Algorithm

http://algorithm.yuanbin.me

目錄

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

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

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

14. Exhaustive Search
    i. Subsets
    ii. Unique Subsets
    iii. Permutations
    iv. Unique Permutations
    v. Next Permutation
    vi. Previous Permuation
    vii. Unique Binary Search Trees II
    viii. Permutation Index
    ix. Permutation Index II
    x. Permutation Sequence
    xi. Palindrome Partitioning
    xii. Combinations
    xiii. Combination Sum
    xiv. Combination Sum II
    xv. Minimum Depth of Binary Tree
15. Dynamic Programming
    i. Triangle
    ii. Backpack
    iii. Minimum Path Sum
    iv. Unique Paths
    v. Unique Paths II
    vi. Climbing Stairs
    vii. Jump Game
    viii. Word Break
    ix. Longest Increasing Subsequence
    x. Palindrome Partitioning II
    xi. Longest Common Subsequence
    xii. Edit Distance
    xiii. Jump Game II
    xiv. Best Time to Buy and Sell Stock
    xv. Best Time to Buy and Sell Stock II
    xvi. Best Time to Buy and Sell Stock III
    xvii. Best Time to Buy and Sell Stock IV
    xviii. Distinct Subsequences
    xix. Interleaving String
    xx. Maximum Subarray
    xxi. Maximum Subarray II
    xxii. Longest Increasing Continuous subsequence
    xxiii. Longest Increasing Continuous subsequence II
16. Graph
    i. Topological Sorting
17. Data Structure
    i. Implement Queue by Two Stacks
    ii. Min Stack
    iii. Sliding Window Maximum

    iv. Longest Words
18. Problem Misc
    i. Nuts and Bolts Problem
    ii. Heapify
    iii. String to Integer
    iv. Insert Interval
    v. Merge Intervals
    vi. Minimum Subarray
    vii. Matrix Zigzag Traversal
    viii. Valid Sudoku
    ix. Add Binary
    x. Reverse Integer
    xi. Gray Code
19. Part III - Contest
20. Google APAC
    i. APAC 2015 Round B
        i. Problem A. Password Attacker
21. Appendix I Interview and Resume
    i. Interview
    ii. Resume

資料結構與演算法/leetcode/lintcode題解

GITTER JOIN CHAT → build error

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 中 star 該項目查看更新，RSS 種子功能正在開發中。

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, GitCafe(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, GitCafe(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, GitCafe(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 who?
- 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

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。

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**是插入和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。

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;
}
```

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。

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。

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。

節點定義

這裡的節點統一使用LeetCode的定義

Python

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

C++

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

Java

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

相關演算法——遞迴法遍歷

請參考Chapter 11

相關演算法——分治法(Divide & Conquer)

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征：
    1. 問題的規模縮小到一定的程度就可以容易地解決。
    2. 問題可以分解為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為該問題的解。
    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）：將原問題分解為若干子問題，這些子問題都是原問題規模較小的實例。
    2. 解決（Conquer）：遞迴求解各子問題。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為原問題的解。
- 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. 合併排序(merge sort)
    6. 快速排序
    7. 循環賽日程表
    8. 河內塔(tower of Hanoi)

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

定義：

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(key)及相應的值(value)，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組，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數組搜尋的高效性結合起來的高效符號表實現。

二元樹中每個節點只能有一個父節點(根節點無父節點)，只有左右兩個連結，分別為**左子節點**和**右子節點**。

基本實現

節點包含

- 一個鍵
- 一個值
- 一條左鏈接
- 一條右鏈接

```
template<typename Key, typename Value>
struct BSTNode{
    Key key;
    Value val;
    BSTNode* left;
    BSTNode* right;
    BSTNode(Key k, Value v, BSTNode* l = NULL, BSTNode* r = NULL)
        :key(k), val(v), left(l), right(r){}
};
```

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	10

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。

Priority Queue - 優先隊列

應用程序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，例如作業系統(operating system)中的任務調度。

優先隊列與其說是資料結構，不如說是一種抽象資料型別(Abstract Data Type，ADT)，其介面(interface)至少需要三個基本的方法(method)：

- 插入一筆優先級別資料 (insert_with_priority)
- 取出最優先資料 (pull_highest_priority_element)
- 查看最優先資料 (peak) 若使用C++的STL提供的介面則如下所示

```
template <typename T> class priority_queue{
    void push (const T& val);
    void pop ();
    const T& top() const;
};
```

優先隊列可以使用數組或鏈表實現，從時間和空間複雜度來說，往往用堆(heap)來實現。

Reference

- 優先佇列 - 維基百科，自由的百科全書
- STL: priority_queue

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

6 5 3 1 8 7 2 4

Implementation

Python

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


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

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

        alist[index] = item_i

    return alist

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

Java

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

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

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

實現(C++):

```
template
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
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=""
            // 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
```

C++

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

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

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

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

Source

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

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

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

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

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

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

Challenge
O(n) time.

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

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

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

```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

Source

- 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

Source

- lintcode: (55) Compare Strings

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

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

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

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

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

Source

- lintcode: (8) Rotate String

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

Example
Given "abcdefg"

for offset=0, return "abcdefg"

for offset=1, return "gabcdef"

for offset=2, return "fgabcde"

for offset=3, return "efgabcd"

...
```

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]

Source

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

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

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

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

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

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
     */
```

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

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

        return true;
    }
}
```

源碼分析

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]

Source

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

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

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

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 {
    /**
     * @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` 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

Source

- lintcode: (212) Space Replacement

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

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

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

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

Challenge
Do it in-place.
```

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

Source

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

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

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

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

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

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

Given an integer n, generate the nth sequence.

Example
Given n = 5, return "111221".

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

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

Source

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

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

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

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

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

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

Source

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

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

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

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

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++

```cpp
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

Source

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

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

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

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

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

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++

```
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

Source

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

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

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

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

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

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;
    }
};
```

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
如果ans>0說明S[k]過大，k向前移；如果ans
```

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

Source

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

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

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

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

Source

- lintcode: (100) Remove Duplicates from Sorted Array

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

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

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

Source

- lintcode: (101) Remove Duplicates from Sorted Array II

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

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

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

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

Source

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

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

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

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

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

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

Source

- lintcode: (64) Merge Sorted Array II

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

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

B=[2,4,5,6]

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

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

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<Integer> B
        if (A == null || A.isEmpty()) return B;
        if (B == null || B.isEmpty()) return A;

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

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

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

        return C;
    }
}
```

源碼分析

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

Source

- lintcode: lintcode - (14) Binary Search

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

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

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

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

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

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` ， `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

Source

- lintcode: (60) Search Insert Position

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

You may assume no duplicates in the array.

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

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

Source

- lintcode: (61) Search for a Range

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

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

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

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

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` ， `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 searchRange(vector &A, int target) {
        // good, fail are the result
        // When found, returns good, otherwise returns fail
        int N = A.size();
        vector fail = {-1, -1};
        if(N == 0)
            return fail;
        vector 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

Source

- 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
You can determine whether a version is bad by the following interface:

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

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

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

Example
Given n=5

Call isBadVersion(3), get false

Call isBadVersion(5), get true

Call isBadVersion(4), get true

return 4 is the first bad version

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

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
# 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。

```
// 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

Source

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

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

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

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

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

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

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

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

Challenge
Time complexity O(logN)
```

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。

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

Source

- 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！

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

Source

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

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

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

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

Source

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

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

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

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; }
 * }
 */
```

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

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

        return dummy.next;
    }
}
```

源碼分析

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

Reverse Linked List

Source

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

```
Reverse a linked list.

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

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

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

Source

- lintcode: (165) Merge Two Sorted Lists
- leetcode: Merge Two Sorted Lists | LeetCode OJ

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

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

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

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

Source

- leetcode: Binary Tree Preorder Traversal | LeetCode OJ
- lintcode: (66) Binary Tree Preorder Traversal

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

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

   1
    \
     2
    /
   3

return [1,2,3].

Example
Challenge
Can you do it without recursion?
```

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

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

```python
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

```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) {
            // 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

```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;
        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

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def preorderTraversal(self, root):
        if root is None:
            return []

        result = []
        s = []
        s.append(root)
        while s:
            root = s.pop()
            result.append(root.val)
            if root.right is not None:
                s.append(root.right)
            if root.left is not None:
                s.append(root.left)

        return result
```

C++

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        if (root == NULL) return result;

        stack<TreeNode *> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *node = s.top();
            s.pop();
            result.push_back(node->val);
            if (node->right != NULL) {
                s.push(node->right);
            }
            if (node->left != NULL) {
                s.push(node->left);
            }
        }

        return result;
    }
};
```

Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root == null) return result;
```

```
        Stack<TreeNode> s = new Stack<TreeNode>();
        s.push(root);
        while (!s.empty()) {
            TreeNode node = s.pop();
            result.add(node.val);
            if (node.right != null) s.push(node.right);
            if (node.left != null) s.push(node.left);
        }

        return result;
    }
}
```

源碼分析

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 $O(n)$.

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

Source

- leetcode: Subsets | LeetCode OJ
- lintcode: (17) Subsets

```
Given a set of distinct integers, return all possible subsets.

Note
Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

Example
If S = [1,2,3], a solution is:

[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

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

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

Source

- leetcode: String to Integer (atoi) | LeetCode OJ
- lintcode: (54) String to Integer(atoi)

```
Implement function atoi to convert a string to an integer.

If no valid conversion could be performed, a zero value is returned.

If the correct value is out of the range of representable values,
INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

Example
"10" => 10

"-1" => -1

"123123123123123" => 2147483647

"1.0" => 1
```

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

Valid Sudoku

Source

- leetcode: Valid Sudoku | LeetCode OJ
- lintcode: (389) Valid Sudoku

```
Determine whether a Sudoku is valid.

The Sudoku board could be partially filled,
where empty cells are filled with the character ..

Example
The following partially filed sudoku is valid.
```

```

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

```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

Source

- leetcode: Add Binary | LeetCode OJ
- lintcode: (408) Add Binary

```
Given two binary strings, return their sum (also a binary string).

For example,
a = "11"
b = "1"
Return "100".
```

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$O(n)$。