

# LeetCode Summary

deepwaterooo

January 26, 2015



# Contents

<b>1</b>	<b>数组 Array</b>	<b>5</b>
1.1	Easy	6
1.1.1	Remove Element	6
1.1.2	Remove Duplicates from Sorted Array	6
1.1.3	Plus One	6
1.1.4	Pascal's Triangle	6
1.1.5	Pascal's Triangle II	6
1.1.6	Merge Sorted Array	6
1.1.7	Majority Element	6
1.2	Medium	6
1.2.1	Two Sum	6
1.2.2	3Sum Closest	6
1.2.3	Container With Most Water	6
1.2.4	Set Matrix Zeroes	6
1.2.5	Minimum Path Sum	6
1.2.6	Search Insert Position	6
1.2.7	Unique Paths	6
1.2.8	Search in Rotated Sorted Array II	6
1.2.9	Search for a Range	6
1.2.10	Search a 2D Matrix	6
1.2.11	Rotate Image	6
1.2.12	3Sum	6
1.2.13	4Sum	6
1.2.14	Remove Duplicates from Sorted Array II	6
1.2.15	Best Time to Buy and Sell Stock II	6
1.2.16	Unique Paths II	6
1.2.17	Best Time to Buy and Sell Stock	6
1.2.18	Next Permutation	6
1.2.19	Missing Ranges	6
1.2.20	Find Minimum in Rotated Sorted Array	6
1.2.21	Combination Sum II	6
1.2.22	Two Sum II - Input array is sorted	6
1.2.23	Triangle	6
1.2.24	Maximum Subarray	6
1.2.25	Maximum Product Subarray	6
1.2.26	Word Search	6
1.2.27	Combination Sum	6
1.2.28	Subsets	6
1.2.29	Subsets II	6
1.2.30	Jump Game	6
1.2.31	Spiral Matrix	6
1.2.32	Spiral Matrix II	6
1.2.33	Sort Colors	6
1.2.34	Find Peak Element	6
1.3	Hard	6
1.3.1	Trapping Rain Water	6
1.3.2	Find Minimum in Rotated Sorted Array II	6
1.3.3	Insert Interval	6

1.3.4	Jump Game II	6
1.3.5	Largest Rectangle in Histogram	6
1.3.6	Longest Consecutive Sequence	6
1.3.7	First Missing Positive	6
1.3.8	Median of Two Sorted Arrays	6
1.3.9	Merge Intervals	6
1.3.10	Word Ladder II	6
1.3.11	Best Time to Buy and Sell Stock III	6
1.3.12	Search in Rotated Sorted Array	6
1.3.13	Maximal Rectangle	6
1.4	Others	6
1.4.1	Construct Binary Tree from Inorder and Postorder Traversal	6
1.4.2	Construct Binary Tree from Preorder and Inorder Traversal	6
2	字符串 String	7
2.1	Easy	8
2.1.1	Valid Parentheses	8
2.1.2	Add Binary	8
2.1.3	Compare Version Numbers	8
2.1.4	Count and Say	8
2.1.5	Longest Common Prefix	8
2.1.6	Roman to Integer	8
2.1.7	Valid Palindrome	8
2.1.8	Read N Characters Given Read4	8
2.1.9	Implement strStr()	8
2.1.10	ZigZag Conversion	8
2.1.11	String to Integer (atoi)	8
2.1.12	Length of Last Word	8
2.2	Medium	8
2.2.1	Restore IP Addresses	8
2.2.2	Anagrams	8
2.2.3	Integer to Roman	8
2.2.4	Multiply Strings	8
2.2.5	One Edit Distance	8
2.2.6	Longest Palindromic Substring	8
2.2.7	Letter Combinations of a Phone Number	8
2.2.8	Decode Ways	8
2.2.9	Longest Substring Without Repeating Characters	8
2.2.10	Generate Parentheses	8
2.2.11	Reverse Words in a String	8
2.2.12	Simplify Path	8
2.3	Hard	8
2.3.1	Edit Distance	8
2.3.2	Minimum Window Substring	8
2.3.3	Longest Valid Parentheses	8
2.3.4	Longest Substring with At Most Two Distinct Characters	8
2.3.5	Read N Characters Given Read4 II - Call multiple times	8
2.3.6	Regular Expression Matching	8
2.3.7	Interleaving String	8
2.3.8	Wildcard Matching	8
2.3.9	Scramble String	8
2.3.10	Distinct Subsequences	8
2.3.11	Substring with Concatenation of All Words	8
2.3.12	Text Justification	8
2.3.13	Valid Number	8
2.3.14	Word Ladder II	8

3	Linked List	9
3.1	Easy	9
3.1.1	Remove Nth Node From End of List	9
3.1.2	Remove Duplicates from Sorted List	9
3.1.3	Merge Two Sorted Lists	9
3.1.4	Intersection of Two Linked Lists	9
3.2	Medium	9
3.2.1	Reverse Linked List II	9
3.2.2	Reorder List	9
3.2.3	Convert Sorted List to Binary Search Tree	9
3.2.4	Rotate List	9
3.2.5	Remove Duplicates from Sorted List II	9
3.2.6	Sort List	9
3.2.7	Insertion Sort List	9
3.2.8	Swap Nodes in Pairs	9
3.2.9	Linked List Cycle	9
3.2.10	Linked List Cycle II	9
3.2.11	Add Two Numbers	9
3.2.12	Partition List	9
3.3	Hard	9
3.3.1	Copy List with Random Pointer	9
3.3.2	Merge k Sorted Lists	9
3.3.3	Reverse Nodes in k-Group	9
4	树 Binary Tree, Binary Search Tree	11
4.1	二叉树的遍历	12
4.1.1	Binary Tree Preorder Traversal	12
4.1.2	Binary Tree Inorder Traversal	12
4.1.3	Binary Tree Postorder Traversal	12
4.1.4	Binary Tree Level Order Traversal	12
4.1.5	Binary Tree Level Order Traversal II	12
4.1.6	Binary Tree Zigzag Level Order Traversal	12
4.1.7	Recover Binary Search Tree	12
4.1.8	Same Tree	12
4.1.9	Symmetric Tree	12
4.1.10	Balanced Binary Tree	12
4.1.11	Flatten Binary Tree to Linked List	12
4.1.12	Populating Next Right Pointers in Each Node II	12
4.2	二叉树的构建	12
4.2.1	Construct Binary Tree from Preorder and Inorder Traversal	12
4.2.2	Construct Binary Tree from Inorder and Postorder Traversal	12
4.3	二叉树查找	12
4.3.1	Unique Binary Search Trees	12
4.3.2	Unique Binary Search Trees II	12
4.3.3	Validate Binary Search Tree	12
4.3.4	Convert Sorted Array to Binary Search Tree	12
4.3.5	Convert Sorted List to Binary Search Tree	12
4.4	二叉树递归	12
4.4.1	Minimum Depth of Binary Tree	12
4.4.2	Maximum Depth of Binary Tree	12
4.4.3	Path Sum	12
4.4.4	Path Sum II	12
4.4.5	Binary Tree Maximum Path Sum	12
4.4.6	Populating Next Right Pointers in Each Node	12
4.4.7	Sum Root to Leaf Numbers	12
4.4.8	Binary Tree Upside Down: this 2 ?	12
4.4.9	Binary Search Tree Iterator	12

5	栈和队列	13
5.1	栈	13
5.2	Easy	13
5.2.1	Valid Parentheses	13
5.2.2	Min Stack	13
5.3	Medium	13
5.3.1	Binary Tree Inorder Traversal	13
5.3.2	Binary Search Tree Iterator	13
5.3.3	Evaluate Reverse Polish Notation	13
5.3.4	Simplify Path	13
5.3.5	Binary Tree Preorder Traversal	13
5.3.6	Binary Tree Zigzag Level Order Traversal	13
5.4	Hard	13
5.4.1	Binary Tree Postorder Traversal	13
5.4.2	Largest Rectangle in Histogram	13
5.4.3	Trapping Rain Water	13
5.4.4	Maximal Rectangle	13
5.5	队列	13
5.5.1		13
5.6	Heap:	13
5.6.1	Merge k sorted List	13
6	Hash Table	15
6.1	Easy	15
6.1.1	Valid Sudoku	15
6.1.2	Two Sum III	15
6.2	Medium	15
6.2.1	Two Sum	15
6.2.2	4Sum	15
6.2.3	Binary Tree Inorder Traversal	15
6.2.4	Fraction to Recurring Decimal	15
6.2.5	Single Number	15
6.2.6	Anagrams	15
6.2.7	Longest Substring Without Repeating Characters	15
6.3	Hard	15
6.3.1	Minimum Window Substring	15
6.3.2	Maximal Rectangle	15
6.3.3	Copy List with Random Pointer	15
6.3.4	Sudoku Solver	15
6.3.5	Max Points on a Line	15
6.3.6	Substring with Concatenation of All Words	15
6.3.7	Longest Substring with At Most Two Distinct Characters	15
7	广度优先搜索 Breadth First Search	17
7.1	Easy	17
7.1.1	Binary Tree Level Order Traversal	17
7.1.2	Binary Tree Level Order Traversal II	17
7.2	Medium	17
7.2.1	Word Ladder	17
7.2.2	Surrounded Regions	17
7.2.3	Clone Graph	17
7.2.4	Binary Tree Zigzag Level Order Traversal	17
7.2.5	Minimum Depth of Binary Tree	17
7.3	Hard	17
7.3.1	Word Ladder II	17

<b>8</b>	<b>深度优先搜索 Depth First Search</b>	<b>19</b>
8.1	Easy	19
8.1.1	Path Sum	19
8.1.2	Balanced Binary Tree	19
8.1.3	Same Tree	19
8.1.4	Symmetric Tree	19
8.1.5	Maximum Depth of Binary Tree	19
8.2	Medium	19
8.2.1	Sum Root to Leaf Numbers	19
8.2.2	Convert Sorted List to Binary Search Tree	19
8.2.3	Convert Sorted Array to Binary Search Tree	19
8.2.4	Construct Binary Tree from Preorder and Inorder Traversal	19
8.2.5	Construct Binary Tree from Inorder and Postorder Traversal	19
8.2.6	Clone Graph	19
8.2.7	Flatten Binary Tree to Linked List	19
8.2.8	Populating Next Right Pointers in Each Node	19
8.2.9	Validate Binary Search Tree	19
8.2.10	Path Sum II	19
8.3	Hard	19
8.3.1	Recover Binary Search Tree	19
8.3.2	Populating Next Right Pointers in Each Node II	19
8.3.3	Binary Tree Maximum Path Sum	19
<b>9</b>	<b>排序</b>	<b>21</b>
9.1	Medium	21
9.1.1	Sort Colors	21
9.1.2	Sort List	21
9.1.3	Largest Number	21
9.2	Hard	21
9.2.1	Insertion Sort List	21
9.2.2	Maximum Gap	21
9.2.3	Merge Intervals	21
9.2.4	Insert Interval	21
<b>10</b>	<b>查找 Binary Search</b>	<b>23</b>
10.1	Medium	23
10.1.1	Sqrt(x)	23
10.1.2	Divide Two Integers	23
10.1.3	Search in Rotated Sorted Array II	23
10.1.4	Two Sum II - Input array is sorted	23
10.1.5	Pow(x, n)	23
10.1.6	Search for a Range	23
10.1.7	Search a 2D Matrix	23
10.1.8	Find Minimum in Rotated Sorted Array	23
10.1.9	Search Insert Position	23
10.1.10	Find Peak Element	23
10.2	Hard	23
10.2.1	Median of Two Sorted Arrays	23
10.2.2	Dungeon Game	23
10.2.3	Find Minimum in Rotated Sorted Array II	23
10.2.4	Search in Rotated Sorted Array	23
<b>11</b>	<b>暴力枚举法</b>	<b>25</b>
11.1		25
11.1.1		25

<b>12</b>	<b>分治法 Divide and Conquer</b>	<b>27</b>
12.1	Easy	27
12.1.1	Majority Element	27
12.2	Medium	27
12.2.1	Maximum Subarray	27
12.3	Hard	27
12.3.1	Merge K sorted List	27
12.3.2	Median of Two Sorted Array	27
<b>13</b>	<b>贪心法 Greedy Search</b>	<b>29</b>
13.1	Medium	29
13.1.1	Gas Station	29
13.1.2	Best Time to Buy and Sell Stock II	29
13.1.3	Jump Game	29
13.2	Hard	29
13.2.1	Jump Game II	29
13.2.2	Candy	29
13.2.3	Wildcard Matching	29
13.2.4		29
<b>14</b>	<b>动态归划 Dynamic Programming</b>	<b>31</b>
14.1	Easy	31
14.1.1	Climbing Stairs	31
14.2	Medium	32
14.2.1	Unique Path	32
14.2.2	Unique Path II	32
14.2.3	Unique Binary Search Tree	32
14.2.4	Unique Binary Search Tree II	32
14.2.5	Minimum Path Sum	32
14.2.6	Maximum Sum Subarray	32
14.2.7	Maximum Product Subarray	32
14.2.8	Decode Ways	32
14.2.9	Triangle	32
14.2.10	Word Break	32
14.2.11	Best Time to Buy and Sell Stock	32
14.3	Hard	32
14.3.1	Best Time to Buy and Sell Stock III	32
14.3.2	Longest Valid Parentheses	32
14.3.3	Coins in a Line	32
14.3.4	Word Break II	32
14.3.5	Maximum Rectangle	32
14.3.6	Edit Distance	32
14.3.7	Distinct Subsequence	32
14.3.8	Palindrome Partitioning II	32
14.3.9	Interleaving String	33
14.3.10	Scramble String	33
14.3.11	Regular Expression Matching	33
14.3.12	Wildcard Matching	33
<b>15</b>	<b>Two Pointers and Sliding Window</b>	<b>35</b>
15.1	Easy	35
15.1.1	Valid Palindrome	35
15.1.2	Remove Nth Node From End of List	35
15.1.3	Remove Element	35
15.1.4	Remove Duplicates from Sorted Array	35
15.1.5	Merge Sorted Array	35
15.1.6	Implement strStr()	35
15.2	Medium	35
15.2.1	3Sum	35



15.2.2	4Sum	35
15.2.3	Container With Most Water	35
15.2.4	Remove Duplicates from Sorted Array II	35
15.2.5	Partition List	35
15.2.6	Two Sum II - Input array is sorted	35
15.2.7	Linked List Cycle II	35
15.2.8	Longest Substring Without Repeating Characters	35
15.2.9	3Sum Closest	35
15.2.10	Linked List Cycle	35
15.2.11	Sort Colors	35
15.2.12	Rotate List	35
15.3	Hard	35
15.3.1	Trapping Rain Water	35
15.3.2	Longest Substring with At Most Two Distinct Characters	35
15.3.3	Substring with Concatenation of All Words	35
15.3.4	Minimum Window Substring	35
16	Backtracing and Recursion	37
16.1	排列	37
16.1.1	Permutation	37
16.1.2	Permutation II	37
16.1.3	Permutation Sequence	37
16.2	组合	37
16.2.1	Combinationas	37
16.2.2	Combination Sum	38
16.2.3	Combination Sum II	38
16.3	Subsets	38
16.3.1	Subsets: Bit Manipulation	38
16.3.2	Subsets II	39
16.4	others with Recursion	39
16.4.1	Letter Combinationas of Phone Number	39
16.4.2	Restore IP Addresses	39
16.4.3	Generate Parentheses	39
16.4.4	Gray Code	39
16.4.5	Word Search	40
16.4.6	Palindrome Partitioning	40
16.4.7	N-Queens	41
16.4.8	N-Queens II	42
16.4.9	Sudoku Solver	42
16.4.10	Regular Expression Matching	42
16.4.11	Wild Card Matching	42
16.4.12	Word Break II	43
16.4.13	Word Ladder II	43
17	Bit Manipulation	47
17.1	Easy	47
17.1.1	Majority Element	47
17.2	Medium	47
17.2.1	Subsets: Bit Manipulation	47
17.2.2	Single Number	47
17.2.3	Single Number II	47
18	图 Graphics	49
18.1	Medium	49
18.1.1	Clone Graph	49
18.2	Word Ladder, Word Ladder II: Backtracing	51
18.2.1	Word Ladder	51
18.2.2	Word Ladder II	51
18.2.3	Check whether the graph is bigraph	52

<b>19 Data Structure</b>	<b>55</b>
19.1 Easy	55
19.1.1 Two Sum III	55
19.1.2 Min Stack	55
19.2 Hard	56
19.2.1 LRU Cache	56
<b>20 细节实现题</b>	<b>59</b>
<b>21 Math</b>	<b>61</b>
21.1 Easy	61
21.1.1 Add Binary	61
21.1.2 Roman to Integers	61
21.1.3 String to Integer (atoi)	61
21.1.4 Palindrome Number	61
21.1.5 Plus One	61
21.1.6 Factorial Trailing Zeroes	61
21.1.7 Excel Sheet Column Title	61
21.1.8 Excel Sheet Column Number	61
21.1.9 Reverse Integer	61
21.2 Medium	61
21.2.1 Multiply Strings	61
21.2.2 Sqrt(x)	61
21.2.3 Divide Two Integers	61
21.2.4 Pow(x, n)	61
21.2.5 Fraction to Recurring Decimal	61
21.2.6 Permutation Sequence	61
21.2.7 Integer to Roman	61
21.2.8 Next Permutation: Math	61
21.3 Hard	61
21.3.1 Valid Number	61
21.3.2 Max Points on a Line	61



# Chapter 1

## 数组 Array

### 1.1 Easy

- 1.1.1 Remove Element
- 1.1.2 Remove Duplicates from Sorted Array
- 1.1.3 Plus One
- 1.1.4 Pascal's Triangle
- 1.1.5 Pascal's Triangle II
- 1.1.6 Merge Sorted Array
- 1.1.7 Majority Element

### 1.2 Medium

- 1.2.1 Two Sum
- 1.2.2 3Sum Closest
- 1.2.3 Container With Most Water
- 1.2.4 Set Matrix Zeroes
- 1.2.5 Minimum Path Sum
- 1.2.6 Search Insert Position
- 1.2.7 Unique Paths
- 1.2.8 Search in Rotated Sorted Array II
- 1.2.9 Search for a Range
- 1.2.10 Search a 2D Matrix
- 1.2.11 Rotate Image
- 1.2.12 3Sum
- 1.2.13 4Sum
- 1.2.14 Remove Duplicates from Sorted Array II
- 1.2.15 Best Time to Buy and Sell Stock II
- 1.2.16 Unique Paths II
- 1.2.17 Best Time to Buy and Sell Stock
- 1.2.18 Next Permutation
- 1.2.19 Missing Ranges
- 1.2.20 Find Minimum in Rotated Sorted Array



# Chapter 2

## 字符串 String

### 2.1 Easy

2.1.1 Valid Parentheses

2.1.2 Add Binary

2.1.3 Compare Version Numbers

2.1.4 Count and Say

2.1.5 Longest Common Prefix

2.1.6 Roman to Integer

2.1.7 Valid Palindrome

2.1.8 Read N Characters Given Read4

2.1.9 Implement strStr()

2.1.10 ZigZag Conversion

2.1.11 String to Integer (atoi)

2.1.12 Length of Last Word

### 2.2 Medium

2.2.1 Restore IP Addresses

2.2.2 Anagrams

2.2.3 Integer to Roman

2.2.4 Multiply Strings

2.2.5 One Edit Distance

2.2.6 Longest Palindromic Substring

2.2.7 Letter Combinations of a Phone Number

2.2.8 Decode Ways

2.2.9 Longest Substring Without Repeating Characters

2.2.10 Generate Parentheses

2.2.11 Reverse Words in a String

2.2.12 Simplify Path

### 2.3 Hard

2.3.1 Edit Distance

# Chapter 3

## Linked List

### 3.1 Easy

3.1.1 Remove Nth Node From End of List

3.1.2 Remove Duplicates from Sorted List

3.1.3 Merge Two Sorted Lists

3.1.4 Intersection of Two Linked Lists

### 3.2 Medium

3.2.1 Reverse Linked List II

3.2.2 Reorder List

3.2.3 Convert Sorted List to Binary Search Tree

3.2.4 Rotate List

3.2.5 Remove Duplicates from Sorted List II

3.2.6 Sort List

3.2.7 Insertion Sort List

3.2.8 Swap Nodes in Pairs

3.2.9 Linked List Cycle

3.2.10 Linked List Cycle II

3.2.11 Add Two Numbers

3.2.12 Partition List

### 3.3 Hard

3.3.1 Copy List with Random Pointer

3.3.2 Merge k Sorted Lists

3.3.3 Reverse Nodes in k-Group







# Chapter 4

## 树 Binary Tree, Binary Search Tree

### 4.1 二叉树的遍历

4.1.1 Binary Tree Preorder Traversal

4.1.2 Binary Tree Inorder Traversal

4.1.3 Binary Tree Postorder Traversal

4.1.4 Binary Tree Level Order Traversal

4.1.5 Binary Tree Level Order Traversal II

4.1.6 Binary Tree Zigzag Level Order Traversal

4.1.7 Recover Binary Search Tree

4.1.8 Same Tree

4.1.9 Symmetric Tree

4.1.10 Balanced Binary Tree

4.1.11 Flatten Binary Tree to Linked List

4.1.12 Populating Next Right Pointers in Each Node II

### 4.2 二叉树的构建

4.2.1 Construct Binary Tree from Preorder and Inorder Traversal

4.2.2 Construct Binary Tree from Inorder and Postorder Traversal

### 4.3 二叉树查找

4.3.1 Unique Binary Search Trees

4.3.2 Unique Binary Search Trees II

4.3.3 Validate Binary Search Tree

4.3.4 Convert Sorted Array to Binary Search Tree

4.3.5 Convert Sorted List to Binary Search Tree

### 4.4 二叉树递归

4.4.1 Minimum Depth of Binary Tree

4.4.2 Maximum Depth of Binary Tree

4.4.3 Path Sum

4.4.4 Path Sum II

4.4.5 Binary Tree Maximum Path Sum

# Chapter 5

## 栈和队列

### 5.1 栈

### 5.2 Easy

#### 5.2.1 Valid Parentheses

#### 5.2.2 Min Stack

### 5.3 Medium

#### 5.3.1 Binary Tree Inorder Traversal

#### 5.3.2 Binary Search Tree Iterator

#### 5.3.3 Evaluate Reverse Polish Notation

#### 5.3.4 Simplify Path

#### 5.3.5 Binary Tree Preorder Traversal

#### 5.3.6 Binary Tree Zigzag Level Order Traversal

### 5.4 Hard

#### 5.4.1 Binary Tree Postorder Traversal

#### 5.4.2 Largest Rectangle in Histogram

#### 5.4.3 Trapping Rain Water

#### 5.4.4 Maximal Rectangle

### 5.5 队列

#### 5.5.1

### 5.6 Heap:

#### 5.6.1 Merge k sorted List

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

**Tags:** Divide and Conquer, Linked List, Heap

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {  
    if (l1 == null) return l2;  
    if (l2 == null) return l1;  
    ListNode result;  
    if (l1.val < l2.val) {  
        result = l1;  
        l1 = l1.next;  
        result.next = null;  
    } else {
```

```

        result = l2;
        l2 = l2.next;
        result.next = null;
    }
    ListNode curr = result;
    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            curr.next = l1;
            curr = curr.next;
            l1 = l1.next;
            curr.next = null;
        } else {
            curr.next = l2;
            curr = curr.next;
            l2 = l2.next;
            curr.next = null;
        }
    }
    if (l1 == null && l2 == null)
        return result;
    l1 = (l1 == null) ? l2 : l1;
    curr.next = l1;
    return result;
}

public ListNode mergeKLists(List<ListNode> lists) {
    if (lists.size() == 0) return null;
    if (lists.size() == 1) return lists.get(0);
    if (lists.size() == 2) return mergeTwoLists(lists.get(0), lists.get(1));
    return mergeTwoLists((mergeKLists(lists.subList(0, lists.size() / 2))),
        (mergeKLists(lists.subList(lists.size() / 2, lists.size()))));
}

```

# Chapter 6

## Hash Table

### 6.1 Easy

6.1.1 Valid Sudoku

6.1.2 Two Sum III

### 6.2 Medium

6.2.1 Two Sum

6.2.2 4Sum

6.2.3 Binary Tree Inorder Traversal

6.2.4 Fraction to Recurring Decimal

6.2.5 Single Number

6.2.6 Anagrams

6.2.7 Longest Substring Without Repeating Characters

### 6.3 Hard

6.3.1 Minimum Window Substring

6.3.2 Maximal Rectangle

6.3.3 Copy List with Random Pointer

6.3.4 Sudoku Solver

6.3.5 Max Points on a Line

6.3.6 Substring with Concatenation of All Words

6.3.7 Longest Substring with At Most Two Distinct Characters



# Chapter 7

## 广度优先搜索 Breadth First Search

### 7.1 Easy

#### 7.1.1 Binary Tree Level Order Traversal

#### 7.1.2 Binary Tree Level Order Traversal II

### 7.2 Medium

#### 7.2.1 Word Ladder

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

For example,

Given:

```
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.
```

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

#### 7.2.2 Surrounded Regions

#### 7.2.3 Clone Graph

#### 7.2.4 Binary Tree Zigzag Level Order Traversal

#### 7.2.5 Minimum Depth of Binary Tree

### 7.3 Hard

#### 7.3.1 Word Ladder II





# Chapter 8

## 深度优先搜索 Depth First Search

### 8.1 Easy

8.1.1 Path Sum

8.1.2 Balanced Binary Tree

8.1.3 Same Tree

8.1.4 Symmetric Tree

8.1.5 Maximum Depth of Binary Tree

### 8.2 Medium

8.2.1 Sum Root to Leaf Numbers

8.2.2 Convert Sorted List to Binary Search Tree

8.2.3 Convert Sorted Array to Binary Search Tree

8.2.4 Construct Binary Tree from Preorder and Inorder Traversal

8.2.5 Construct Binary Tree from Inorder and Postorder Traversal

8.2.6 Clone Graph

8.2.7 Flatten Binary Tree to Linked List

8.2.8 Populating Next Right Pointers in Each Node

8.2.9 Validate Binary Search Tree

8.2.10 Path Sum II

### 8.3 Hard

8.3.1 Recover Binary Search Tree

8.3.2 Populating Next Right Pointers in Each Node II

8.3.3 Binary Tree Maximum Path Sum



# Chapter 9

## 排序

### 9.1 Medium

#### 9.1.1 Sort Colors

#### 9.1.2 Sort List

#### 9.1.3 Largest Number

### 9.2 Hard

#### 9.2.1 Insertion Sort List

#### 9.2.2 Maximum Gap

#### 9.2.3 Merge Intervals

#### 9.2.4 Insert Interval



# Chapter 10

## 查找 Binary Search

### 10.1 Medium

10.1.1 Sqrt(x)

10.1.2 Divide Two Integers

10.1.3 Search in Rotated Sorted Array II

10.1.4 Two Sum II - Input array is sorted

10.1.5 Pow(x, n)

10.1.6 Search for a Range

10.1.7 Search a 2D Matrix

10.1.8 Find Minimum in Rotated Sorted Array

10.1.9 Search Insert Position

10.1.10 Find Peak Element

### 10.2 Hard

10.2.1 Median of Two Sorted Arrays

10.2.2 Dungeon Game

10.2.3 Find Minimum in Rotated Sorted Array II

10.2.4 Search in Rotated Sorted Array



# Chapter 11

## 暴力枚举法

### 11.1

#### 11.1.1





# Chapter 12

## 分治法 Divide and Conquer

### 12.1 Easy

#### 12.1.1 Majority Element

### 12.2 Medium

#### 12.2.1 Maximum Subarray

### 12.3 Hard

#### 12.3.1 Merge K sorted List

#### 12.3.2 Median of Two Sorted Array



# Chapter 13

## 贪心法 Greedy Search

### 13.1 Medium

#### 13.1.1 Gas Station

#### 13.1.2 Best Time to Buy and Sell Stock II

#### 13.1.3 Jump Game

### 13.2 Hard

#### 13.2.1 Jump Game II

#### 13.2.2 Candy

#### 13.2.3 Wildcard Matching

#### 13.2.4



# Chapter 14

## 动态归划 Dynamic Programming

### 14.1 Easy

#### 14.1.1 Climbing Stairs

You are climbing a stair case. It takes  $n$  steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

1.  $O(n)$

```
public int climbStairs(int n) {
    int [] res = new int[n + 1];
    res[0] = 1; // 1 stair
    res[1] = 2; // 2 stair
    for (int i = 2; i < n; i++)
        res[i] = res[i - 2] + res[i - 1];
    return res[n - 1];
}
```

Considering odd and even, 实现滚动数组

```
public int climbStairs(int n){
    if(n <= 0) return 0;
    int [] stairs = {1, 2};
    for(int i = 2; i < n; i++)
        stairs[i % 2] = stairs[0] + stairs[1];
    return n % 2 == 0 ? stairs[1] : stairs[0];
}
```

2.  $O(\log(n))$

<https://oj.leetcode.com/discuss/11211/o-log-n-solution-with-matrix-multiplication>

I saw most solutions posted in discussion are DP with runtime  $O(n)$  and  $O(1)$  space which is accepted by OJ.

The only  $O(\log(n))$  solution so far is lucastan's using Binet's formula.

There actually is a matrix multiplication solution which also runs in  $O(\log(n))$ . It basically calculates fibonacci numbers by power of matrix  $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{(n-1)}$ .

```
private int[][] pow(int[][] a, int n) {
    int[][] ret = {{1, 0}, {0, 1}};
    while (n > 0) {
        if ((n & 1) == 1)
            ret = multiply(ret, a);
        n >>= 1;
        a = multiply(a, a);
    }
    return ret;
}
```

```
private int[][] multiply(int[][] a, int[][] b) {
    int[][] c = new int[2][2];
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
```

```

        c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
    }

    return c;
}

public int climbStairs(int n) {
    int[][] a = {{0, 1}, {1, 1}};
    int[][] m = pow(a, n - 1);
    return m[0][1] + m[1][1];
}

```

## 14.2 Medium

### 14.2.1 Unique Path

### 14.2.2 Unique Path II

### 14.2.3 Unique Binary Search Tree

### 14.2.4 Unique Binary Search Tree II

### 14.2.5 Minimum Path Sum

### 14.2.6 Maximum Sum Subarray

### 14.2.7 Maximum Product Subarray

### 14.2.8 Decode Ways

### 14.2.9 Triangle

### 14.2.10 Word Break

Given a string `s` and a dictionary of words `dict`, determine if `s` can be segmented into a space-separated sequence of one or more dictionary words.

For example, given

```

s = "leetcode",
dict = ["leet", "code"].
Return true because "leetcode" can be segmented as "leet code".

```

### 14.2.11 Best Time to Buy and Sell Stock

## 14.3 Hard

### 14.3.1 Best Time to Buy and Sell Stock III

### 14.3.2 Longest Valid Parentheses

### 14.3.3 Coins in a Line

### 14.3.4 Word Break II

### 14.3.5 Maximum Rectangle

### 14.3.6 Edit Distance

### 14.3.7 Distinct Subsequence

### 14.3.8 Palindrome Partitioning II

Given a string `s`, partition `s` such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of `s`.

For example,

```

given s = "aab",
Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

```

**14.3.9 Interleaving String**

**14.3.10 Scramble String**

**14.3.11 Regular Expression Matching**

**14.3.12 Wildcard Matching**





# Chapter 15

## Two Pointers and Sliding Window

### 15.1 Easy

15.1.1 Valid Palindrome

15.1.2 Remove Nth Node From End of List

15.1.3 Remove Element

15.1.4 Remove Duplicates from Sorted Array

15.1.5 Merge Sorted Array

15.1.6 Implement strStr()

### 15.2 Medium

15.2.1 3Sum

15.2.2 4Sum

15.2.3 Container With Most Water

15.2.4 Remove Duplicates from Sorted Array II

15.2.5 Partition List

15.2.6 Two Sum II - Input array is sorted

15.2.7 Linked List Cycle II

15.2.8 Longest Substring Without Repeating Characters

15.2.9 3Sum Closest

15.2.10 Linked List Cycle

15.2.11 Sort Colors

15.2.12 Rotate List

### 15.3 Hard

15.3.1 Trapping Rain Water

15.3.2 Longest Substring with At Most Two Distinct Characters

15.3.3 Substring with Concatenation of All Words

15.3.4 Minimum Window Substring



# Chapter 16

## Backtracing and Recursion

### 16.1 排列

#### 16.1.1 Permutation

Given a collection of numbers, return all possible permutations.

For example,

[1,2,3] have the following permutations:

[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

#### 16.1.2 Permutation II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

[1,1,2] have the following unique permutations:

[1,1,2], [1,2,1], and [2,1,1].

#### 16.1.3 Permutation Sequence

The set [1,2,3,...,n] contains a total of  $n!$  unique permutations.

By listing and labeling all of the permutations in order, We get the following sequence (ie, for  $n = 3$ ):

```
"123"
"132"
"213"
"231"
"312"
"321"
```

Given  $n$  and  $k$ , return the  $k$ th permutation sequence.

Note: Given  $n$  will be between 1 and 9 inclusive.

### 16.2 组合

#### 16.2.1 Combinationas

Given two integers  $n$  and  $k$ , return all possible combinations of  $k$  numbers out of  $1 \cdots n$ .

For example,

If  $n = 4$  and  $k = 2$ , a solution is:

```
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

## 16.2.2 Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, ..., ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ ... ≤ ak).
- The solution set must not contain duplicate combinations.

For example, given candidate set 2,3,6,7 and target 7,

A solution set is:

```
[7]
[2, 2, 3]
```

## 16.2.3 Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

Each number in C may only be used once in the combination.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, ..., ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ ... ≤ ak).
- The solution set must not contain duplicate combinations.

For example, given candidate set 10,1,2,7,6,1,5 and target 8,

A solution set is:

```
[1, 7]
[1, 2, 5]
[2, 6]
[1, 1, 6]
```

## 16.3 Subsets

### 16.3.1 Subsets: Bit Manipulation

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

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,

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

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

**Tags:** Array Backtracking, Bit Manipulation

### 16.3.2 Subsets II

Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,

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

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

## 16.4 others with Recursion

### 16.4.1 Letter Combinationas of Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

Input:Digit string "23"

Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

### 16.4.2 Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

### 16.4.3 Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

"((()))", "(()())", "()(())", "()()()", "()"

### 16.4.4 Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note:

- For a given n, a gray code sequence is not uniquely defined.
- For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.
- For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.



Figure 16.1: Letter Combinationas of Phone Number

### 16.4.5 Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,

```
Given board =
[
  ["ABCE"],
  ["SFCS"],
  ["ADEE"]
]
word = "ABCCED", -> returns true,
word = "SEE", -> returns true,
word = "ABCB", -> returns false.
```

### 16.4.6 Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example,

```
given s = "aab",
Return
```

```
[
  ["aa", "b"],
  ["a", "a", "b"]
]
```

### 16.4.7 N-Queens

The n-queens puzzle is the problem of placing n queens on an  $n \times n$  chessboard such that no two queens attack each other.

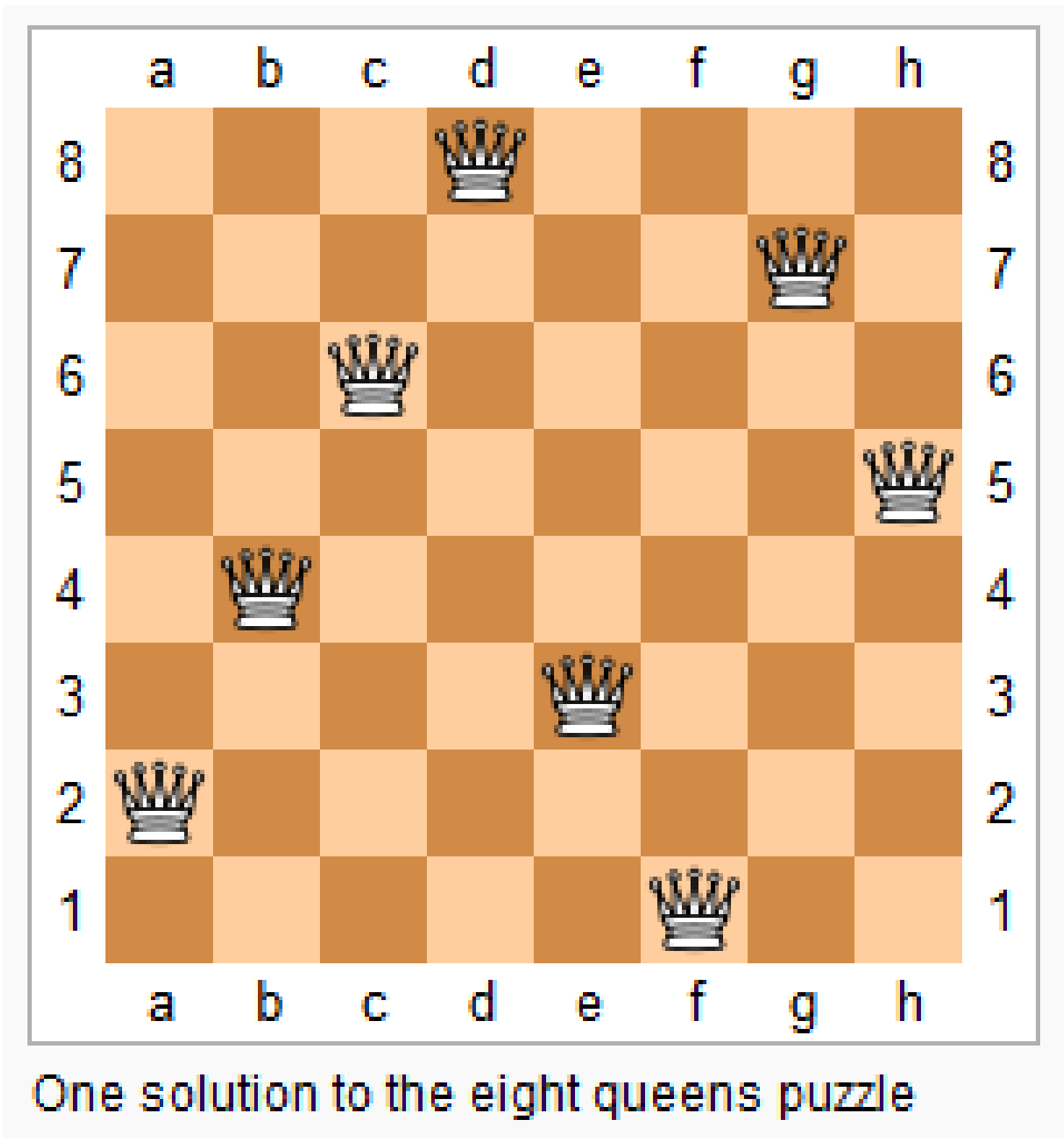


Figure 16.2: N-Queens

Given an integer  $n$ , return all distinct solutions to the  $n$ -queens puzzle.

Each solution contains a distinct board configuration of the  $n$ -queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

For example,

There exist two distinct solutions to the 4-queens puzzle:

```
[
[".Q...", // Solution 1
 "...Q",
 "Q...",
 "...Q."],

["...Q.", // Solution 2
 "Q...",
 "...Q.",
 ".Q..."]
]
```

### 16.4.8 N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

### 16.4.9 Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.

A sudoku puzzle...

...and its solution numbers marked in red.

**Tags:** Backtracking, Hash Table

### 16.4.10 Regular Expression Matching

Implement regular expression matching with support for '.' and '\*'.

- '.' Matches any single character.
- '\*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

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

**Tags:** Dynamic Programming, Backtracking, String

### 16.4.11 Wild Card Matching

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

- '?' Matches any single character.
- '\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:



```

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

```

**Tags:** Dynamic Programming, Backtracking, Greedy, String

### 16.4.12 Word Break II

Given a string `s` and a dictionary of words `dict`, add spaces in `s` to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

```

s = "catsanddog",
dict = ["cat", "cats", "and", "sand", "dog"].

```

A solution is `["cats and dog", "cat sand dog"]`.

**Tags:** Dynamic Programming Backtracking

### 16.4.13 Word Ladder II

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

For example,

```

Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
Return
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]

```

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

**Tags:** Array, Backtracking, Breadth-first Search, String

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 16.3: Sudoku Solver 1

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 16.4: Sudoku Solver 2



# Chapter 17

## Bit Manipulation

### 17.1 Easy

#### 17.1.1 Majority Element

### 17.2 Medium

#### 17.2.1 Subsets: Bit Manipulation

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

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example, If  $S = [1,2,3]$ , a solution is:

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

**Tags:** Array Backtracking, Bit Manipulation

#### 17.2.2 Single Number

#### 17.2.3 Single Number II



# Chapter 18

## 图 Graphics

### 18.1 Medium

#### 18.1.1 Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization:

Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.

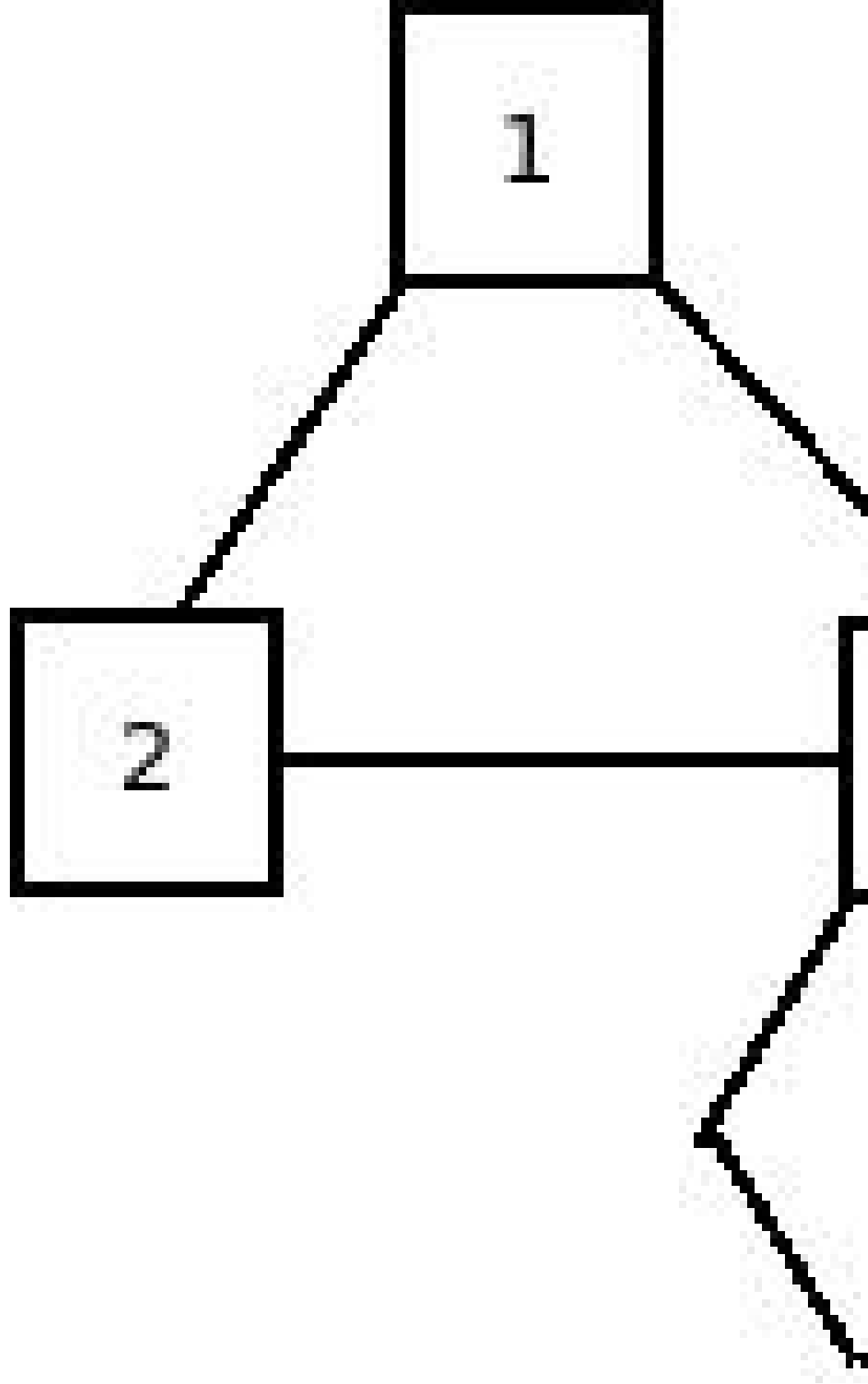
As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

1. First node is labeled as 0. Connect node 0 to both nodes 1 and 2.

2. Second node is labeled as 1. Connect node 1 to node 2.

3. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.



Visually, the graph looks like the following:

1. 分析：广度优先遍历或深度优先遍历都可以
2. DFS：时间复杂度  $O(n)$ , 空间复杂度  $O(n)$

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     List<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }
 * };
 */
public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
```



```

if (node == null) return null;
UndirectedGraphNode res = new UndirectedGraphNode(node.label); // result head
if (node.neighbors == null || node.neighbors.size() == 0) return res;

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();
Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGraphNode>();
q.add(node); // added first node, need add its all Neighbors as well
map.put(node, res);

List<UndirectedGraphNode> curNbr = new ArrayList<UndirectedGraphNode>();
UndirectedGraphNode curr = null;
while (!q.isEmpty()) {
    curr = q.poll();
    curNbr = curr.neighbors; // ori
    for (UndirectedGraphNode aNbr : curNbr) { // for build connection among copies
        if (!map.containsKey(aNbr)) {
            UndirectedGraphNode acpNbr = new UndirectedGraphNode(aNbr.label);
            map.put(aNbr, acpNbr);
            map.get(curr).neighbors.add(acpNbr);
            q.add(aNbr);
        } else
            map.get(curr).neighbors.add(map.get(aNbr));
    }
}
return res;
}

```

3. BFS:

## 18.2 Word Ladder, Word Ladder II: Backtracing

### 18.2.1 Word Ladder

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

For example,

Given: start = "hit", end = "cog"

dict = ["hot","dot","dog","lot","log"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",  
return its length 5.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

### 18.2.2 Word Ladder II

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

For example,

Given:

start = "hit", end = "cog"

dict = ["hot","dot","dog","lot","log"]

Return

```
[ "hit", "hot", "dot", "dog", "cog" ],  
[ "hit", "hot", "lot", "log", "cog" ]  
]
```

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

### 18.2.3 Check whether the graph is bigraph

1. Topological Sort Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge  $uv$ , vertex  $u$  comes before  $v$  in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0 ” . There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0 ” . The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

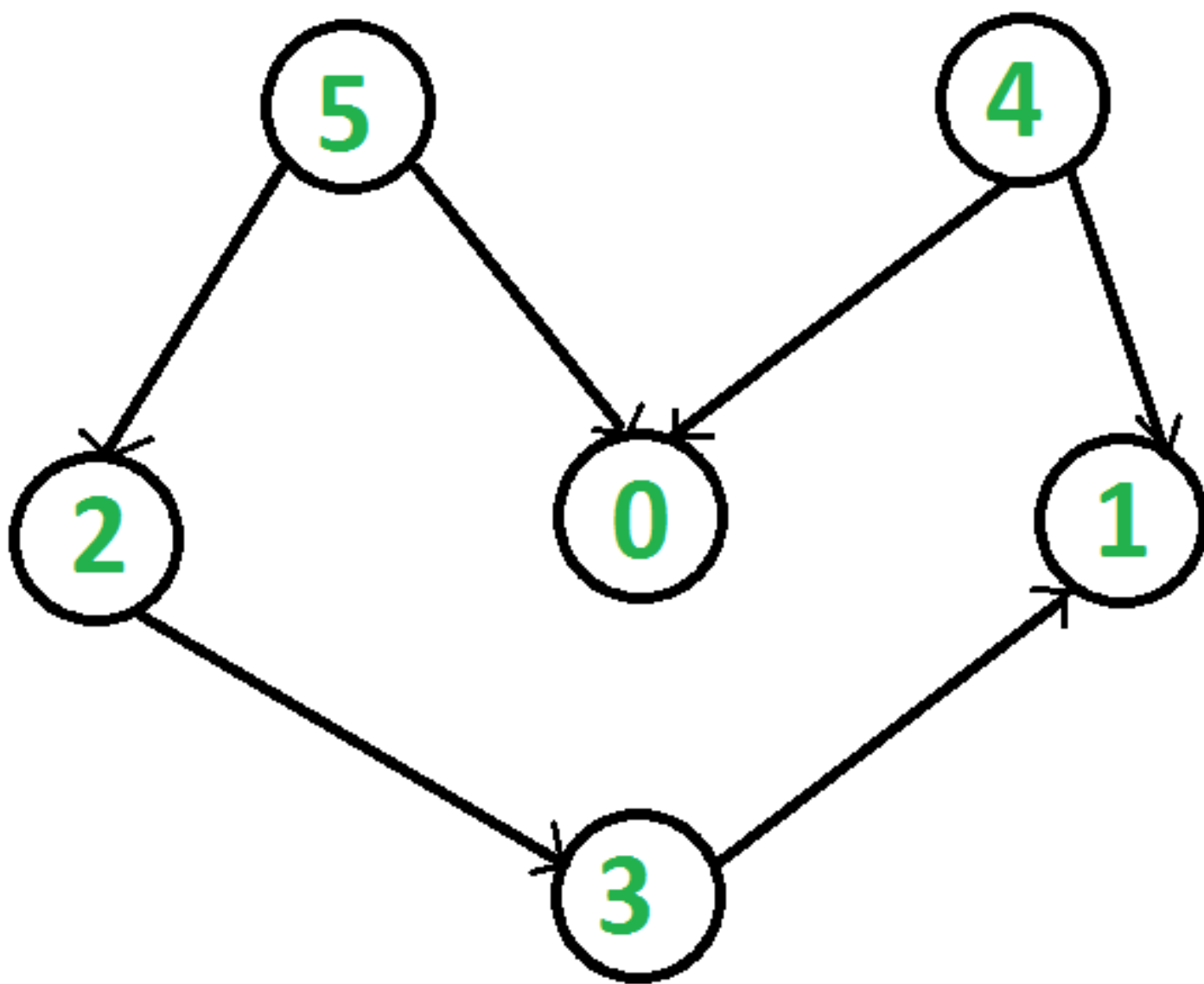


Figure 18.1: Topological Sorting

2. Topological Sorting vs Depth First Traversal (DFS): In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’ , but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’ . So Topological sorting is different from DFS. For example, a DFS of the above graph is “5 2 3 1 0 4 ” , but it is not a topological sorting.

3. Algorithm to find Topological Sorting: We recommend to first see implementation of DFS [here](#). We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.



# Chapter 19

## Data Structure

### 19.1 Easy

#### 19.1.1 Two Sum III

Design and implement a TwoSum class. It should support the following operations: add and find.

- add - Add the number to an internal data structure.
- find - Find if there exists any pair of numbers which sum is equal to the value.

For example,

```
add(1); add(3); add(5);  
find(4) -> true  
find(7) -> false
```

**Tags:** Hash Table, Data Structure

#### 19.1.2 Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

- push(x) – Push element x onto stack.
- pop() – Removes the element on top of the stack.
- top() – Get the top element.
- getMin() – Retrieve the minimum element in the stack. **Tags:** Stack Data Structure

```
public static class MinStack {  
    Stack<Integer> stack = new Stack<Integer>();  
    Stack<Integer> minStack = new Stack<Integer>();  
    public void push(int x) {  
        stack.push(x);  
        if (minStack.isEmpty() || x <= minStack.peek()) {  
            minStack.push(x);  
        }  
    }  
  
    public void pop() { // java boxing & unboxing, container, object specific methods  
        if (stack.peek().intValue() == minStack.peek().intValue())  
            minStack.pop();  
        stack.pop();  
    }  
  
    public int top() {  
        return stack.peek();  
    }  
  
    public int getMin() {  
        if (!minStack.isEmpty()) return minStack.peek();  
        else return -1;  
    }  
}
```

```
}
}
```

## 19.2 Hard

### 19.2.1 LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

- get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
- set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

```
public static class LRUCache {
    public class Node {
        int key;
        int value;
        Node prev;
        Node next;
        public Node(int x, int y){
            key = x;
            value = y;
        }
    }

    private HashMap<Integer, Node> hash;
    private int cap;
    private int number;
    Node head;
    Node tail;
    public LRUCache(int capacity) {
        cap = capacity;
        number = 0;
        head = new Node(-1, -1);
        head.prev = null;
        head.next = null;
        tail = head;
        hash = new HashMap<Integer, Node>(capacity); // so I can restrict a size !!
    }

    public int get(int key) {
        Node res = hash.get(new Integer(key)); // don't understand here
        if (res != null) {
            refresh(res); // update usage frequency
            return res.value;
        } else return -1;
        /*
        if (hash.containsKey(key)) {
            //Node res = hash.get(new Integer(key)); // don't understand here
            Node res = hash.get(key);
            refresh(res); // update usage frequency
            return res.value;
        } else {
            return -1;
        }
        */
    }

    // so still, must maintain a doubly-linked list to order usage frequency
    public void refresh(Node tmp) {
        if (tmp == head.next) return; // it's head already

        Node temp = head.next; // head node in the hash
```

```

Node prevNode = tmp.prev;
Node nextNode = tmp.next;
// set to be most recently used~~ move the tmp node to be head.next, connections
head.next = tmp;
tmp.prev = head;
tmp.next = temp;
temp.prev = tmp;
prevNode.next = nextNode;
if (nextNode != null)
    nextNode.prev = prevNode;
else tail = prevNode;    // remember tail as well
}

public void set(int key, int value) {
Node res = hash.get(new Integer(key));
if (res != null) {
    refresh(res);
    res.value = value;
} else {
    //if (!hash.containsKey(key)) { // another way of detecting existing
Node prevNode = new Node(key, value);
Node temp;
if (number == cap) { // remove tail;
    temp = tail.prev;
    hash.remove(tail.key);    // remember to remove from hash too !!!
    if (temp != null) {
        temp.next = null;
    }
    tail.prev = null;
    tail = temp;
    number--;
}
// add to tail first
tail.next = prevNode;
prevNode.prev = tail;
tail = prevNode;
refresh(prevNode);
hash.put(key, prevNode);
number++;    // count node numbers
}
}
}

```





## Chapter 20

### 细节实现题



# Chapter 21

## Math

### 21.1 Easy

21.1.1 Add Binary

21.1.2 Roman to Integers

21.1.3 String to Integer (atoi)

21.1.4 Palindrome Number

21.1.5 Plus One

21.1.6 Factorial Trailing Zeroes

21.1.7 Excel Sheet Column Title

21.1.8 Excel Sheet Column Number

21.1.9 Reverse Integer

### 21.2 Medium

21.2.1 Multiply Strings

21.2.2 Sqrt(x)

21.2.3 Divide Two Integers

21.2.4 Pow(x, n)

21.2.5 Fraction to Recurring Decimal

21.2.6 Permutation Sequence

21.2.7 Integer to Roman

21.2.8 Next Permutation: Math

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

### 21.3 Hard

21.3.1 Valid Number

21.3.2 Max Points on a Line