上個解法一樣，但是在數組last中保留上一層的最短和的，因此不用hashmap，空間複雜度是O(n)

# Backpack

## Question

- lintcode: (92) Backpack

### Problem Statement

Given $n$ items with size $A_i$, an integer $m$ denotes the size of a backpack. How full you can fill this backpack?

#### Example

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

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

#### Note

You can not divide any item into small pieces.

#### Challenge

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

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

## 題解1

本題是典型的01揹包問題，每種類型的物品最多只能選擇一件。參考前文 Knapsack 中總結的解法，這個題中可以將揹包的 size 理解爲傳統揹包中的重量；題目問的是能達到的最大 size, 故可將每個揹包的 size 類比爲傳統揹包中的價值。

考慮到數組索引從0開始，故定義狀態 `bp[i + 1][j]` 爲前 `i` 個物品中選出重量不超過 `j` 時總價值的最大值。狀態轉移方程則爲分 `A[i] > j` 與否兩種情況考慮。初始化均爲0，相當於沒有放任何物品。

## Java

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
```

```
     */
    public int backPack(int m, int[] A) {
        if (A == null || A.length == 0) return 0;

        final int M = m;
        final int N = A.length;
        int[][] bp = new int[N + 1][M + 1];

        for (int i = 0; i < N; i++) {
            for (int j = 0; j <= M; j++) {
                if (A[i] > j) {
                    bp[i + 1][j] = bp[i][j];
                } else {
                    bp[i + 1][j] = Math.max(bp[i][j], bp[i][j - A[i]] + A[i]);
                }
            }
        }

        return bp[N][M];
    }
}
```

## 源碼分析

注意索引及初始化的值，尤其是 N 和 M 的區別，內循環處可等於 M。

## 複雜度分析

兩重 for 循環，時間複雜度爲 $O(m \times n)$, 二維矩陣的空間複雜度爲 $O(m \times n)$, 一維矩陣的空間複雜度爲 $O(m)$.

## 題解2

接下來看看 九章算法 的題解，**這種解法感覺不是很直觀，推薦使用題解1的解法。**

1. 狀態: result[i][S] 表示前i個物品，取出一些物品能否組成體積和爲S的揹包
2. 狀態轉移方程: $f[i][S] = f[i-1][S-A[i]]\ or\ f[i-1][S]$ (A[i]爲第i個物品的大小)
   - 欲從前i個物品中取出一些組成體積和爲S的揹包，可從兩個狀態轉換得到。
     i. $f[i-1][S-A[i]]$: **放入第i個物品**，前 $i-1$ 個物品能否取出一些體積和爲 $S-A[i]$ 的揹包。
     ii. $f[i-1][S]$: **不放入第i個物品**，前 $i-1$ 個物品能否取出一些組成體積和爲S的揹包。
3. 狀態初始化: $f[1 \cdots n][0] = true;\ f[0][1 \cdots m] = false.$ 前1~n個物品組成體積和爲0的揹包始終爲真，其他情況爲假。
4. 返回結果: 尋找使 $f[n][S]$ 值爲true的最大S ($1 \le S \le m$)

## C++ - 2D vector

```
class Solution {
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    int backPack(int m, vector<int> A) {
        if (A.empty() || m < 1) {
            return 0;
        }

        const int N = A.size() + 1;
        const int M = m + 1;
        vector<vector<bool> > result;
        result.resize(N);
        for (vector<int>::size_type i = 0; i != N; ++i) {
            result[i].resize(M);
            std::fill(result[i].begin(), result[i].end(), false);
        }

        result[0][0] = true;
        for (int i = 1; i != N; ++i) {
            for (int j = 0; j != M; ++j) {
                if (j < A[i - 1]) {
                    result[i][j] = result[i - 1][j];
                } else {
                    result[i][j] = result[i - 1][j] || result[i - 1][j - A[i - 1]];
                }
            }
        }

        // return the largest i if true
        for (int i = M; i > 0; --i) {
            if (result[N - 1][i - 1]) {
                return (i - 1);
            }
        }
        return 0;
    }
};
```

## 源碼分析

1. 異常處理
2. 初始化結果矩陣，注意這裏需要使用 `resize` 而不是 `reserve` ，否則可能會出現段錯誤
3. 實現狀態轉移邏輯，一定要分 `j < A[i - 1]` 與否來討論
4. 返回結果，只需要比較 `result[N - 1][i - 1]` 的結果，返回true的最大值

狀態轉移邏輯中代碼可以進一步簡化，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;
                }
            }
        }
```

考慮揹包問題的核心——狀態轉移方程，如何優化此轉移方程？原始方案中用到了二維矩陣來保存result，注意到result的第i行僅依賴於第i-1行的結果，那麼能否用一維數組來代替這種隱含的關係呢？我們**在內循環j處遞減即可**。如此即可避免 `result[i][S]` 的值由本輪 `result[i][S-A[i]]` 遞推得到。

## C++ - 1D vector

```
class Solution {
public:
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    int backPack(int m, vector<int> A) {
        if (A.empty() || m < 1) {
            return 0;
        }

        const int N = A.size();
        vector<bool> result;
        result.resize(m + 1);
        std::fill(result.begin(), result.end(), false);

        result[0] = true;
        for (int i = 0; i != N; ++i) {
            for (int j = m; j >= 0; --j) {
                if (j >= A[i] && result[j - A[i]]) {
                    result[j] = true;
                }
            }
        }

        // return the largest i if true
        for (int i = m; i > 0; --i) {
            if (result[i]) {
                return i;
            }
```

```
        }
        return 0;
    }
};
```

## 複雜度分析

兩重 for 循環，時間複雜度均爲 $O(m \times n)$, 二維矩陣的空間複雜度爲 $O(m \times n)$, 一維矩陣的空間複雜度爲 $O(m)$.

## Reference

- 《挑戰程序設計競賽》第二章
- Lintcode: Backpack - neverlandly - 博客園
- 九章算法 | 揹包問題
- 崔添翼 § 翼若垂天之雲 › 《揹包問題九講》2.0 alpha1

# Climbing Stairs

## Question

- 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
```

## 題解

題目問的是到達頂端的方法數，我們採用序列類問題的通用分析方法，可以得到如下四要素：

1. State: f[i] 爬到第i級的方法數
2. Function: f[i]=f[i-1]+f[i-2]
3. Initialization: f[0]=1,f[1]=1
4. Answer: f[n]

尤其注意狀態轉移方程的寫法，f[i]只可能由兩個中間狀態轉化而來，一個是f[i-1]，由f[i-1]到f[i]其方法總數並未增加；另一個是f[i-2]，由f[i-2]到f[i]隔了兩個臺階，因此有1+1和2兩個方法，因此容易寫成f[i]=f[i-1]+f[i-2]+1，但仔細分析後能發現，由f[i-2]到f[i]的中間狀態f[i-1]已經被利用過一次，故f[i]=f[i-1]+f[i-2]. 使用動規思想解題時需要分清『重疊子狀態』, 如果有重複的需要去掉。

## C++

```cpp
class Solution {
public:
    /**
     * @param n: An integer
     * @return: An integer
     */
    int climbStairs(int n) {
        if (n < 1) {
            return 0;
        }

        vector<int> ret(n + 1, 1);
```

```
        for (int i = 2; i != n + 1; ++i) {
            ret[i] = ret[i - 1] + ret[i - 2];
        }

        return ret[n];
    }
};
```

1. 異常處理
2. 初始化n+1個元素，初始值均爲1。之所以用n+1個元素是下標分析起來更方便
3. 狀態轉移方程
4. 返回ret[n]

初始化ret[0]也爲1，可以認爲到第0級也是一種方法。

以上答案的空間複雜度爲 $O(n)$，仔細觀察後可以發現在狀態轉移方程中，我們可以使用三個變數來替代長度爲n+1的數組。具體程式碼可參考 climbing-stairs | 九章算法

## Python

```
class Solution:
    def climbStairs(n):
        if n < 1:
            return 0

        l = r = 1
        for _ in xrange(n - 1):
            l, r = r, r + l
        return r
```

## 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;
    }
};
```

# Graph

本章主要總結圖與搜索相關題目。

# Data Structure

本章主要總結如 Queue, Stack 等資料結構相關的題目。

# Problem Misc

本章主要總結暫時不方便歸到其他章節的題目。

# String to Integer

## Question

- leetcode: String to Integer (atoi) | LeetCode OJ
- lintcode: (54) String to Integer(atoi)

```
Implement function atoi to convert a string to an integer.

If no valid conversion could be performed, a zero value is returned.

If the correct value is out of the range of representable values,
INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

Example
"10" => 10

"-1" => -1

"123123123123123" => 2147483647

"1.0" => 1
```

## 題解

經典的字符串轉整數題，邊界條件比較多，比如是否需要考慮小數點，空白及非法字符的處理，正負號的處理，科學計數法等。最先處理的是空白字符，然後是正負號，接下來只要出現非法字符(包含正負號，小數點等，無需對這兩類單獨處理)即退出，否則按照正負號的整數進位加法處理。

### Java

```java
public class Solution {
    /**
     * @param str: A string
     * @return An integer
     */
    public int atoi(String str) {
        if (str == null || str.length() == 0) return 0;

        // trim left and right spaces
        String strTrim = str.trim();
        int len = strTrim.length();
        // sign symbol for positive and negative
        int sign = 1;
        // index for iteration
```

```
        int i = 0;
        if (strTrim.charAt(i) == '+') {
            i++;
        } else if (strTrim.charAt(i) == '-') {
            sign = -1;
            i++;
        }

        // store the result as long to avoid overflow
        long result = 0;
        while (i < len) {
            if (strTrim.charAt(i) < '0' || strTrim.charAt(i) > '9') {
                break;
            }
            result = 10 * result + sign * (strTrim.charAt(i) - '0');
            // overflow
            if (result > Integer.MAX_VALUE) {
                return Integer.MAX_VALUE;
            } else if (result < Integer.MIN_VALUE) {
                return Integer.MIN_VALUE;
            }
            i++;
        }

        return (int)result;
    }
}
```

## 源碼分析

符號位使用整數型表示，便於後期相乘相加。在 while 循環中需要注意判斷是否已經溢位，如果放在 while 循環外面則有可能超過 long 型範圍。

## C++

```
class Solution {
public:
    bool overflow(string str, string help){
        if(str.size() > help.size()) return true;
        else if(str.size() < help.size()) return false;
        for(int i = 0; i < str.size(); i++){
            if(str[i] > help[i]) return true;
            else if(str[i] < help[i]) return false;
        }
        return false;
    }
    int myAtoi(string str) {
        // ans: number, sign: +1 or -1
```

```
        int ans = 0;
        int sign = 1;
        int i = 0;
        int N = str.size();

        // eliminate spaces
        while(i < N){
            if(isspace(str[i]))
                i++;
            else
                break;
        }

        // if the whole string contains only spaces, return
        if(i == N) return ans;


        if(str[i] == '+')
            i++;
        else if(str[i] == '-'){
            sign = -1;
            i++;
        }

        // "help" gets the string of valid numbers
        string help;
        while(i < N){
            if('0' <= str[i] and str[i] <= '9')
                help += str[i++];
            else
                break;
        }

        const string maxINT = "2147483647";
        const string minINT = "2147483648";

        // test whether overflow, test only number parts with both signs

        if(sign == 1){
            if(overflow(help, maxINT)) return INT_MAX;
        }
        else{
            if(overflow(help, minINT)) return INT_MIN;
        }

        for(int j=0; j<help.size(); j++){
            ans = 10 * ans + int(help[j] - '0');
        }

        return ans*sign;
    }
```

```
};
```

## 源碼分析

C++解法並沒有假設任何內建的string演算法，因此也適合使用在純C語言的字元陣列上，此外溢位的判斷直接使用字串用一個輔助函數比大小，這樣如果面試官要求改成string to long 也有辦法應付，不過此方法會變成machine-dependent，嚴格來說還需要寫一個輔助小函數把 `INT_MAX` 和 `INT_MIN` 轉換成字串來使用，這邊就先省略了，有興趣的同學可以自己嘗試練習。

## 複雜度分析

略

# Reference

- String to Integer (atoi) 參考程序 Java/C++/Python

# Minimum Subarray

## Question

- lintcode: (44) Minimum Subarray

```
Given an array of integers, find the subarray with smallest sum.

Return the sum of the subarray.

Example
For [1, -1, -2, 1], return -3

Note
The subarray should contain at least one integer.
```

## 題解

題目 Maximum Subarray 的變形，使用區間和容易理解和實現。

### Java

```java
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: A integer indicate the sum of minimum subarray
     */
    public int minSubArray(ArrayList<Integer> nums) {
        if (nums == null || nums.isEmpty()) return -1;

        int sum = 0, maxSum = 0, minSub = Integer.MAX_VALUE;
        for (int num : nums) {
            maxSum = Math.max(maxSum, sum);
            sum += num;
            minSub = Math.min(minSub, sum - maxSum);
        }

        return minSub;
    }
}
```

## 源碼分析

略

## 複雜度分析

略

# Valid Sudoku

## Question

- leetcode: Valid Sudoku | LeetCode OJ
- lintcode: (389) Valid Sudoku

```
Determine whether a Sudoku is valid.

The Sudoku board could be partially filled,
where empty cells are filled with the character ..

Example
The following partially filed sudoku is valid.
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

```
Valid Sudoku

Note
A valid Sudoku board (partially filled) is not necessarily solvable.
Only the filled cells need to be validated.
Clarification
What is Sudoku?

http://sudoku.com.au/TheRules.aspx
https://zh.wikipedia.org/wiki/%E6%95%B8%E7%8D%A8
https://en.wikipedia.org/wiki/Sudoku
http://baike.baidu.com/subview/961/10842669.htm
```

## 題解

看懂數獨的含義就好了，分為三點考慮，一是每行無重複數字；二是每列無重複數字；三是小的九宮格中無重複數字。

## Java

```
class Solution {
    /**
      * @param board: the board
        @return: wether the Sudoku is valid
      */
    public boolean isValidSudoku(char[][] board) {
        if (board == null || board.length == 0) return false;

        // check row
        for (int i = 0; i < 9; i++) {
            boolean[] numUsed = new boolean[9];
            for (int j = 0; j < 9; j++) {
                if (isDuplicate(board[i][j], numUsed)) {
                    return false;
                }
            }
        }

        // check column
        for (int i = 0; i < 9; i++) {
            boolean[] numUsed = new boolean[9];
            for (int j = 0; j < 9; j++) {
                if (isDuplicate(board[j][i], numUsed)) {
                    return false;
                }
            }
        }

        // check sub box
        for (int i = 0; i < 9; i = i + 3) {
            for (int j = 0; j < 9; j = j + 3) {
                if (!isValidBox(board, i, j)) {
                    return false;
                }
            }
        }

        return true;
    }

    private boolean isValidBox(char[][] box, int x, int y) {
        boolean[] numUsed = new boolean[9];
        for (int i = x; i < x + 3; i++) {
            for (int j = y; j < y + 3; j++) {
```

```
                if (isDuplicate(box[i][j], numUsed)) {
                    return false;
                }
            }
        }
        return true;
    }

    private boolean isDuplicate(char c, boolean[] numUsed) {
        if (c == '.') {
            return false;
        } else if (numUsed[c - '1']) {
            return true;
        } else {
            numUsed[c - '1'] = true;
            return false;
        }
    }
}
```

## C++

```
class Solution {
public:
    bool isValidBlock(vector<int>& v){
        bool ans = true;
        for(int i = 0; i < v.size(); i++){
            if(v[i] > 1)
                ans = false;
            v[i] = 0;
        }
        return ans;
    }
    bool isValidSudoku(vector<vector<char>>& board) {
        vector<int> num(9, 0);

        //row
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                char n = board[i][j];
                if('1' <= n and n <= '9')
                    num[n - '1'] ++;
            }

            if(!isValidBlock(num))
                return false;
        }

        //col
```

```
        for(int j=0; j<9; j++){
            for(int i=0; i<9; i++){
                char n = board[i][j];
                if('1' <= n and n <= '9')
                    num[n - '1']++;
            }
            if(!isValidBlock(num))
                return false;
        }

        //block

        for(int row = 0; row < 3; row++){
            for(int col = 0; col < 3; col++){
                for(int i = 0; i < 3; i++){
                    for(int j = 0; j < 3; j++){
                        char n = board[3*row +i][3*col+j];
                        if('1' <= n and  n <= '9')
                            num[n - '1']++;
                    }
                }
                if(!isValidBlock(num))
                    return false;
            }
        }
        return true;
    }
};
```

## 源碼分析

首先實現兩個小的子功能模塊判斷是否有重複和小的九宮格是否重複。

## 複雜度分析

略

## Reference

- Soulmachine 的 leetcode 題解

# Add Binary

## Question

- leetcode: Add Binary | LeetCode OJ
- lintcode: (408) Add Binary

```
Given two binary strings, return their sum (also a binary string).

For example,
a = "11"
b = "1"
Return "100".
```

## 題解

用字串模擬二進制的加法，加法操作一般使用自後往前遍歷的方法，不同位大小需要補零。

### Java

```java
public class Solution {
    /**
     * @param a a number
     * @param b a number
     * @return the result
     */
    public String addBinary(String a, String b) {
        if (a == null || a.length() == 0) return b;
        if (b == null || b.length() == 0) return a;

        StringBuilder sb = new StringBuilder();
        int aLen = a.length(), bLen = b.length();

        int carry = 0;
        for (int ia = aLen - 1, ib = bLen - 1; ia >= 0 || ib >= 0; ia--, ib--) {
            // replace with 0 if processed
            int aNum = (ia < 0) ? 0 : a.charAt(ia) - '0';
            int bNum = (ib < 0) ? 0 : b.charAt(ib) - '0';

            int num = (aNum + bNum + carry) % 2;
            carry = (aNum + bNum + carry) / 2;
            sb.append(num);
        }
        if (carry == 1) sb.append(1);
```

```
        // important!
        sb.reverse();
        String result = sb.toString();
        return result;
    }
}
```

## 源碼分析

用到的技巧主要有兩點，一是兩個數位數大小不一時用0補上，二是最後需要判斷最高位的進位是否為1。最後需要反轉字符串，因為我們是從低位往高位迭代的。雖然可以使用 insert 避免最後的 reverse 操作，但如此一來時間複雜度就從 $O(n)$ 變為 $O(n^2)$ 了。

## C++

```
class Solution {
public:
    // helper functions
    char XOR(char x, char y) {
        return x == y ? '0' : '1';
    }
    char CARRY(char x, char y){
        return (x == '1' and y == '1') ? '1' : '0';
    }
    string addBinary(string a, string b) {
        if(a.size() < b.size())
            swap(a,b);

        int i = a.size()-1;
        int j = b.size()-1;
        char carry ='0';

        while(0 <= i) {
            char tmp = a[i];
            a[i] = XOR(tmp, carry);
            carry = CARRY(tmp, carry);
            if(0 <= j) {
                tmp = a[i];
                a[i] = XOR(tmp, b[j]);
                carry = XOR(carry, CARRY(tmp, b[j]));
            }
            i--; j--;
        }

        if(carry == '1')
            a = "1" + a;
        return a;
    }
```

```
};
```

## 源碼分析

C++的解法採用了直接操作char的作法，模擬硬體半加器(half adder)的行為，先確保a的長度不小於b的長度後，下標從尾到頭逐位相加，小心處理進位即可。

## 複雜度分析

Java解法遍歷兩個字串，時間複雜度 $O(n)$. reverse 操作時間複雜度 $O(n)$, 故總的時間複雜度 $O(n)$. 使用了 StringBuilder 作為臨時存儲對象，空間複雜度 $O(n)$.

C++解法時間複雜度也是$O(n)$，計算過程中使用的臨時變數，額外空間複雜度是$O(1)$，實際整體空間複雜度則取決於最後我們要將答案擴張一位時，函數的實現方法，最差可能達到$O(n)$。

# Reverse Integer

## Question

- leetcode: Reverse Integer | LeetCode OJ
- lintcode: (413) Reverse Integer

### Problem Statement

Reverse digits of an integer. Returns 0 when the reversed integer overflows (signed 32-bit integer).

#### Example

Given x = 123, return 321

Given x = -123, return -321

## 題解

初看這道題覺得先將其轉換爲字符串然後轉置一下就好了，但是仔細一想這種方法存在兩種缺陷，一是負號需要單獨處理，而是轉置後開頭的0也需要處理。另一種方法是將原數字逐個彈出，然後再將彈出的數字組裝爲新數字，乍看以爲需要用到 stack ，實際上卻是 queue... 所以根本不需要輔助資料結構。關於正負號的處理，我最開始是單獨處理的，後來看其他答案時才發現根本就不用分正負考慮。因爲 `-1 / 10 = 0` .

## Java

```
public class Solution {
    /**
     * @param n the integer to be reversed
     * @return the reversed integer
     */
    public int reverseInteger(int n) {
        long result = 0;
        while (n != 0) {
            result = n % 10 + 10 * result;
            n /= 10;
        }

        if (result < Integer.MIN_VALUE || result > Integer.MAX_VALUE) {
            return 0;
        }
        return (int)result;
    }
}
```

## 源碼分析

注意 lintcode 和 leetcode 的方法名不一樣。使用 long 型保存中間結果，最後判斷是否溢出。

## Reference

- LeetCode-Sol-Res/ReverseInt.java at master · FreeTymeKiyan/LeetCode-Sol-Res

# Find the Missing Number

## Question

- lintcode: (196) Find the Missing Number
- Find the Missing Number - GeeksforGeeks

### Problem Statement

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

#### Example

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

#### Challenge

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

## 題解1 - 位運算

和找單數的題類似，這裡我們不妨試試位運算中 Exlcusive Or (XOR) 的思路。最開始自己想到的是利用相鄰項異或結果看是否會有驚喜，然而發現 `a^(a+1) != a^a + a^1` 之後眼淚掉下來... 如果按照找單數的做法，首先對陣列所有元素異或，得到數 `x1`，現在的問題是如何利用 `x1` 得到缺失的數，由於找單數中其他數都是成對出現的，故最後的結果即是單數，這裏每個數都是單數，怎麼辦呢？我們現在再來分析下如果沒有缺失數的話會是怎樣呢？假設所有元素異或得到數 `x2`，數 `x1` 和 `x2` 有什麼差異呢？假設缺失的數是 `x0`，那麼容易知道 `x2 = x1 ^ x0`，相當於現在已知 `x1` 和 `x2`，要求 `x0`. 根據 Bit Manipulation 中總結的交換律，`x0 = x1 ^ x2`.

位運算的題往往比較靈活，需要好好利用常用等式變換。

### Java

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: an integer
     */
    public int findMissing(int[] nums) {
        if (nums == null || nums.length == 0) return -1;

        // get xor from 0 to N excluding missing number
        int x1 = 0;
        for (int i : nums) {
            x1 ^= i;
```

```
        }

        // get xor from 0 to N
        int x2 = 0;
        for (int i = 0; i <= nums.length; i++) {
            x2 ^= i;
        }

        // missing = x1 ^ x2;
        return x1 ^ x2;
    }
}
```

## 源碼分析

略

## 複雜度分析

遍歷原陣列和 N+1大小的陣列，時間複雜度 $O(n)$, 空間複雜度 $O(1)$.

# 題解2 - 桶排序

非常簡單直觀的想法——排序後檢查缺失元素，但是此題中要求時間複雜度爲 $O(n)$, 因此如果一定要用排序來做，那一定是使用非比較排序如桶排序或者計數排序。題中另一提示則是要求只使用 $O(1)$ 的額外空間，那麼這就是在提示我們應該使用原地交換。根據題意，元素應無重複，可考慮使用桶排，索引和值一一對應即可。第一重 for 循環遍歷原陣列，內循環使用 while, 調整索引處對應的值，直至相等或者索引越界爲止，for 循環結束時桶排結束。最後再遍歷一次陣列找出缺失元素。

初次接觸這種題還是比較難想到使用桶排這種思想的，尤其是利用索引和值一一對應這一特性找出缺失元素，另外此題在實際實現時不容易做到 bug-free, while 循環處容易出現死循環。

## Java

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: an integer
     */
    public int findMissing(int[] nums) {
        if (nums == null || nums.length == 0) return -1;

        bucketSort(nums);
        // find missing number
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != i) {
                return i;
```

```
            }
        }

        return nums.length;
    }

    private void bucketSort(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            while (nums[i] != i) {
                // ignore nums[i] == nums.length
                if (nums[i] == nums.length) {
                    break;
                }
                int nextNum = nums[nums[i]];
                nums[nums[i]] = nums[i];
                nums[i] = nextNum;
            }
        }
    }
}
```

## 源碼分析

難點一在於正確實現桶排，難點二在於陣列元素中最大值 N 如何處理。N 有三種可能：

1. N 不在原陣列中，故最後應該返回 N
2. N 在原陣列中，但不在陣列中的最後一個元素
3. N 在原陣列中且在陣列最後一個元素

其中情況1在遍歷桶排後的陣列時無返回，最後返回 N.

其中2和3在 while 循環處均會遇到 break 跳出，即當前這個索引所對應的值要麼最後還是 N，要麼就是和索引相同的值。如果最後還是 N, 也就意味着原陣列中缺失的是其他值，如果最後被覆蓋掉，那麼桶排後的陣列不會出現 N, 且缺失的一定是 N 之前的數。

綜上，這裏的實現無論 N 出現在哪個索引都能正確返回缺失值。實現上還是比較巧妙的，所以說在沒做過這類題時要在短時間內 bug-free 比較難，當然也可能是我比較菜...

另外一個難點在於如何保證或者證明 while 一定不會出現死循環，可以這麼理解，如果 while 條件不成立且未出現 `nums.length` 這個元素，那麼就一定會使得一個元素正確入桶，又因爲沒有重復元素出現，故一定不會出現死循環。

## 複雜度分析

桶排時間複雜度 $O(n)$, 空間複雜度 $O(1)$. 遍歷原陣列找缺失數時間複雜度 $O(n)$. 故總的時間複雜度爲 $O(n)$, 空間複雜度 $O(1)$.

# Part III - Contest

本節主要總結一些如 Google APAC, Microsoft 校招等線上測試的題目。

# Appendix I Interview and Resume

本章主要總結一些技術面試和撰寫履歷方面的注意事項。

# Tags