LeetCode Summary

deepwaterooo

January 31, 2015

Contents

1 数组 Array **5**

1.1 Easy . . . 6
- 1.1.1 Remove Duplicates from Sorted Array . . . 6
- 1.1.2 Remove Element . . . 6
- 1.1.3 Plus One . . . 6

1.2 Medium . . . 6
- 1.2.1 Rotate Image . . . 6
- 1.2.2 Set Matrix Zeroes . . . 6
- 1.2.3 Remove Duplicates from Sorted Array II . . . 6
- 1.2.4 Search in Rotated Sorted Array II . . . 6
- 1.2.5 3Sum . . . 6
- 1.2.6 3Sum Closest . . . 6
- 1.2.7 4Sum . . . 6
- 1.2.8 Remove Element . . . 6
- 1.2.9 Next Permutation . . . 6
- 1.2.10 Permutation Sequence . . . 6
- 1.2.11 Valid Sudoku . . . 6
- 1.2.12 Trapping Rain Water . . . 6
- 1.2.13 Gray Code . . . 6

1.3 Hard . . . 6
- 1.3.1 Search in Rotated Sorted Array . . . 6
- 1.3.2 Median of Two Sorted Arrays . . . 6
- 1.3.3 Longest Consecutive Sequence . . . 6
- 1.3.4 Trapping Rain Water . . . 6

1.4 others Covered . . . 6
- 1.4.1 Majority Element . . . 7
- 1.4.2 Container With Most Water . . . 7
- 1.4.3 Minimum Path Sum . . . 7
- 1.4.4 Missing Ranges . . . 8
- 1.4.5 Find Minimum in Rotated Sorted Array . . . 8
- 1.4.6 Triangle . . . 8
- 1.4.7 Find Peak Element . . . 8
- 1.4.8 Find Minimum in Rotated Sorted Array II . . . 8
- 1.4.9 Jump Game II . . . 8

2 字符串 String **9**
2.1 Easy . . . 10
2.1.1 Valid Palindrome . . . 10
2.1.2 Implement strStr() . . . 10
2.1.3 String to Integer (atoi) . . . 10
2.1.4 Add Binary . . . 10
2.1.5 Longest Common Prefix . . . 10
2.1.6 Roman to Integer . . . 10
2.1.7 Count and Say . . . 10
2.1.8 Length of Last Word . . . 10
2.2 Medium . . . 10
2.2.1 Longest Palindromic Substring . . . 10
2.2.2 Integer to Roman . . . 10
2.2.3 Anagrams . . . 10
2.2.4 Simplify Path . . . 10
2.3 Hard . . . 10
2.3.1 Valid Number . . . 10
2.3.2 Edit Distance . . . 10
2.3.3 Minimum Window Substring . . . 10
2.3.4 Longest Substring with At Most Two Distinct Characters . . . 10
2.3.5 Read N Characters Given Read4 II - Call multiple times . . . 10
2.4 other Covered . . . 10
2.4.1 Valid Parentheses . . . 11
2.4.2 Compare Version Numbers . . . 11
2.4.3 Read N Characters Given Read4 . . . 11
2.4.4 One Edit Distance . . . 11
2.4.5 Decode Ways . . . 11
2.4.6 Longest Substring Without Repeating Characters . . . 11
2.4.7 Reverse Words in a String . . . 11
2.4.8 Interleaving String . . . 11

3 Linked List **13**
3.1 Easy . . . 13
3.1.1 Remove Duplicates from Sorted List . . . 13
3.1.2 Remove Nth Node From End of List . . . 13
3.1.3 Merge Two Sorted Lists . . . 14
3.1.4 Intersection of Two Linked Lists . . . 14
3.2 Medium . . . 14
3.2.1 Add Two Numbers . . . 14
3.2.2 Reverse Linked List II . . . 14
3.2.3 Partition List . . . 15
3.2.4 Remove Duplicates from Sorted List . . . 16
3.2.5 Remove Duplicates from Sorted List II . . . 17
3.2.6 Rotate List . . . 18
3.2.7 Swap Nodes in Pairs . . . 18
3.2.8 Reorder List . . . 19
3.2.9 Convert Sorted List to Binary Search Tree . . . 20
3.2.10 Linked List Cycle . . . 20

3.2.11 Linked List Cycle II . . . 21
3.3 Hard . . . 22
3.3.1 Reverse Nodes in k-Group . . . 22
3.3.2 Copy List with Random Pointer . . . 23
3.4 other Covered . . . 24

4 树 Binary Tree, Binary Search Tree 25
4.1 Morris 专题 . . . 25
4.1.1 中序遍历 (Inorder Traversal) . . . 25
4.1.2 先序遍历 (Preorder Traversal) . . . 26
4.1.3 后序遍历 (Postorder Traversal) . . . 27
4.2 二叉树的遍历 . . . 28
4.2.1 Binary Tree Preorder Traversal . . . 28
4.2.2 Binary Tree Inorder Traversal . . . 30
4.2.3 Binary Tree Postorder Traversal . . . 31
4.2.4 Binary Tree Level Order Traversal . . . 32
4.2.5 Binary Tree Level Order Traversal II . . . 32
4.2.6 Binary Tree Zigzag Level Order Traversal . . . 33
4.2.7 Recover Binary Search Tree . . . 33
4.2.8 Same Tree . . . 34
4.2.9 Symmetric Tree . . . 34
4.2.10 Balanced Binary Tree . . . 34
4.2.11 Flatten Binary Tree to Linked List . . . 34
4.2.12 Populating Next Right Pointers in Each Node II . . . 35
4.3 二叉树的构建 . . . 36
4.3.1 Construct Binary Tree from Preorder and Inorder Traversal . . . 36
4.3.2 Construct Binary Tree from Inorder and Postorder Traversal . . . 36
4.4 二叉树查找 . . . 36
4.4.1 Validate Binary Search Tree . . . 36
4.4.2 Convert Sorted Array to Binary Search Tree . . . 36
4.4.3 Convert Sorted List to Binary Search Tree . . . 36
4.5 二叉树递归 . . . 37
4.5.1 Minimum Depth of Binary Tree . . . 37
4.5.2 Maximum Depth of Binary Tree . . . 37
4.5.3 Path Sum . . . 37
4.5.4 Path Sum II . . . 37
4.5.5 Binary Tree Maximum Path Sum . . . 38
4.5.6 Populating Next Right Pointers in Each Node . . . 38
4.5.7 Sum Root to Leaf Numbers . . . 39
4.5.8 Binary Tree Upside Down . . . 39
4.5.9 Binary Search Tree Iterator . . . 40
4.5.10 Binary Tree Maximum Path Sum . . . 40
4.6 others Covered . . . 40

5 栈 Stack 41
5.1 Easy . . . 41
5.1.1 Valid Parentheses . . . 41
5.2 Medium . . . 41

5.2.1 Evaluate Reverse Polish Notation . . . 41
5.2.2 Simplify Path . . . 41
5.3 Hard . . . 42
5.3.1 Longest Valid Parentheses . . . 42
5.3.2 Trapping Rain Water . . . 44
5.3.3 Largest Rectangle in Histogram . . . 45
5.4 others Covered . . . 47

6 Hash Table **51**
6.1 Easy . . . 52
6.1.1 Valid Sudoku . . . 52
6.1.2 Two Sum III . . . 52
6.2 Medium . . . 52
6.2.1 Two Sum . . . 52
6.2.2 4Sum . . . 52
6.2.3 Binary Tree Inorder Traversal . . . 52
6.2.4 Fraction to Recurring Decimal . . . 52
6.2.5 Single Number . . . 52
6.2.6 Anagrams . . . 52
6.2.7 Longest Substring Without Repeating Characters . . . 52
6.3 Hard . . . 52
6.3.1 Minimum Window Substring . . . 52
6.3.2 Copy List with Random Pointer . . . 52
6.3.3 Sudoku Solver . . . 52
6.3.4 Max Points on a Line . . . 52
6.3.5 Substring with Concatenation of All Words . . . 52
6.3.6 Longest Substring with At Most Two Distinct Characters . . . 52
6.4 other Covered . . . 52

7 排序 **53**
7.1 Easy . . . 53
7.1.1 Merge Sorted Array . . . 53
7.1.2 Merge Two Sorted List . . . 53
7.2 Medium . . . 54
7.2.1 Sort Colors . . . 54
7.2.2 Sort List . . . 55
7.3 Hard . . . 56
7.3.1 Merge k sorted Lists . . . 56
7.3.2 Insertion Sort List . . . 57
7.3.3 First Missing Positive . . . 57
7.4 other Covered . . . 59
7.4.1 Largest Number . . . 59
7.4.2 Maximum Gap . . . 59

8 查找 **61**
8.1 Medium . . . 61
8.1.1 Search for a Range . . . 61
8.1.2 Search Insert Position . . . 62

8.1.3 Search a 2D Matrix . . . 63

9 暴力枚举法 65
9.1 Medium . . . 65
9.1.1 Subsets . . . 65
9.1.2 Subsets II . . . 67
9.1.3 Permutation . . . 69
9.1.4 Combinationas . . . 69
9.1.5 Letter Combinationas of Phone Number . . . 70
9.2 Hard . . . 72
9.2.1 Permutation II . . . 72

10 广度优先搜索 Breadth First Search 75
10.1 Medium . . . 75
10.1.1 Surrounded Regions . . . 75
10.1.2 Word Ladder . . . 76
10.2 Hard . . . 78
10.2.1 Word Ladder II . . . 78
10.3 other Covered . . . 81
10.4 小结 . . . 81
10.4.1 适用场景 . . . 81
10.4.2 思考的步骤 . . . 81

11 深度优先搜索 Depth First Search 83
11.1 Medium . . . 83
11.1.1 Palindrome Partitioning . . . 83
11.1.2 Restore IP Addresses . . . 84
11.1.3 Generate Parentheses . . . 84
11.1.4 Combination Sum . . . 85
11.1.5 Combination Sum II . . . 86
11.2 Hard . . . 87
11.2.1 N-Queens . . . 87
11.2.2 N-Queens II . . . 88
11.2.3 Sudoku Solver . . . 88
11.2.4 Word Search . . . 90
11.3 other Covered . . . 91
11.4 小结 . . . 92
11.4.1 适用场景 . . . 92
11.4.2 思考的步骤 . . . 92
11.4.3 代码模板 . . . 93

12 分治法 Divide and Conqure 97
12.1 Medium . . . 97
12.1.1 Pow(x, n) . . . 97
12.1.2 Sqrt(x) . . . 97
12.2 other Covered . . . 98

13 贪心法 Greedy Search **99**
13.1 Medium . . . 99
13.1.1 Gas Station . . . 99
13.1.2 Best Time to Buy and Sell Stock . . . 99
13.1.3 Best Time to Buy and Sell Stock II . . . 99
13.1.4 Jump Game . . . 100
13.2 Hard . . . 101
13.2.1 Jump Game II . . . 101
13.2.2 Candy . . . 101
13.3 other Covered . . . 101

14 动态归划 Dynamic Programming **103**
14.1 Easy . . . 103
14.1.1 Climbing Stairs . . . 103
14.2 Medium . . . 104
14.2.1 Unique Path . . . 104
14.2.2 Unique Path with Obstacles . . . 107
14.2.3 Minimum Path Sum . . . 109
14.2.4 Unique Binary Search Tree . . . 110
14.2.5 Unique Binary Search Tree II . . . 111
14.2.6 Maximum Sum Subarray . . . 112
14.2.7 Maximum Product Subarray . . . 113
14.2.8 Decode Ways, bug~ !! . . . 113
14.2.9 Triangle . . . 115
14.2.10 Word Break . . . 116
14.3 Hard . . . 117
14.3.1 Best Time to Buy and Sell Stock III . . . 117
14.3.2 Coins in a Line . . . 118
14.3.3 Maximal Rectangle . . . 118
14.3.4 Edit Distance . . . 120
14.3.5 Distinct Subsequence . . . 121
14.3.6 Palindrome Partitioning II . . . 122
14.3.7 Interleaving String . . . 123
14.3.8 Scramble String . . . 125
14.3.9 Regular Expression Matching . . . 126
14.3.10 Wildcard Matching . . . 129
14.4 others Covered . . . 131

15 Two Pointers and Sliding Window **133**
15.1 Easy . . . 134
15.1.1 Valid Palindrome . . . 134
15.1.2 Remove Nth Node From End of List . . . 134
15.1.3 Remove Element . . . 134
15.1.4 Remove Duplicates from Sorted Array . . . 134
15.1.5 Merge Sorted Array . . . 134
15.1.6 Implement strStr() . . . 134
15.2 Medium . . . 134
15.2.1 3Sum . . . 134

15.2.2 4Sum . . . 134
15.2.3 Container With Most Water . . . 134
15.2.4 Remove Duplicates from Sorted Array II . . . 134
15.2.5 Partition List . . . 134
15.2.6 Two Sum II - Input array is sorted . . . 134
15.2.7 Linked List Cycle II . . . 134
15.2.8 Longest Substring Without Repeating Characters . . . 134
15.2.9 3Sum Closest . . . 134
15.2.10 Linked List Cycle . . . 134
15.2.11 Sort Colors . . . 134
15.2.12 Rotate List . . . 134
15.3 Hard . . . 134
15.3.1 Trapping Rain Water . . . 134
15.3.2 Longest Substring with At Most Two Distinct Characters . . . 134
15.3.3 Substring with Concatenation of All Words . . . 134
15.3.4 Minimum Window Substring . . . 134

16 Backtracing and Recursion **135**
16.1 Medium . . . 135
16.1.1 Permutation Sequence . . . 135
16.1.2 Gray Code . . . 135
16.2 Hard . . . 137
16.2.1 Word Break II . . . 137
16.3 other Covered . . . 138

17 Bit Manipulation **139**
17.1 Easy . . . 139
17.1.1 Majority Element . . . 139
17.2 Medium . . . 139
17.2.1 Subsets: Bit Manipulation . . . 139
17.2.2 Single Number . . . 140
17.2.3 Single Number II . . . 140

18 图 Graphics **141**
18.1 Medium . . . 141
18.1.1 Clone Graph . . . 141
18.1.2 Check whether the graph is bigraph . . . 143
18.2 other Covered . . . 144

19 Data Structure **145**
19.1 Easy . . . 145
19.1.1 Two Sum III . . . 145
19.1.2 Min Stack . . . 145
19.2 Hard . . . 146
19.2.1 LRU Cache . . . 146

20 细节实现题 **149**
20.1 Medium . . . 149
20.1.1 Pascal's Triangle . . . 149
20.1.2 Pascal's Triangle II . . . 149
20.1.3 Two Sum . . . 149
20.1.4 Two Sum II - Input array is sorted . . . 149
20.1.5 Insert Interval . . . 149
20.1.6 Merge Intervals . . . 149
20.1.7 Spiral Matrix . . . 149
20.1.8 Spiral Matrix II . . . 149
20.1.9 Multiply Strings . . . 149
20.1.10 Substring with Concatenation of All Words . . . 149
20.1.11 ZigZag Conversion . . . 149
20.1.12 Text Justification . . . 149
20.2 other Covered . . . 149

21 Math **151**
21.1 Easy . . . 151
21.1.1 Add Binary . . . 151
21.1.2 String to Integer (atoi) . . . 151
21.1.3 Palindrome Number . . . 151
21.1.4 Factorial Trailing Zeroes . . . 151
21.1.5 Excel Sheet Column Title . . . 151
21.1.6 Excel Sheet Column Number . . . 151
21.1.7 Reverse Integer . . . 151
21.2 Medium . . . 151
21.2.1 Multiply Strings . . . 151
21.2.2 Divide Two Integers . . . 151
21.2.3 Fraction to Recurring Decimal . . . 151
21.2.4 Permutation Sequence . . . 151
21.2.5 Next Permutation . . . 151
21.3 Hard . . . 152
21.3.1 Valid Number . . . 152
21.3.2 Max Points on a Line . . . 152
21.4 other Covered . . . 152

Chapter 1

数组 Array

1.1 Easy

1.1.1 Remove Duplicates from Sorted Array

1.1.2 Remove Element

1.1.3 Plus One

1.2 Medium

1.2.1 Rotate Image

1.2.2 Set Matrix Zeroes

1.2.3 Remove Duplicates from Sorted Array II

1.2.4 Search in Rotated Sorted Array II

1.2.5 3Sum

1.2.6 3Sum Closest

1.2.7 4Sum

1.2.8 Remove Element

1.2.9 Next Permutation

1.2.10 Permutation Sequence

1.2.11 Valid Sudoku

1.2.12 Trapping Rain Water

1.2.13 Gray Code

1.3 Hard

1.3.1 Search in Rotated Sorted Array

1.3.2 Median of Two Sorted Arrays

- Construct Binary Tree from Preorder and Inorder Traversal 4.3.1
- Best Time to Buy and Sell Stock 13.1.2
- Best Time to Buy and Sell Stock II 13.1.3
- Largest Rectangle in Histogram 5.3.3
- Maximal Rectangle `Maximum Rectangle`
- Pascal's Triangle
- Pascal's Triangle II
- Merge Sorted Array 7.1.1
- Two Sum 6.2.1
- Two Sum II - Input array is sorted 15.2.6
- Spiral Matrix 20.1.7
- Spiral Matrix II 20.1.8
- Sort Colors 7.2.1
- Insert Interval 20.1.5
- Merge Intervals 20.1.6

1.4.1 Majority Element

1.4.2 Container With Most Water

1.4.3 Minimum Path Sum

- Search Insert Position 8.1.2
- Unique Paths
- Search for a Range 8.1.1
- Search a 2D Matrix 8.1.3
- Unique Paths II

1.4.4 Missing Ranges

1.4.5 Find Minimum in Rotated Sorted Array

1.4.6 Triangle

- Maximum Subarray `Maximum Subarray`
- Maximum Product Subarray 14.2.7
- Word Search 11.2.4
- Combination Sum 11.1.4
- Combination Sum II 11.1.5
- Subsets 9.1.1
- Subsets II 9.1.2
- Jump Game 13.1.4

1.4.7 Find Peak Element

1.4.8 Find Minimum in Rotated Sorted Array II

1.4.9 Jump Game II

- First Missing Positive 7.3.3
- Word Ladder II

10.2.1

- Best Time to Buy and Sell Stock III 14.3.1
- Roman to Integers `Roman to Integers`
- Integer to Roman 2.2.2
- Pow(x, n) 12.1.1
- Sqrt(x) 12.1.2

Chapter 2

字符串 String

2.1 Easy

2.1.1 Valid Palindrome

2.1.2 Implement strStr()

2.1.3 String to Integer (atoi)

2.1.4 Add Binary

2.1.5 Longest Common Prefix

2.1.6 Roman to Integer

2.1.7 Count and Say

2.1.8 Length of Last Word

2.2 Medium

2.2.1 Longest Palindromic Substring

2.2.2 Integer to Roman

2.2.3 Anagrams

2.2.4 Simplify Path

2.3 Hard

2.3.1 Valid Number

2.3.2 Edit Distance

2.3.3 Minimum Window Substring

2.3.4 Longest Substring with At Most Two Distinct Characters

2.3.5 Read N Characters Given Read4 II - Call multiple times

2.4 other Covered

2.4.1 Valid Parentheses

2.4.2 Compare Version Numbers

2.4.3 Read N Characters Given Read4

2.4.4 One Edit Distance

2.4.5 Decode Ways

2.4.6 Longest Substring Without Repeating Characters

2.4.7 Reverse Words in a String

2.4.8 Interleaving String

- Multiply Strings
- Substring with Concatenation of All Words
- ZigZag Conversion
- Text Justification
- Restore IP Addresses
- Letter Combinations of a Phone Number
- Generate Parentheses
- Regular Expression Matching
- Wildcard Matching
- Scramble String
- Distinct Subsequences
- Word Ladder II 10.2.1

Chapter 3

Linked List

3.1 Easy

3.1.1 Remove Duplicates from Sorted List

3.1.2 Remove Nth Node From End of List

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

For example,

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

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

Given n will always be valid.

Try to do this in one pass.

Tags: Linked List, Two Pointers

设两个指针 p, q, 让 q 先走 n 步, 然后 p 和 q 一起走, 直到 q 走到尾节点, 删除 p->next 即可。

时间复杂度 O(n), 空间复杂度 O(1).

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(-1);
    ListNode p = dummy;
    ListNode q = dummy;
    p.next = head;
    q.next = head;
    for (int i = 0; i < n; i++) {    // q 先走 n 步
        q = q.next;
    }
    while (q.next != null) {              // 一起走
        p = p.next;
        q = q.next;
    }
    ListNode tmp = p.next;
    p.next = tmp.next;
    return dummy.next;
}
```

3.1.3 Merge Two Sorted Lists

3.1.4 Intersection of Two Linked Lists

3.2 Medium

3.2.1 Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

跟 Add Binary(见 §3.4) 很类似.

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(-1);   // dummy 头节点
    int carry = 0;
    ListNode prev = head;
    for (ListNode pa = l1, pb = l2;
         pa != null || pb != null;
         pa = pa == null ? null : pa.next,
             pb = pb == null ? null : pb.next,
             prev = prev.next) {
        int aval = pa == null ? 0 : pa.val;
        int bval = pb == null ? 0 : pb.val;
        int value = (aval + bval + carry) % 10;
        carry = (aval + bval + carry) / 10;
        prev.next = new ListNode(value); // 尾插法
    }
    if (carry > 0)
        prev.next = new ListNode(carry);
    return head.next;
}
```

3.2.2 Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:

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

return 1->4->3->2->5->NULL.

Note:

Given m, n satisfy the following condition:

$1 \leqslant m \leqslant n \leqslant$ length of list.

这题非常繁琐, 有很多边界检查,15 分钟内做到 bug free 很有难度!

1. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode reverseBetween(ListNode head, int m, int n) {
    ListNode dummy = new ListNode(-1);
    dummy.next = head;
    ListNode prevM = null;
```

```
        ListNode curr = dummy;
        for (int i = 1; i <= n ; i++) {
            if (i == m) prevM = curr;
            if (i > m && i <= n) { // prev_m 和 curr 静止不动
                curr.next = head.next;
                head.next = prevM.next;
                prevM.next = head;
                head = curr;
            }
            curr = head;
            head = head.next;
        }
        return dummy.next;
    }
```

再仔细好好想一下这个代码 ~

3.2.3 Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given 1->4->3->2->5->2 and x = 3,

return 1->2->2->4->3->5.

Tags: Linked List, Two Pointers

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
    public ListNode partition(ListNode head, int x) {
        if (head == null) return head;
        ListNode lftDummy = new ListNode(0);
        ListNode ritDummy = new ListNode(0);
        ListNode lftCurr = lftDummy; // 左头结点
        ListNode ritCurr = ritDummy; // 右头结点

        for (; head != null; head = head.next) {
            if (head.val < x) {
                lftCurr.next = head;
                lftCurr = head;
            } else {
                ritCurr.next = head;
                ritCurr = head;
            }
        }
        lftCurr.next = ritDummy.next;
        ritCurr.next = null; // 无限寻还
        return lftDummy.next;
    }
```

3.2.4 Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.
For example,
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(1)

c++ 版:

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (!head) return head;
        ListNode dummy{head->val + 1, head}; // 值只要跟 head 不同即可
        recur(&dummy, head);
        return dummy.next;
    }
private:
    static void recur(ListNode *prev, ListNode *cur) {
        if (cur == nullptr) return;
        if (prev->val == cur->val) { // 删除 head
            prev->next = cur->next;
            delete cur;
            recur(prev, prev->next);
        } else {
            recur(prev->next, cur->next);
        }
    }
};
```

Java 版:

```
public void recursion(ListNode prev, ListNode curr) {
    if (curr == null) return;
    if (prev.val == curr.val) {     // 删除 head
        prev.next = curr.next;
        recursion(prev, curr.next);
    } else {
        prev.next = curr;   // important for Java
        recursion(prev.next, curr.next);
    }
}

public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return head;
    ListNode dummy = new ListNode(head.val + 1);
    // 值只要跟 head 不同即可
    recursion(dummy, head);
    return dummy.next;
}
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return head;
    for (ListNode prev = head, curr = head.next; curr != null; curr = curr.next) {
        if (prev.val == curr.val) {
            prev.next = curr.next;
        } else {
            prev = curr;
        }
    }
    return head;
}
```

虽然觉得自己链表和树的原理掌握得还不错，可这两个版块都因为代码没有受过正规训练，所以代码奇差无比，看看别人写的。。。照葫芦画饼地写一遍，也算是一个提高的过程吧。。。

3.2.5 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.

For example,
Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode deleteDuplicates(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode curr = head.next;
    if (head.val == curr.val) {
        while (curr != null && head.val == curr.val)
            curr = curr.next;
        return deleteDuplicates(curr);
    } else {
        head.next = deleteDuplicates(head.next);
        return head;
    }
}
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return head;
    ListNode dummy = new ListNode(Integer.MIN_VALUE);
    dummy.next = head;
    ListNode prev = dummy;
    ListNode curr = head;
    while (curr != null) {
        boolean dup = false;
        while (curr.next != null && curr.val == curr.next.val) {
            dup = true;
```

```
                curr = curr.next;
            }
            if (dup) {     // 删除重复的最后一个元素
                curr = curr.next;
                continue;
            }
            prev.next = curr;
            prev = prev.next;
            curr = curr.next;
        }
        prev.next = curr;
        return dummy.next;
    }
```

3.2.6 Rotate List

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

For example:

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

return 4->5->1->2->3->NULL.

Tags: Linked List, Two Pointers

先遍历一遍, 得出链表长度 len, 注意 k 可能大于 len, 因此令 k% = len。将尾节点 next 指针指向首节点, 形成一个环, 接着往后跑 len k 步, 从这里断开, 就是要求的结果了。(这个思路真是好。。。)

```
public ListNode rotateRight(ListNode head, int n) {
    if (head == null || n == 0) return head;
    int len = 1;
    ListNode curr = head;
    while (curr.next != null) {     // 求长度
        len++;
        curr = curr.next;
    }
    n = len - n % len;
    curr.next = head;              // 首尾相连
    for (int step = 0; step < n; step++) {
        curr = curr.next;       // 接着往后跑
    }
    head = curr.next;           // 新的首节点
    curr.next = null;              // 断开环
    return head;
}
```

3.2.7 Swap Nodes in Pairs

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

For example,

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

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

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode swapPairs(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode dummy = new ListNode(-1);
    dummy.next = head;
    for (ListNode prev = dummy, curr = prev.next, next = curr.next;
         next != null;
         prev = curr, curr = curr.next, next = curr != null ? curr.next : null) {
        prev.next = next;
        curr.next = next.next;
        next.next = curr;
    }
    return dummy.next;
}
```

2. 简洁版：时间复杂度 O(n), 空间复杂度 O(1)

   下面这种写法更简洁, 但题目规定了不准这样做。

```
public ListNode swapPairs(ListNode head) {
    ListNode p = head;
    int tmp = 0;
    while (p != null && p.next != null) {
        tmp = p.val;
        p.val = p.next.val;
        p.next.val = tmp;
        p = p.next.next;
    }
    return head;
}
```

3.2.8 Reorder List

Given a singly linked list L: $L_{0\rightarrow L1\rightarrow\ldots\rightarrow Ln-1}\rightarrow L_n$,
reorder it to: $L_{0\rightarrow Ln\rightarrow L1\rightarrow Ln-1}\rightarrow L_{2\rightarrow Ln-2}\rightarrow\cdots$
You must do this in-place without altering the nodes' values.
For example,
Given {1, 2, 3, 4}, reorder it to {1, 4, 2, 3}.
题目规定要 in-place, 也就是说只能使用 O(1) 的空间。
可以找到中间节点, 断开, 把后半截单链表 reverse 一下, 再合并两个单链表。
时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode reverse(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode prev = head;
    for (ListNode curr = head.next, next = curr.next;
         curr != null;
         prev = curr, curr = next, next = next != null ? next.next : null) {
        curr.next = prev;
    }
    head.next = null;
```

```
    return prev;
}

public void reorderList(ListNode head) {
    if (head == null || head.next == null) return;
    ListNode slow = head;
    ListNode fast = head;
    ListNode prev = null;
    while (fast != null && fast.next != null) {
        prev = slow;
        slow = slow.next;
        fast = fast.next.next;
    }
    prev.next = null;      // cut at middle
    slow = reverse(slow);
    ListNode curr = head;
    while (curr.next != null) {
        ListNode tmp = curr.next;
        curr.next = slow;
        slow = slow.next;
        curr.next.next = tmp;
        curr = tmp;
    }
    curr.next = slow;
}
```

3.2.9 Convert Sorted List to Binary Search Tree

3.2.10 Linked List Cycle

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

Follow up:

Can you solve it without using extra space?

Tags: Linked List，Two Pointers

最容易想到的方法是, 用一个哈希表 unordered$_{map}$<int, bool> visited, 记录每个元素是否被访问过, 一旦出现某个元素被重复访问, 说明存在环。空间复杂度 O(n), 时间复杂度 O(N)。

最好的方法是时间复杂度 O(n), 空间复杂度 O(1) 的。设置两个指针, 一个快一个慢, 快的指针每次走两步, 慢的指针每次走一步, 如果快指针和慢指针相遇, 则说明有环。参考 http://leetcode.com/2010/09/detecting-loop-in-singly-linked-list.html

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
public boolean hasCycle(ListNode head) {
    // 设置两个指针一个快一个,
    ListNode fast = head;
    ListNode slow = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
```

```
        return false;
    }
```

3.2.11 Linked List Cycle II

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

Follow up:

Can you solve it without using extra space?

Tags: Linked List, Two Pointers

当 fast 与 slow 相遇时, slow 肯定没有遍历完链表, 而 fast 已经在环内循环了 n 圈 ($1 \leqslant n$)。假设 slow 走了 s 步, 则 fast 走了 2s 步 (fast 步数还等于 s 加上在环上多转的 n 圈), 设环长为 r, 则:

$2s = s + nr$

$s = nr$

设整个链表长 L, 环入口点与相遇点距离为 a, 起点到环入口点的距离为 x, 则

$x + a = nr = (n-1)r + r = (n - 1)r + L - x$

$x = (n - 1)r + (L - x - a)$

L-x-a 为相遇点到环入口点的距离, 由此可知, 从链表头到环入口点等于 n - 1 圈内环 + 相遇点到环入口点, 于是我们可以从 head 开始另设一个指针 slow2, 两个慢指针每次前进一步, 它俩一定会在环入口点相遇。

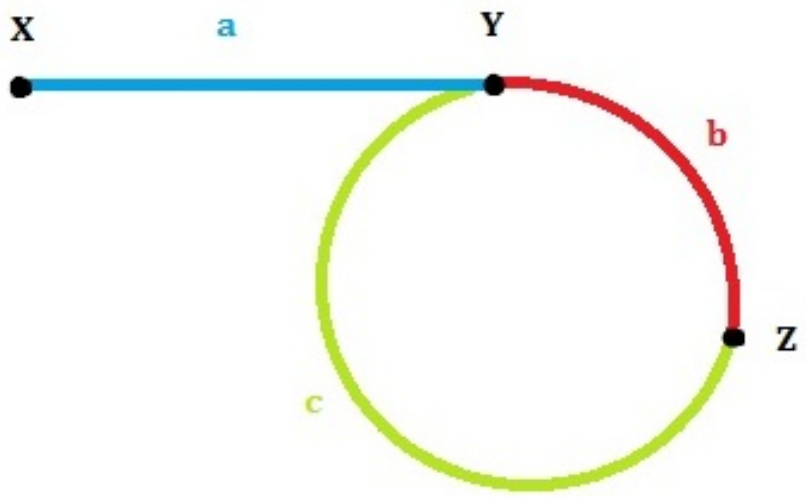


Figure 3.1: Linked List Cycle II

1. 方法一（网上都是这个答案）：

   第一次相遇后，让 slow,fast 继续走，记录到下次相遇时循环了几次。因为当 fast 第二次到达 Z 点时，fast 走了一圈，slow 走了半圈，而当 fast 第三次到达 Z 点时，fast 走了两圈，slow 走了一圈，正好还在 Z 点相遇。

   方法二：

   第一次相遇后，让 fast 停着不走了，slow 继续走，记录到下次相遇时循环了几次。

   方法三（最简单）：

第一次相遇时 slow 走过的距离：a+b，fast 走过的距离：a+b+c+b。

因为 fast 的速度是 slow 的两倍，所以 fast 走的距离是 slow 的两倍，有 2(a+b) = a+b+c+b，可以得到 a=c（这个结论很重要!）。

我们发现 L=b+c=a+b，也就是说，从一开始到二者第一次相遇，循环的次数就等于环的长度。

2. 我们已经得到了结论 a=c，那么让两个指针分别从 X 和 Z 开始走，每次走一步，那么正好会在 Y 相遇！也就是环的第一个节点。

3. 在上一个问题的最后，将 c 段中 Y 点之前的那个节点与 Y 的链接切断即可。

4. 如何判断两个单链表是否有交点？先判断两个链表是否有环，如果一个有环一个没环，肯定不相交；如果两个都没有环，判断两个列表的尾部是否相等；如果两个都有环，判断一个链表上的 Z 点是否在另一个链表上。

   如何找到第一个相交的节点？求出两个链表的长度 L1,L2（如果有环，则将 Y 点当做尾节点来算），假设 L1<L2，用两个指针分别从两个链表的头部开始走，长度为 L2 的链表先走 (L2-L1) 步，然后两个一起走，直到二者相遇。

   时间复杂度 O(n), 空间复杂度 O(1).

```
public ListNode detectCycle(ListNode head) {
    ListNode fast = head;
    ListNode slow = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next; // 2X speed
        if (fast == slow) {
            ListNode curr = head;
            while (curr != slow) {
                curr = curr.next;
                slow = slow.next;
            }
            return curr;
        }
    }
    return null;
}
```

3.3 Hard

3.3.1 Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.
If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.
You may not alter the values in the nodes, only nodes itself may be changed.
Only constant memory is allowed.
For example,
Given this linked list: 1->2->3->4->5
For k = 2, you should return: 2->1->4->3->5
For k = 3, you should return: 3->2->1->4->5

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(1)

```
public ListNode reverseKGroup(ListNode head, int k) {
    if (head == null || head.next == null || k < 2) return head;
    ListNode gNext = head;
    for (int i = 0; i < k; i++) {
        if (gNext != null) {
            gNext = gNext.next;
        } else return head;
    }
    // gNext is the head of next group
    // newGNext is the new head of next group after reversion
    ListNode newGNext = reverseKGroup(gNext, k);
    ListNode prev = null;
    ListNode curr = head;
    while (curr != gNext) {
        ListNode next = curr.next;
        curr.next = prev != null ? prev : newGNext;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```
bug~~
```

3.3.2 Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

Tags: Hash Table, Linked List

```
public RandomListNode copyRandomList(RandomListNode head) {
    if (head == null) return null;
    for (RandomListNode curr = head; curr != null; ) {
        RandomListNode node = new RandomListNode(curr.label);
        node.next = curr.next;
        curr.next = node;
        curr = node.next;
    }
    for (RandomListNode curr = head; curr != null; ) {
        if (curr.random != null)
            curr.next.random = curr.random.next;
        curr = curr.next.next;
    }
    // 分拆两个单链表
    RandomListNode dummy = new RandomListNode(-1);
    for (RandomListNode curr = head, newC = dummy; curr != null; ) {
        newC.next = curr.next;
```

```
            newC = newC.next;
            curr.next = curr.next.next;
            curr = curr.next;
        }
        return dummy.next;
    }
```

改天去找一下 HashMap 的解法，或者自己写一个。

3.4 other Covered

- Sort List 7.2.2
- Insertion Sort List 7.3.2
- Merge k Sorted Lists `Merge k Sorted Lists`

Chapter 4

树 Binary Tree, Binary Search Tree

4.1 Morris 专题

http://blog.ygui.me/algorithms/datastructure/2014-09/166

二叉树的 Morris 遍历可不使用用栈，在常量空间 O(1)、线性时间 O(n) 内实现二叉树的前中后序遍历，且遍历后不破坏二叉树的形状（中间过程允许改变其形状）。

Morris 遍历的基本原理是利用所有叶子结点的 right 指针，指向其后继结点，组成一个环，在第二次遍历到这个结点时，由于其左子树已经遍历完了，则访问该结点。

4.1.1 中序遍历 (Inorder Traversal)

中序遍历的算法步骤如下：

1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。
2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

   A. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

   B. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状）。输出当前节点。当前节点更新为当前节点的右孩子。
3. 重复以上 1、2 直到当前节点为空。

下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。

```
vector inorderTraversal(TreeNode *root) {
    vector order;
    for(TreeNode *now = root, *tmp; now;) {
        if (now->left == NULL) {
                order.push_back(now->val);
                now = now->right;
        } else {
                for (tmp = now->left;
                     tmp->right != NULL && tmp->right != now;)
                    tmp = tmp->right;
                if (tmp->right) {
```

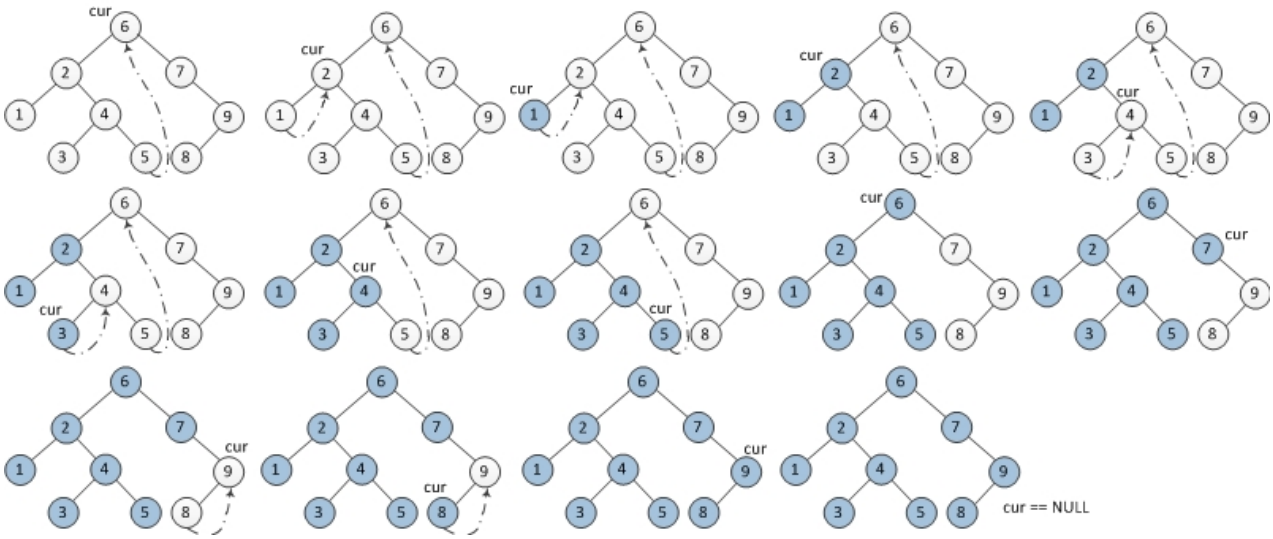


Figure 4.1: Morris 中序遍历

```
                    order.push_back(now->val);
                    tmp->right = NULL;
                    now = now->right;
                } else {
                    tmp->right = now;
                    now = now->left;
                }
            }
        }
        return order;
}
```

4.1.2 先序遍历 (Preorder Traversal)

先序遍历与中序遍历相似，代码上只有一行不同，不同就在于输出的顺序。

1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。
2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。
   A. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。输出当前节点并将当前节点更新为当前节点的左孩子。

   B. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。当前节点更新为当前节点的右孩子。
3. 重复以上 1、2 直到当前节点为空。

   下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。

```
vector preorderTraversal(TreeNode *root) {
        vector order;
        for (TreeNode *now = root, *tmp; now;) {
                if (now->left =  = NULL) {
```

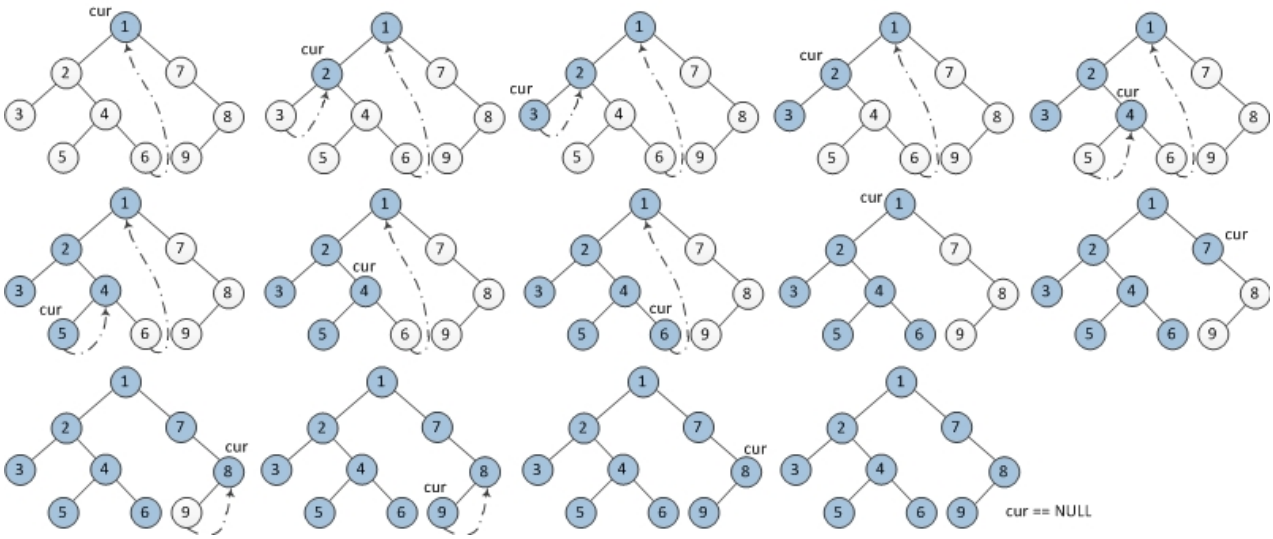


Figure 4.2: Morris 先序遍历

```
                order.push_back(now->val);
                now = now->right;
            } else {
                for (tmp = now->left; tmp->right! = NULL && tmp->right! = now;) {
                    tmp = tmp->right;
                if (tmp->right) {
                                        tmp->right = NULL;
                                        now = now->right;
                } else {
                                        order.push_back(now->val);
                                        tmp->right = now;
                                        now = now->left;
                                }
                }
            }
            return order;
    }
```

4.1.3 后序遍历 (Postorder Traversal)

后序遍历比较复杂，它的思路是利用中序遍历，所以首先产生了一个假的根结点，其左子树为原来的二叉树，从假的根结点开始中序遍历它和中序遍历有所不同，在发现当前结点左子树为空时，不访问此结点 (后序遍历需要保证访问完右子树后才能访问根结点)，直接访问右子树。第二次遍历到某个结点时，将该结点左子树的最右路径反序输出即可。具体步骤如下：

1. 当前节点设置为临时节点 dump。

2. 如果当前节点的左孩子为空，则将其右孩子作为当前节点。

3. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

A. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

B. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。倒序输出从当前节点的左孩子到该前驱节点这条路径上的所有节点。当前节点更新为当前节点的右孩子。

4. 重复以上 2、3 直到当前节点为空。

下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。

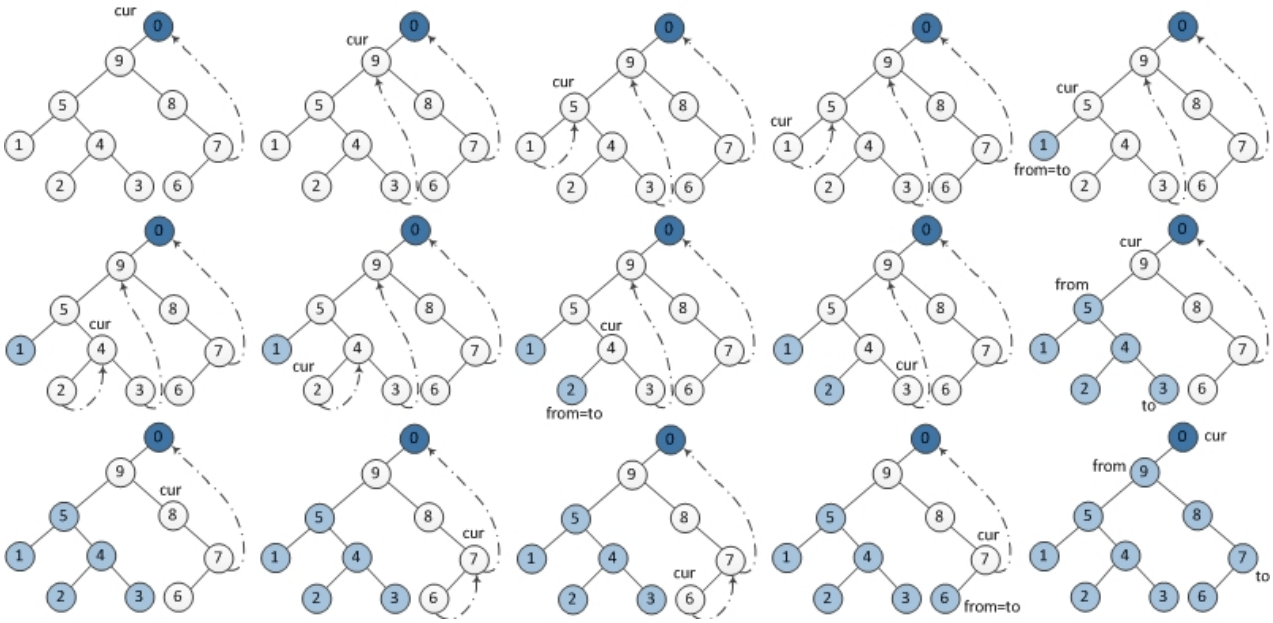


Figure 4.3: Morris 后序遍历

4.2 二叉树的遍历

树的遍历有两类: 深度优先遍历和宽度优先遍历。深度优先遍历又可分为两种: 先根 (次序) 遍历和后根 (次序) 遍历。

树的先根遍历是: 先访问树的根结点, 然后依次先根遍历根的各棵子树。树的先跟遍历的结果与对应二叉树 (孩子兄弟表示法) 的先序遍历的结果相同。

树的后根遍历是: 先依次后根遍历树根的各棵子树, 然后访问根结点。树的后跟遍历的结果与对应二叉树的中序遍历的结果相同。

二叉树的先根遍历有: 先序遍历 (root -> left -> right), root -> right -> left; 后根遍历有: 后序遍历 (left -> right -> root), right -> left -> root; 二叉树还有个一般的树没有的遍历次序, 中序遍历 (left -> root -> right)。

4.2.1 Binary Tree Preorder Traversal

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

For example:

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

```
1
 \
  2
 /
3
```

return [1,2,3].
Note: Recursive solution is trivial, could you do it iteratively?
用栈或 Morris 遍历

1. 栈：使用栈, 时间复杂度 O(n), 空间复杂度 O(n)

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    if (root == null) return res;
    Stack<TreeNode> s = new Stack<TreeNode>();
    TreeNode curr = root;
    if (curr != null) s.push(curr);
    while (!s.isEmpty()) {
        curr = s.pop();
        res.add(curr.val);
        if (curr.right != null) s.push(curr.right);
        if (curr.left != null) s.push(curr.left);
    }
    return res;
}
```

2. Morris 先序遍历：Morris 先序遍历, 时间复杂度 O(n), 空间复杂度 O(1)

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    TreeNode curr = root;
    TreeNode prev = null;
    while (curr != null) {
        if (curr.left == null) {
            res.add(curr.val);
            prev = curr;       /* cur 刚刚被访问过 */
            curr = curr.right;
        } else {                       /* 查找前驱 */
            TreeNode node = curr.left;
            while (node.right != null && node.right != curr) {
                node = node.right;
            }
            if (node.right == null) {    /* 还没线索化则建立线索, */
                res.add(curr.val);     /* 仅这一行的位置与中序不同 */
                node.right = curr;
                prev = curr;         /* cur 刚刚被访问过 */
                curr = curr.left;
            } else {             /* 已经线索化则删除线索, */
                node.right = null;
                /* prev = cur; 不能有这句,cur 已经被访问 */
                curr = curr.right;
```

```
                }
            }
        }
        return res;
    }
```

4.2.2 Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
1
 \
  2
 /
3
```

return [1,3,2].
Note: Recursive solution is trivial, could you do it iteratively?

1. 栈：使用栈, 时间复杂度 O(n), 空间复杂度 O(n)

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    TreeNode p = root;
    Stack<TreeNode> s = new Stack<TreeNode>();
    while (!s.isEmpty() || p != null) {
        if (p != null) {
            s.push(p);
            p = p.left;
        } else {
            p = s.pop();
            res.add(p.val);
            p = p.right;
        }
    }
    return res;
}
```

2. Morris 先序遍历：Morris 先序遍历, 时间复杂度 O(n), 空间复杂度 O(1)

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    TreeNode prev = null;
    TreeNode curr = root;
    Stack<TreeNode> s = new Stack<TreeNode>();
    while (curr != null) {
        if (curr.left == null) {
            res.add(curr.val);
            prev = curr;
            curr = curr.right;
```

```
            } else {
                /* 查找前驱 */
                TreeNode node = curr.left;
                while (node.right != null && node.right != curr) {
                    node = node.right;
                }
                if (node.right == null) {  /* 还没线索化则建立线索, */
                    node.right = curr;
                    /* prev = cur; 不能有这句,cur 还没有被访问 */
                    curr = curr.left;
                } else {          /* 已经线索化则访问节点并删除线索,, */
                    res.add(curr.val);
                    node.right = null;
                    curr = curr.right;
                }
            }
        }
        return res;
    }
```

4.2.3 Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
1
 \
  2
 /
3
```

return [3,2,1].
Note: Recursive solution is trivial, could you do it iteratively?

1. 栈：使用栈, 时间复杂度 O(n), 空间复杂度 O(n)

```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    /* curr, 正在访问的结点; prev刚刚访问过的结点, */
    TreeNode prev = null;
    TreeNode curr = root;
    Stack<TreeNode> s = new Stack<TreeNode>();
    do {
        while (curr != null) {      /* 往左下走 */
            s.push(curr);
            curr = curr.left;
        }
        prev = null;
        while (!s.isEmpty()) {
            curr = s.pop();
            /* 右孩子不存在或已被访问访问之, */
```

```
                if (curr.right == prev) {
                    res.add(curr.val);
                    prev = curr;  /* 保存刚访问过的结点 */
                } else {
                    /* 当前结点不能访问需第二次进栈, */
                    s.push(curr);
                    curr = curr.right; /* 先处理右子树 */
                    break;
                }
            }
        } while (!s.isEmpty());
        return res;
    }
```

2. Morris 先序遍历：Morris 先序遍历, 时间复杂度 O(n), 空间复杂度 O(1)
   这个参考的方法看起来有点儿复杂，就今天下午的状态，应该是做不出来了，晚上再看吧。。。

4.2.4 Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:

Given binary tree {3,9,20,#,#,15,7},

```
  3
 / \
9  20
  /  \
 15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

1. 递归版
2. 迭代版

4.2.5 Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example: Given binary tree {3,9,20,#,#,15,7},

```
  3
 / \
9  20
  /  \
 15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

1. 递归版

2. 迭代版

4.2.6 Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

Given binary tree {3,9,20,#,#,15,7},

```
  3
 / \
9  20
  /  \
 15   7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

1. 递归版

2. 迭代版

4.2.7 Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

O(n) 空间的解法是, 开一个指针数组, 中序遍历, 将节点指针依次存放到数组里, 然后寻找两处逆向的位置, 先从前往后找第一个逆序的位置, 然后从后往前找第二个逆序的位置, 交换这两个指针的值。

中序遍历一般需要用到栈, 空间也是 O(n) 的, 如何才能不使用栈?Morris 中序遍历。

4.2.8 Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(logn)
2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(logn)

4.2.9 Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following is not:

```
  1
 / \
2   2
 \   \
 3    3
```

Note:

Bonus points if you could solve it both recursively and iteratively.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(logn)
2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(logn)

4.2.10 Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

时间复杂度 O(n), 空间复杂度 O(logn)

4.2.11 Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example, Given

```
    1
   / \
  2   5
 / \   \
3   4   6
```

The flattened tree should look like:

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

click to show hints.

Hints:

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

1. 递归版 1: 时间复杂度 O(n), 空间复杂度 O(logn)
2. 递归版 2: 时间复杂度 O(n), 空间复杂度 O(logn)
3. 迭代版: 时间复杂度 O(n), 空间复杂度 O(logn)

4.2.12 Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space.

For example,

Given the following binary tree,

```
     1
   /  \
  2    3
 / \    \
4   5    7
```

After calling your function, the tree should look like:

```
     1 -> NULL
   /  \
  2 -> 3 -> NULL
 / \    \
4-> 5 -> 7 -> NULL
```

要处理一个节点, 可能需要最右边的兄弟节点, 首先想到用广搜。但广搜不是常数空间的, 本题要求常数空间。

注意, 这题的代码原封不动, 也可以解决 Populating Next Right Pointers in Each Node I.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(1)
2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(1)

4.3 二叉树的构建

4.3.1 Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.

4.3.2 Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.

4.4 二叉树查找

4.4.1 Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).
Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

  时间复杂度 O(n), 空间复杂度 O(log(n))

4.4.2 Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.
二分法。

4.4.3 Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

这题与上一题类似, 但是单链表不能随机访问, 而自顶向下的二分法必须需要 RandomAccessIterator, 因此前面的方法不适用本题。

存在一种自底向上 (bottom-up) 的方法, 见
http://leetcode.com/2010/11/convert-sorted-list-to-balanced-binary.html

1. 分治法：自顶向下
2. 自底向上

4.5 二叉树递归

二叉树是一个递归的数据结构, 因此是一个用来考察递归思维能力的绝佳数据结构。递归一定是深搜 (见“深搜与递归的区别”), 由于在二叉树上, 递归的味道更浓些, 因此本节用“二叉树的递归”作为标题, 而不是“二叉树的深搜”, 尽管本节所有的算法都属于深搜。

二叉树的先序、中序、后序遍历都可以看做是 DFS, 此外还有其他顺序的深度优先遍历, 共有 3! = 6 种。其他 3 种顺序是 root -> right -> left,right -> root -> left, right -> left -> root。

4.5.1 Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(log(n))
2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(log(n))

4.5.2 Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

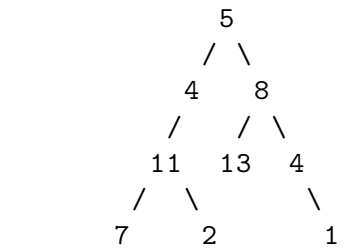
The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

4.5.3 Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

Given the below binary tree and sum = 22,



return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

题目只要求返回 true 或者 false, 因此不需要记录路径。

由于只需要求出一个结果, 因此, 当左、右任意一棵子树求到了满意结果, 都可以及时 return。

由于题目没有说节点的数据一定是正整数, 必须要走到叶子节点才能判断, 因此中途没法剪枝, 只能进行朴素深搜。

时间复杂度 O(n), 空间复杂度 O(log(n))

4.5.4 Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:
Given the below binary tree and sum = 22,

```
      5
     / \
    4   8
   /   / \
  11  13  4
 /  \    / \
7    2  5   1
```

return

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

跟上一题相比, 本题是求路径本身。且要求出所有结果, 左子树求到了满意结果, 不能 return, 要接着求右子树。
时间复杂度 O(n), 空间复杂度 O(log(n))

4.5.5 Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.
The path may start and end at any node in the tree.
For example: Given the below binary tree,

```
  1
 / \
2   3
```

Return 6.
这题很难, 路径可以从任意节点开始, 到任意节点结束。
可以利用“最大连续子序列和”问题的思路, 见第?????? 节。如果说 Array 只有一个方向的话, 那么 Binary Tree 其实只是左、右两个方向而已, 我们需要比较两个方向上的值。
不过,Array 可以从头到尾遍历, 那么 Binary Tree 怎么办呢, 我们可以采用 Binary Tree 最常用的 dfs 来进行遍历。先算出左右子树的结果 L 和 R, 如果 L 大于 0, 那么对后续结果是有利的, 我们加上 L, 如果 R 大于 0, 对后续结果也是有利的, 继续加上 R。
时间复杂度 O(n), 空间复杂度 O(log(n))

4.5.6 Populating Next Right Pointers in Each Node

Given a binary tree

```
struct TreeLinkNode {
    TreeLinkNode *left;
    TreeLinkNode *right;
    TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space.

You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,

```
     1
   /  \
  2    3
 / \  / \
4  5  6  7
```

After calling your function, the tree should look like:

```
     1 -> NULL
   /  \
  2 -> 3 -> NULL
 / \  / \
4->5->6->7 -> NULL
```

时间复杂度 O(n), 空间复杂度 O(log(n))

4.5.7 Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

```
  1
 / \
2   3
```

The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

时间复杂度 O(n), 空间复杂度 O(log(n))

4.5.8 Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example:

Given a binary tree {1, 2, 3, 4, 5},

```
    1
   / \
  2   3
 / \
4   5
```

return the root of the binary tree [4, 5, 2, #, #, 3, 1].

```
  4
 / \
5   2
   / \
  3   1
```

4.5.9 Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

4.5.10 Binary Tree Maximum Path Sum

4.6 others Covered

- Unique Binary Search Tree 14.2.4
- Unique Binary Search Tree II 14.2.5

Chapter 5

栈 Stack

5.1 Easy

5.1.1 Valid Parentheses

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

时间复杂度 O(n), 空间复杂度 O(n)。

5.2 Medium

5.2.1 Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Some examples:

```
["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

5.2.2 Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,

```
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"
```

click to show corner cases.

Corner Cases:

```
Did you consider the case where path = "/../"?
In this case, you should return "/".
Another corner case is the path might contain multiple slashes '/' together, such as "/
In this case, you should ignore redundant slashes and return "/home/foo".
```

5.3 Hard

5.3.1 Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

1. 使用栈: 时间复杂度 O(n), 空间复杂度 O(n)

   这个问题有点不一样的地方是，输入很可能不合理，也就是括号很可能不匹配，这怎么做？我的做法是用一个数组来做标记，依然是左括号入栈（入栈的是他的位置）。右括号出栈，当一对括号成功匹配的时候，把标记数组中他们相对应的位置标记为 1。这样过一遍之后，扫描一下整个标记数组，有多少个连续的 1，就是有多少个连续的匹配，输出一下个数就行了。

```
public int longestValidParentheses(String s) {
    Stack<Integer> k = new Stack<Integer>();
    int [] res = new int[s.length()];
    char j;
    int tmp = 0;
    for (int i = 0; i < s.length(); i++) {
        j = s.charAt(i);
        if (j == '('
            k.push(i);
        else if (j == ')'){
            if (!k.isEmpty()) {
                res[i] = 1;
                tmp = k.pop();
                res[tmp] = 1;
            }
        }
    }
    int max = 0;
    int i = 0;
    while (i < s.length()) {
        tmp = 0;  // reset value
        while (i < s.length() && res[i] == 0) i++;
        while (i < s.length() && res[i] == 1) {
            tmp++;
            i++;
        }
        max = Math.max(max, tmp);
    }
    return max;
}
```

   不用数组的思路：

   在处理括号匹配问题上，我们一般使用栈来解决。这一题也可以。

   顺序扫描字符串：

初始化：在栈中压入 -1

一、若碰到'('，则把当前位置压入栈中

二、若碰到')'：

(1)、如果栈顶元素不是'('，则把当前位置压入栈中；

(2)、如果栈顶元素时'('：栈顶元素出栈，当前的合法子串长度 = 当前字符索引 - 新的栈顶元素；更新最大子串长度

```
public int longestValidParentheses1(String s) {
    Stack<Integer> k = new Stack<Integer>();
    int res = 0;
    int tmp = 0;
    k.push(-1); //
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '('
            k.push(i);
        else { // ')'
            if (k.isEmpty())
                k.push(i);
            else {
                tmp = k.peek();
                if (tmp >= 0 && s.charAt(tmp) == '(') {
                    k.pop();
                    res = Math.max(res, i - k.peek());
                } else k.push(i);
            }
        }
    }
    return res;
}
```

2. 动态规划：

这种括号匹配的问题一般使用动态规划进行解决。使用 dp[i] 表示从位置 i 开始的最长合法括号序列的长度。

考虑第 i 个位置的括号：

- 如果是右括号，则 dp[i] = 0，因为第一个是右括号是没有合法匹配
- 如果是左括号，则考虑下一个位置 i+1 的情况，事实上，这是 dp[i+1] 的问题，可以以同样方法解决。之后可以知道第 i 位该匹配的位置为 i + dp[i+1] + 1 的位置，直接判断其是否匹配即可。

注意做括号匹配到其对应的右括号时，后面可能还有左括号以继续匹配。

```
public int longestValidParentheses(String s) {
    int n = s.length();
    int res = 0;
    int [] dp = new int[n]; // dp[i] 表示从 i 开始的最长合法括号序列
    int j = 0;
    for (int i = n - 2; i >= 0; i--) {
        if (s.charAt(i) == '(') {  // 第一个括号必须为左括号
```

```
                j = dp[i + 1] + i + 1; // 匹配该左括号的位置
                if (j < n && s.charAt(j) == ')') { // 满足匹配
                    dp[i] = dp[i + 1] + 2;
                    if (j + 1 < n)
                        dp[i] += dp[j + 1];      // 后面可能还有
                    res = Math.max(res, dp[i]);
                }
            }
        }
        return res;
    }
```

5.3.2 Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,
Given [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1], return 6.

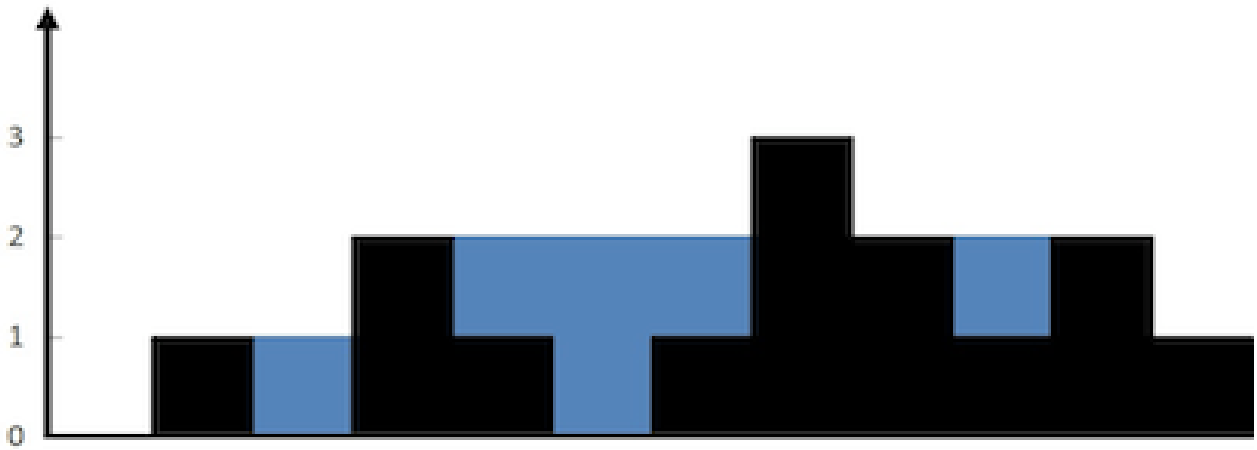


Figure 5.1: Trapping Rain Water

The above elevation map is represented by array [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

Tags: Array, Stack, Two Pointers

对于每个柱子, 找到其左右两边最高的柱子, 该柱子能容纳的面积就是 $\min(\max_{left}, \max_{right})$ - height。所以,

1. 从左往右扫描一遍, 对于每个柱子, 求取左边最大值;
2. 从右往左扫描一遍, 对于每个柱子, 求最大右值;
3. 再扫描一遍, 把每个柱子的面积并累加。

也可以,

1. 扫描一遍, 找到最高的柱子, 这个柱子将数组分为两半;

2. 处理左边一半;
3. 处理右边一半。

1. 思路 1, 时间复杂度 O(n), 空间复杂度 O(n)
2. 思路 2, 时间复杂度 O(n), 空间复杂度 O(1)
3. 思路 3, 时间复杂度 O(n), 空间复杂度 O(n)
4.

5.3.3 Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

Above is a histogram where width of each bar is 1, given height = [2, 1, 5, 6, 2, 3].

The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example,

Given height = [2, 1, 5, 6, 2, 3],

return 10.

简单的, 类似于 Container With Most Water(), 对每个柱子, 左右扩展, 直到碰到比自己矮的, 计算这个矩形的面积, 用一个变量记录最大的面积, 复杂度 O(n^2), 会超时。

如图所示, 从左到右处理直方, 当 i = 4 时, 小于当前栈顶 (即直方 3), 对于直方 3, 无论后面还是前面的直方, 都不可能得到比目前栈顶元素更高的高度了, 处理掉直方 3(计算从直方 3 到直方 4 之间的矩形的面积, 然后从栈里弹出); 对于直方 2 也是如此; 直到碰到比直方 4 更矮的直方 1。

这就意味着, 可以维护一个递增的栈, 每次比较栈顶与当前元素。如果当前元素小于栈顶元素, 则入栈, 否则合并现有栈, 直至栈顶元素小于当前元素。结尾时入栈元素 0, 重复合并一次。

1. O (n^2)

   最暴力的方法就是枚举所有区间的最大矩形值，然后选择最大的。可以通过分别枚举区间右边界和区间左边界，时间复杂度 O(n^2) 这样做大数据会超时。

   枚举的过程中可以优化一下：如果 height[i+1] > height[i], 那么区间 [k⋯i] 内的最大矩形肯定不会超过区间 [k⋯i+1] 内的最大矩形，比如上例中的区间 [0⋯3] 内的矩形要大于 [0⋯2] 内的矩形，因为 height[3] > height[2]。因此我们在枚举区间右边界时，只选择那些 height 上升区间的最大值处作为右边界（比如例子中的 2 、6 、3）。优化后可以通过 leetcode 的大数据，虽然做了优化，但是时间复杂度还是 O(n^2)。

```
public int largestRectangleArea(int[] height) {
    int n = height.length;
    int res = 0;
    int right = 0;
    int local = 0;
    int minVal = 0;
    while (right < n) {
        if (right + 1 < n && height[right + 1] >= height[right])
            right++;
        else { // found right border
            minVal = height[right];
            for (int left = right; left >= 0; left--) {
```

```
                minVal = Math.min(minVal, height[left]);
                local = minVal * (right - left + 1);
                res = Math.max(res, local);
            }
            right++;
            local = 0;
            minVal = 0;
        }
    }
    return res;
}
```

2. 栈：时间复杂度 O(n), 空间复杂度 O(n)。

   主要思路：

   我们知道，每个木板将对应多个包含自身的直方图面积，我们从中选取最大的一个面积。

   以上题为例，height[1] 对应的包含自身直方图面积集合为 {1, 2, 3, 4, 5, 6 } ，我们只需要知道最大的面积 6 即可，其它的面积就不用考虑。

   我们就比较所有 height[i] 对应最大直方图面积，把整体最大的直方图面积记为 maxArea。

   分析到这，我们很明显的可以知道时间复杂度为 O(n)。

   现在关键问题就变为如何得到 height[i] 对应的最大直方图面积。

   这里用一个栈 s 来保存之前的元素索引。

   s 为空时，直接让 i 栈: 以下取 i 从 0 到 n-1 。

   - 当 height[i] < height[s.top()] 时，说明 height[s.top()] 对应的直方图面积不会再增加了，此时达到最大，可以出栈了；
   - 当 height[i] >= height[s.top()] 时，说明 height[s.top()] 对应的直方图面积还是有潜力增加的哈，此时，i 入栈，++i 。

   当 i 达到 n 时，而栈不为空，说明还有部分 height 中元素对应的最大直方图面积没有求出来.

   于是，当栈不为空时，求 height[s.top()] 对应最大直方图面积。

```
public int largestRectangleArea(int[] height) {
    Stack<Integer> s = new Stack<Integer>();
    int [] hnew = new int[height.length + 1];// 数组末尾插入 dummy 元素 0
    hnew = Arrays.copyOf(height, height.length + 1);
    int res = 0;
    int tmp;
    int left;
    for (int i = 0; i <= height.length; i++) {
        if (s.isEmpty() || hnew[i] > hnew[s.peek()])
            s.push(i); // 栈内保存的是数组 height 的下标索引
        else {
            tmp = s.pop();
            res = Math.max(res, hnew[tmp] *
                           (s.isEmpty() ? i : i - s.peek() - 1));
            i--;
        }
```

```
        }
        return res;
    }
```

这题应该还有很多很巧妙的解法，念天脑袋不太转，暂时留在这里吧，改天再回来修改 ~

http://www.cnblogs.com/felixfang/p/3676193.html

5.4 others Covered

- Binary Tree Preorder Traversal 4.2.1
- Binary Tree Inorder Traversal 4.2.2
- Binary Tree Postorder Traversal 4.2.3
- Binary Search Tree Iterator 4.5.9
- Binary Tree Zigzag Level Order Traversal 4.2.6
- Maximal Rectangle 14.3.3
- Min Stack 19.1.2

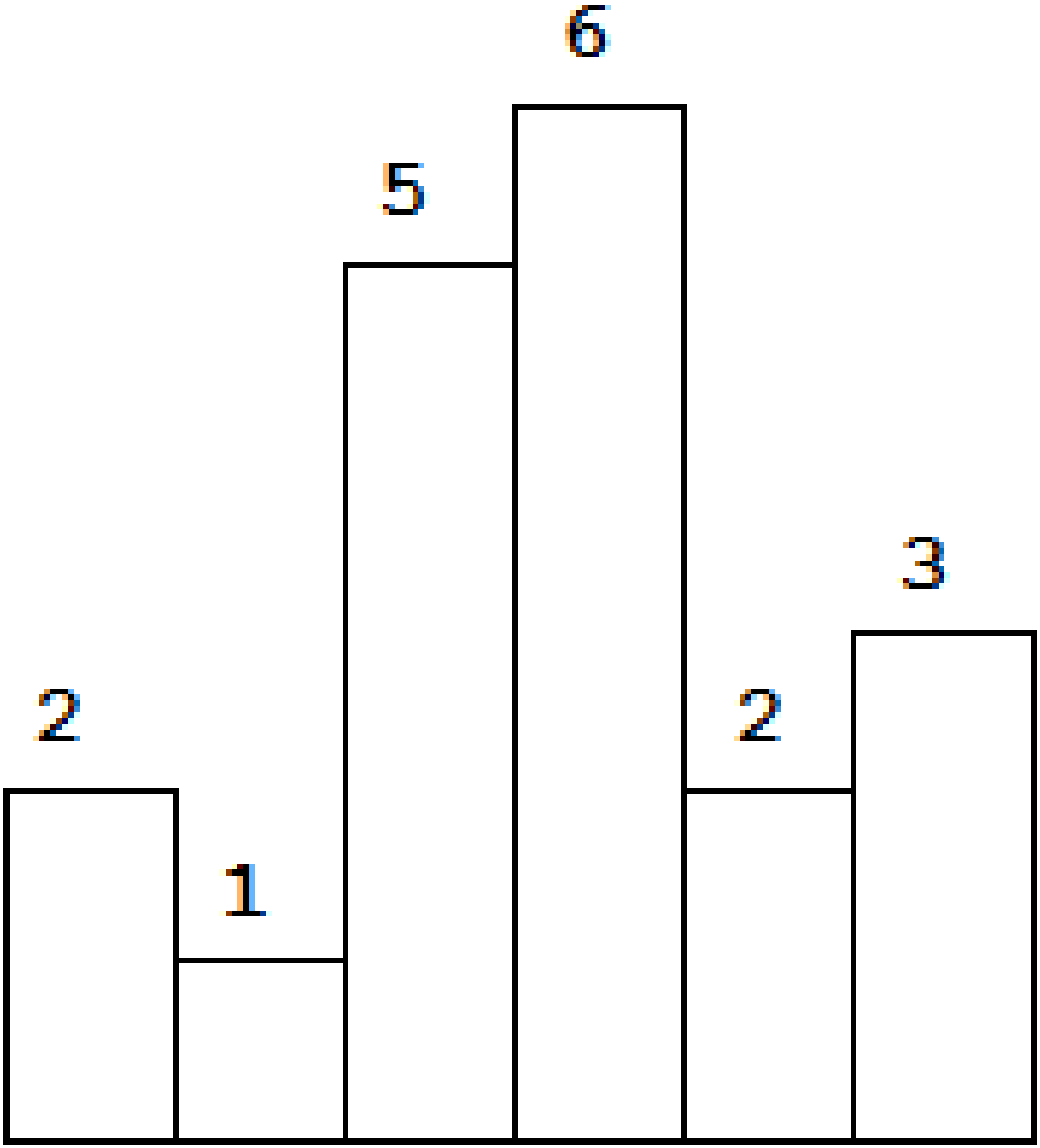


Figure 5.2: Largest Rectangle in Histogram 1

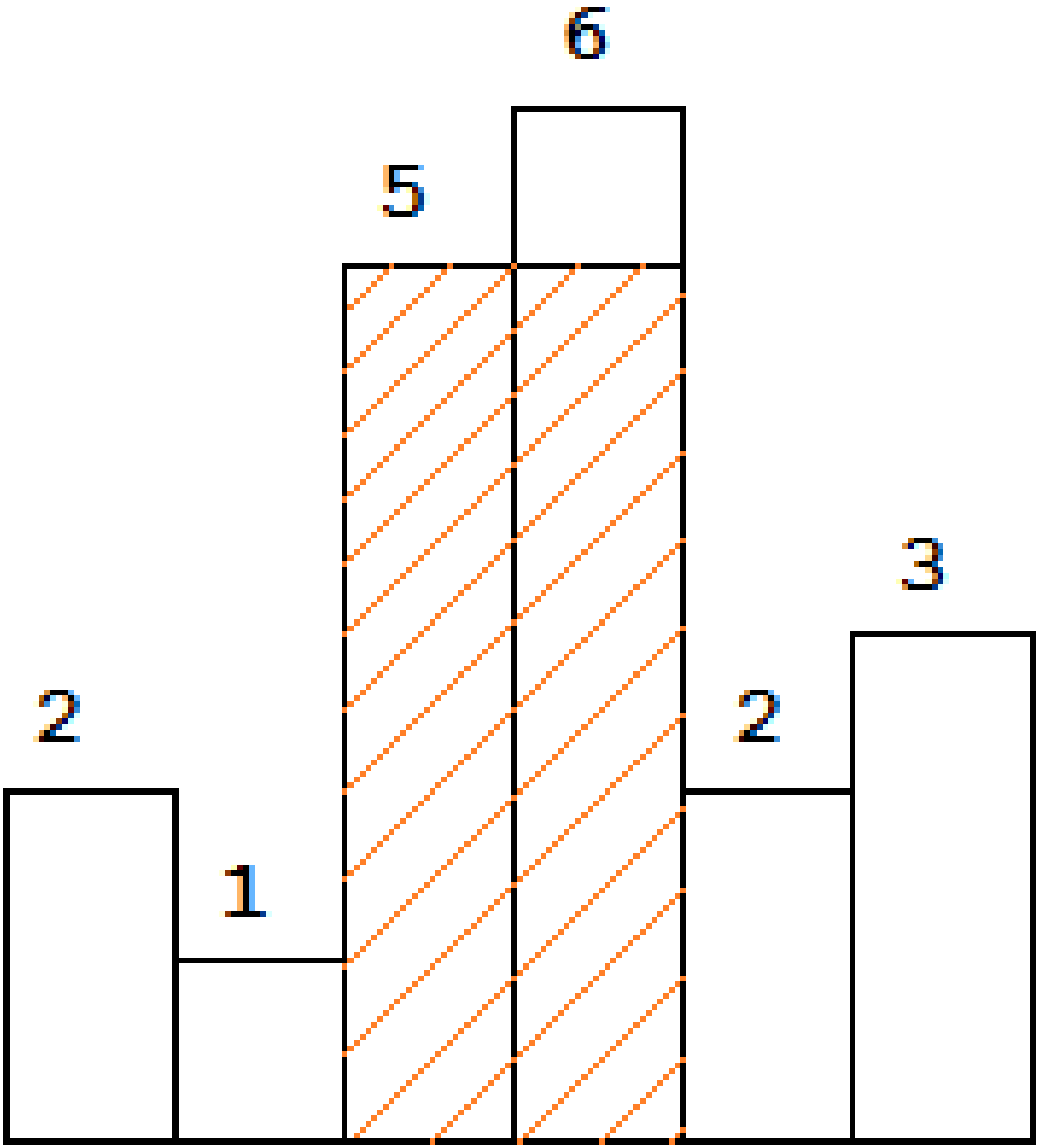


Figure 5.3: Largest Rectangle in Histogram 2

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 Copy List with Random Pointer

6.3.3 Sudoku Solver

6.3.4 Max Points on a Line

6.3.5 Substring with Concatenation of All Words

6.3.6 Longest Substring with At Most Two Distinct Characters

6.4 other Covered

- Maximal Rectangle 14.3.3

Chapter 7

排序

7.1 Easy

7.1.1 Merge Sorted Array

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

Note:

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

时间复杂度 O(m+n), 空间复杂度 O(1).

```
public void merge(int a[], int m, int b[], int n) {
    int ia = m - 1;
    int ib = n - 1;
    int icur = m + n - 1;
    while (ia >= 0 && ib >= 0)
        a[icur--] = a[ia] > b[ib] ? a[ia--] : b[ib--];
    while (ib >= 0)
        a[icur--] = b[ib--];
}
```

7.1.2 Merge Two Sorted List

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.

时间复杂度 O(m+n), 空间复杂度 O(1).

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(-1);     // dummy List ~!
    for (ListNode p = head; l1 != null || l2 != null; p = p.next) {
        int val1 = l1 == null ? Integer.MAX_VALUE : l1.val;
        int val2 = l2 == null ? Integer.MAX_VALUE : l2.val;
        if (val1 < val2) {
            p.next = l1;
            l1 = l1.next;
        } else {
            p.next = l2;
            l2 = l2.next;
```

```
        }
    }
    return head.next;
}
```

7.2 Medium

7.2.1 Sort Colors

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:

You are not suppose to use the library's sort function for this problem.

click to show follow up.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

Tags: Array, Two Pointers, Sort

由于 0, 1, 2 非常紧凑, 首先想到计数排序 (counting sort), 但需要扫描两遍, 不符合题目要求。

由于只有三种颜色, 可以设置两个 index, 一个是 red 的 index, 一个是 blue 的 index, 两边往中间走。时间复杂度 O(n), 空间复杂度 O(1)。

第 3 种思路, 利用快速排序里 partition 的思想, 第一次将数组按 0 分割, 第二次按 1 分割, 排序完毕, 可以推广到 n 种颜色, 每种颜色有重复元素的情况。

时间复杂度 O(n), 空间复杂度 O(1).

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

```
public void sortColors(int[] a) {
    int [] cnt = new int[3];  // 记录每个颜色出现的次数
    for (int i = 0; i < a.length; i++)
        cnt[a[i]]++;
    for (int i = 0, index = 0; i < 3; i++) {
        for (int j = 0; j < cnt[i]; j++) {
            a[index++] = i;
        }
    }
}
```

2. 双指针: 时间复杂度 O(n), 空间复杂度 O(1)

```
public void swap(int [] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}

public void sortColors(int[] a) {
```

```
        if (a == null || a.length < 2) return;
        int n = a.length;
        int i = 0;
        // 一个是 red 的 index一个是, blue 的 index两边往中间走,
        int zEnd = -1;  // 0 end
        int tEnd = n;   // 2 end
        while (i < tEnd) {
            if (a[i] == 0 && i != ++zEnd) // first 0
                swap(a, zEnd, i);
            else if (a[i] == 2 && i != --tEnd)
                swap(a, i, tEnd);
            else i++;
        }
        return;
    }
```

3. use partition():

4. use partition():

7.2.2 Sort List

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

```
public ListNode divideList(ListNode head) {
    ListNode fast = head;
    ListNode slow = head;
    ListNode prev = head;
    while (fast != null && fast.next != null) {
        prev = slow;
        slow = slow.next;
        fast = fast.next.next;
    }
    prev.next = null;
    return slow;
}

public ListNode mergeList(ListNode one, ListNode two) {
    if (one == null) return two;
    if (two == null) return one;
    ListNode dummy = new ListNode(Integer.MIN_VALUE);
    ListNode curr = dummy;
    while (one != null && two != null) {
        if (one.val < two.val) {
            curr.next = one;
            one = one.next;
        } else {
            curr.next = two;
            two = two.next;
        }
        curr.next.next = null;
```

```
            curr = curr.next;
        }
        if (one != null) curr.next = one;
        else curr.next = two;
        return dummy.next;
    }

    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode second = divideList(head);
        ListNode firstSorted = sortList(head);
        ListNode secondSorted = sortList(second);
        return mergeList(firstSorted, secondSorted);
    }
```

7.3 Hard

7.3.1 Merge k sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.
Tags: Divide and Conquer, Linked List, Heap
可以复用 Merge Two Sorted Lists(见 Merge Two Sorted List 7.1.2) 的函数
时间复杂度 O(n1 + n2 + ⋯), 空间复杂度 O(1).

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(-1);     // dummy List ~!
    for (ListNode p = head; l1 != null || l2 != null; p = p.next) {
        int val1 = l1 == null ? Integer.MAX_VALUE : l1.val;
        int val2 = l2 == null ? Integer.MAX_VALUE : l2.val;
        if (val1 < val2) {
            p.next = l1;
            l1 = l1.next;
        } else {
            p.next = l2;
            l2 = l2.next;
        }
    }
    return head.next;
}

public ListNode mergeKLists(List<ListNode> lists) {
    if (lists.size() == 0) return null;
    ListNode p = lists.get(0);  // Time Limit Exceeded
    for (int i = 1; i < lists.size(); i++) {
        p = mergeTwoLists(p, lists.get(i));
    }
    return p;
}
```

```
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()))));
}
```

7.3.2 Insertion Sort List

Sort a linked list using insertion sort.
这个，代码过于混乱，需要优化。。。

```
public ListNode insertionSortList(ListNode head) {
    if (head == null) return null;
    ListNode prev = head;
    ListNode curr = head.next;
    ListNode next = head;
    ListNode tnext = head;
    ListNode tmp = head;
    while (curr != null) {
        next = curr.next;
        if (curr.val < prev.val) {
            prev.next = next;
            curr.next = null;
            if (curr.val < head.val) {
                curr.next = head;
                head = curr;
            } else {
                tmp = head;
                while (tmp.next != prev && curr.val > tmp.next.val) {
                    tmp = tmp.next;
                }
                tnext = tmp.next;
                tmp.next = curr;
                curr.next = tnext;
            }
            curr = next;
        } else {
            prev = curr;
            curr = curr.next;
        }
    }
    return head;
}
```

7.3.3 First Missing Positive

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

Given [1, 2, 0] return 3,

and [3, 4, -1, 1] return 2.

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

本质上是桶排序 (bucket sort), 每当 A[i]!= i+1 的时候, 将 A[i] 与 A[A[i]-1] 交换, 直到无法交换为止, 终止条件是 A[i]== A[A[i]-1]。

```
public void swap(int [] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}

public void bucket_sort(int [] a) {
    for (int i = 0; i < a.length; i++) {
        while (a[i] != i + 1) {
            if (a[i] <= 0 || a[i] > a.length || a[i] == a[i] - 1)
                break;
            swap(a, i, a[i] - 1);
        }
    }
}

// need to clean code
// 时间复杂度 O(n), 空间复杂度 O(1)
public int firstMissingPositive(int[] a) {
    // Time Limit Exceeded
    bucket_sort(a);
    for (int i = 0; i < a.length; i++)
        if (a[i] != (i + 1))
            return i + 1;
    return a.length + 1;
}

public int firstMissingPositive0(int[] a) {
    if (a == null || a.length == 0) return 1;
    if (a.length == 1) return a[0] == 1 ? 2 : 1;

    int n = a.length;
    boolean equl = false;
    int i;
    for ( i = 0; i < n; i++) {
        if (a[i] < 0) continue;
        if (a[i] == n) {
            equl = true;
            continue;
        }
        while ( (i == 0 || (i != 0 && a[i] != i))
                && a[i] < n && a[i] > 0 ) {
            swap(a, i, a[i]);
            if (a[i] == n) {
                equl = true;
```

```
                break;
            }
            if ((a[i] > n || a[i] < 0) || (a[i] >= 0 && a[i] == a[a[i]]))
                break; // to avoid infite loop
        }
    }
    for ( i = 1; i < n; i++)
        if (a[i] != i) return i;
    if (equl || a[0] == n) return n + 1;
    else return n;
}
```

7.4 other Covered

7.4.1 Largest Number

7.4.2 Maximum Gap

- Merge Intervals
- Insert Interval

Chapter 8

查找

8.1 Medium

8.1.1 Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.
Your algorithm's runtime complexity must be in the order of O(log n).
If the target is not found in the array, return [-1, -1].
For example,
Given [5, 7, 7, 8, 8, 10] and target value 8,
return [3, 4].
Tags: Array, Binary Search
已经排好了序, 用二分查找。
时间复杂度 O(logn), 空间复杂度 O(1).
分两步来做，找一次第一个，找一次最后一个，返回。

```
public int searchFirsst(int [] a, int target, int start, int end) {
    if (start == end) return (a[end] == target) ? start : -1;
    if (start == end - 1) {
        if (a[start] == target) return start;
        else if (a[end] == target) return end;
        else return -1;
    }

    while (start < end - 1) {
        int mid = start + (end - start) / 2;
        if (a[mid] >= target)
            end = mid;
        else
            start = mid + 1;
    }
    if (start == end) return (a[end] == target) ? start : -1;
    if (start == end - 1) {
        if (a[start] == target) return start;
        else if (a[end] == target) return end;
        else return -1;
    }
    return -1;
```

```
}

public int searchLast(int [] a, int target, int start, int end) {
    if (start == end) return (a[end] == target) ? end : -1;
    if (start == end - 1) {
        if (a[end] == target) return end;
        else if (a[start] == target) return start;
        else return -1;
    }

    while (start < end - 1) {
        int mid = start + (end - start) / 2;
        if (a[mid] > target)
            end = mid - 1;
        else
            start = mid;
    }
    if (start == end) return (a[end] == target) ? end : -1;
    if (start == end - 1) {
        if (a[end] == target) return end;
        else if (a[start] == target) return start;
        else return -1;
    }
    return -1;
}

public int[] searchRange(int[] A, int target) {
    int [] res = new int[2];
    if (A == null || A.length == 0 ||
        (A.length == 1 && A[0] != target)) {
        res[0] = -1;
        res[1] = -1;
        return res;
    } else if (A.length == 1) {
        res[0] = 0;
        res[1] = 0;
        return res;
    }
    res[0] = searchFirsst(A, target, 0, A.length-1);
    res[1] = searchLast(A, target, 0, A.length-1);
    return res;
}
```

8.1.2 Search Insert Position

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

You may assume no duplicates in the array.

Here are few examples.

```
[1, 3, 5, 6],  5 → 2
```

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

直接的二分查找，查找第一个大于或等于 target 的位置。对于中间位置 mid：
如果 A[mid] >= target，那么答案至少应该为 mid，或者为 mid 的左边位置，则 end = mid
否则 A[mid] < target，那么答案肯定在 mid 的右边，而且不可能为 mid，则 bgn = mid + 1
时间复杂度 O(logn), 空间复杂度 O(1).

```
public int searchInsert(int[] a, int target) {
    int bgn = 0, end = a.length - 1;
    while (bgn <= end) {
        int mid = bgn + (end - bgn) / 2;
        if (a[mid] ==  target) return mid;
        if (mid > bgn && a[mid] > target && a[mid - 1] < target) return mid;
        if (a[mid] > target)
            end = mid - 1;
        else
            bgn = mid + 1;
    }
    return bgn;
}
```

8.1.3 Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted from left to right.

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

For example,

Consider the following matrix:

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

Given target = 3, return true.

二分查找。

时间复杂度 O(logn), 空间复杂度 O(1).

```
public boolean binarySearch(int [] a, int target, int bgn, int end) {
    while (bgn <= end) {
        int mid = bgn + (end - bgn) / 2;
        if (a[mid] == target) return true;
        if (a[mid] > target) end = mid - 1;
        else bgn = mid + 1;
    }
    return false;
}
```

```
public boolean searchMatrix(int[][] matrix, int target) {
    int m = matrix.length;
    int n = matrix[0].length;
    int [] a = new int[m * n];
    int cnt = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            a[cnt++] = matrix[i][j];
    return binarySearch(a, target, 0, m * n - 1);
}
```

也可以把两个函数合并成一个,，省去中间合成桥梁数组的步骤，代码如下：

```
public boolean searchMatrix(int[][] matrix, int target) {
    int m = matrix.length;
    int n = matrix[0].length;
    int bgn = 0;
    int end = m * n- 1;
    while (bgn <= end) {
        int mid = bgn + (end - bgn) / 2;
        int val = matrix[mid / n][mid % n];
        if (val == target) return true;
        if (val > target) end = mid - 1;
        else bgn = mid + 1;
    }
    return false;
}
```

Chapter 9

暴力枚举法

9.1 Medium

9.1.1 Subsets

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

1. 递归，增量构造法: 增量构造法, 深搜, 时间复杂度 O(2^n), 空间复杂度 O(n)
   每个元素, 都有两种选择, 选或者不选。

```
public void helper(int [] s, List<Integer> path, int step, List<List<Integer>> res)
    if (step == s.length) {
        res.add(new ArrayList<Integer>(path));
        return;
    }
    helper(s, path, step + 1, res);  // 不选 S[step]
    path.add(s[step]);               // 选 S[step]
    helper(s, path, step + 1, res);
    path.remove(path.size() - 1);
```

```
}

public List<List<Integer>> subsets(int[] s) {
    Arrays.sort(s);             // 输出要求有序
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    helper(s, path, 0, res);
    return res;
}
```

2. 递归，位向量法：时间复杂度 O(2^n), 空间复杂度 O(n).

   开一个位向量 bool selected[n], 每个元素可以选或者不选。

   位向量法, 深搜, 时间复杂度 O(2^n), 空间复杂度 O(n).

```
public void helper(int [] s, boolean [] selected, int step, List<List<Integer>> res
    if (step == s.length) {
        List<Integer> path = new ArrayList<Integer>();
        for (int i = 0; i < s.length; i++) {
            if (selected[i]) path.add(s[i]);
        }
        res.add(path);
        return;
    }
    selected[step] = false;
    helper(s, selected, step + 1, res);
    selected[step] = true;
    helper(s, selected, step + 1, res);
}

public List<List<Integer>> subsets(int[] s) {
    Arrays.sort(s);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    boolean [] selected = new boolean[s.length];
    helper(s, selected, 0, res);
    return res;
}
```

3. 迭代，增量构造法，时间复杂度 O(2^n), 空间复杂度 O(1)

```
vector<vector<int> > subsets(vector<int> &S) {
    sort(S.begin(), S.end()); // 输出要求有序
    vector<vector<int> > result(1);
    for (auto elem : S) {
        result.reserve(result.size() * 2);
        auto half = result.begin() + result.size();
        copy(result.begin(), half, back_inserter(result));
        for_each(half, result.end(), [&elem](decltype(result[0]) &e){
                e.push_back(elem);
            });
    }
    return result;
```

```
}
```

需要改写成 Java 版。

4. 二进制法

本方法的前提是: 集合的元素不超过 int 位数。用一个 int 整数表示位向量, 第 i 位为 1, 则表示选择 S[i], 为 0 则不选择。例如 S={A, B, C, D}, 则 0110=6 表示子集 {B, C}。

这种方法最巧妙。因为它不仅能生成子集, 还能方便的表示集合的并、交、差等集合运算。设两个集合的位向量分别为 B1 和 B2 , 则 B1 | B2 , B1 & B2 , B1 B2 分别对应集合的并、交、对称差。

二进制法, 也可以看做是位向量法, 只不过更加优化。

二进制法, 时间复杂度 $O(2^n)$, 空间复杂度 O(1)

```
public List<List<Integer>> subsets(int[] s) {
    Arrays.sort(s);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    int n = s.length;
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            // if ((i & (1 << j)) != 0) path.add(s[j]);
            if(((i >> j) & 1 ) != 0) path.add(s[j]);
        }
        res.add(path);
        path = new ArrayList<Integer>();
    }
    return res;
}
```

总体看来，不管是用向量数组来保存选与不选，还是位操作，都是选与不选的控制。通过有序控制来得到想要的结果。

9.1.2 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],
  []
]
```

这题有重复元素, 但本质上, 跟上一题很类似, 上一题中元素没有重复, 相当于每个元素只能选 0 或 1 次, 这里扩充到了每个元素可以选 0 到若干次而已。

1. 递归，增量构造法: 增量构造法, 深搜, 时间复杂度 $O(2^n)$, 空间复杂度 O(n)

```
public void helper(int [] s, List<Integer> path, List<List<Integer>> res) {
    res.add(new ArrayList<Integer>(path));
    for (int i = 0; i < s.length; i++) {
        if (i != 0 && s[i] == s[i - 1]) continue;
        path.add(s[i]);
        helper(s, path, res);
        path.remove(path.size() - 1);
    }
}

// Memory Limit Exceeded
public List<List<Integer>> subsetsWithDup(int[] s) {
    Arrays.sort(s);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    helper(s, path, res);
    return res;
}
```

2. 增量构造法, 版本 2, 时间复杂度 $O(2^n)$, 空间复杂度 O(n)
3. 位向量法, 时间复杂度 $O(2^n)$, 空间复杂度 O(n)
   这个相对复杂，改天再写
4. 迭代，增量构造法，时间复杂度 $O(2^n)$, 空间复杂度 O(1)
5. 二进制法

```
public List<List<Integer>> subsetsWithDup(int[] s) {
    Arrays.sort(s);
    HashSet<List<Integer>> res = new HashSet<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    int n = s.length;
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            if ((i & (1 << j)) != 0) path.add(s[j]);
            //if(((i >> j) & 1 ) != 0) path.add(s[j]);
        }
        res.add(path);
        path = new ArrayList<Integer>();
    }
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    result.addAll(res);
    return result;
}
```

与上一题类似，只是多一步控制对重复的处理，或不再执行重复的，或执行后去重处理。

9.1.3 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].
```

偷懒的做法, 可以直接使用 c++ std::next$_{permutation}$()。如果是在 OJ 网站上, 可以用这个 API 偷个懒; 如果是在面试中, 面试官肯定会让你重新实现。
时间复杂度 O(n!), 空间复杂度 O(1)

1. 重新实现 next$_{permutation}$(): 时间复杂度 O(n!), 空间复杂度 O(1)
2. 递归：时间复杂度 O(n!), 空间复杂度 O(n)

   本题是求路径本身, 求所有解, 函数参数需要标记当前走到了哪步, 还需要中间结果的引用, 最终结果的引用。

   扩展节点, 每次从左到右, 选一个没有出现过的元素。

   本题不需要判重, 因为状态装换图是一颗有层次的树。收敛条件是当前走到了最后一个元素。

```
public void helper(int [] num, boolean [] used, List<Integer> one,
                   List<List<Integer>> res) {
    if (one.size() == num.length) {
        res.add(new ArrayList<Integer>(one));
        return;
    }
    for (int i = 0; i < num.length; i++) {
        if (!used[i]) {
            used[i] = true;
            one.add(num[i]);
            helper(num, used, one, res);
            one.remove(one.size() - 1);
            used[i] = false;
        }
    }
    return;
}

public List<List<Integer>> permute(int[] num) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if (num == null || num.length == 0) return res;
    boolean [] used = new boolean[num.length];
    helper(num, used, new ArrayList<Integer>(), res);
    return res;
}
```

9.1.4 Combinationas

Given two integers n and k, return all possible combinations of k numbers out of 1 ⋯ n.
For example,
If n = 4 and k = 2, a solution is:

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

1. 递归: 深搜, 时间复杂度 O(n!), 空间复杂度 O(n)

   时间复杂度 O(n!), 空间复杂度 O(n).

```
// bgn, 开始的数, cur, 已经选择的数目
public void dfs(List<List<Integer>> res, List<Integer> path, int n, int k, int bgn,
    if (cur == k) {
        res.add(new ArrayList<Integer>(path));
    }
    for (int i = bgn; i <= n; i ++) {
        path.add(i);
        dfs(res, new ArrayList<Integer>(path), n, k, i + 1, cur + 1);
        path.remove(path.size() - 1);
    }
}

public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs(res, path, n, k, 1, 0);
    return res;
}
```

2. 迭代：时间复杂度 O((n-k)!), 空间复杂度 O(n)

   时间复杂度 O((n-k)!), 空间复杂度 O(n).

   some other day~ !!@

9.1.5 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.

1. 递归: 深搜, 时间复杂度 O(3^n), 空间复杂度 O(n)

```
public char [] getArr(char x) {
    char [] two = {'a', 'b', 'c'};
    char [] thr = {'d', 'e', 'f'};
```

Figure 9.1: Letter Combinationas of Phone Number

```
    char [] fou = {'g', 'h', 'i'};
    char [] fiv = {'j', 'k', 'l'};
    char [] six = {'m', 'n', 'o'};
    char [] sev = {'p', 'q', 'r', 's'};
    char [] eig = {'t', 'u', 'v'};
    char [] nin = {'w', 'x', 'y', 'z'};
    switch (x) {
    case '2': return two;
    case '3': return thr;
    case '4': return fou;
    case '5': return fiv;
    case '6': return six;
    case '7': return sev;
    case '8': return eig;
    case '9': return nin;
    }
    return null;
}
```

```
public void recursion(String digits, List<String> result) {
    char [] base = getArr(digits.charAt(digits.length()-1));
    int n = result.size();
    //for(String x : result) // ConcurrentModificationException, result is changing
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < base.length; i++)
            result.add(new StringBuffer(base[i]+ result.get(j)).toString());
    }
    for (int i = n-1; i >= 0; i--)
        result.remove(i);
    if (digits.length() > 1)
        recursion(digits.substring(0, digits.length()-1), result);
}

public List<String> letterCombinations(String digits) {
    List<String> result = new ArrayList<String>();
    if (digits.equals("")) {
        result.add("");
        return result;
    }
    char [] base = getArr(digits.charAt(digits.length()-1));
    for (int i = 0; i < base.length; i++)
        result.add("" + base[i]);
    if (digits.length() > 1)
        recursion(digits.substring(0, digits.length()-1), result);
    return result;
}
```

2. 迭代：时间复杂度 O(3^n), 空间复杂度 O(1)

9.2 Hard

9.2.1 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].
```

1. next$_{permutation}$()

   直接使用 std::next$_{permutation}$(), 代码与上一题相同。

2. 重新实现 next$_{permutation}$()

   重新实现 std::next$_{permutation}$(), 代码与上一题相同。

3. 递归:

   递归函数 permute() 的参数 p, 是中间结果, 它的长度又能标记当前走到了哪一步, 用于判断收敛条件。

扩展节点, 每次从小到大, 选一个没有被用光的元素, 直到所有元素被用光。

本题不需要判重, 因为状态装换图是一颗有层次的树。

```
public void helper(int [] num, boolean [] used, List<Integer> one,
                   List<List<Integer>> res, int cnt) {
    if (one.size() == num.length) {
        res.add(new ArrayList<Integer>(one));
        return;
    }
    for (int i = 0; i < num.length; i++) {
        // !used[i - 1] marks the completion of first 0 done, so ignore second 0
        if (i > 0 && !used[i - 1] && num[i] == num[i - 1]) continue;
        if (!used[i]) {
            used[i] = true;
            one.add(num[i]);
            helper(num, used, one, res, ++cnt);
            one.remove(one.size() - 1);
            used[i] = false;
        }
    }
    return;
}

public List<List<Integer>> permuteUnique(int[] num) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if (num == null || num.length == 0) return res;
    boolean [] used = new boolean[num.length];
    Arrays.sort(num);
    helper(num, used, new ArrayList<Integer>(), res, 1);
    return res;
}
```

应该还有其它更好的控制原理，只是今天不在状态，不用多想。。。

Chapter 10

广度优先搜索 Breadth First Search

当题目看不出任何规律, 既不能用分治, 贪心, 也不能用动规时, 这时候万能方法——搜索, 就派上用场了。搜索分为广搜和深搜, 广搜里面又有普通广搜, 双向广搜, A* 搜索等。深搜里面又有普通深搜, 回溯法等。

广搜和深搜非常类似 (除了在扩展节点这部分不一样), 二者有相同的框架, 如何表示状态? 如何扩展状态? 如何判重? 尤其是判重, 解决了这个问题, 基本上整个问题就解决了。

10.1 Medium

10.1.1 Surrounded Regions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Tags: Breadth-first Search

广搜。从上下左右四个边界往里走, 凡是能碰到的'O', 都是跟边界接壤的, 应该删除。

这叫广搜啊? 我还以为延着 0 深搜呢。。。仔细想了下，以树的 level order Traversal 作为类比，这个确实应该是广搜，而不是深搜。

四面的最外层搜索，有 O 通路的就为不被包围的区域，其他都可以置为 X；没把握这个特点，难度为五星级的。思路还是很新颖的。

注意栈溢出总题。

```
public void solve(char[][] board) {
    if (board.length == 0) return;
    int row = board.length;
    int col = board[0].length;
    for (int i = 0; i < row; i++) {
```

```
        for (int j = 0; j < col; j++) {
            if ((i == 0 || j == 0 || i == row-1 || j == col-1) && board[i][j] == 'O') {
                Stack<Integer> stk = new Stack<Integer>();
                stk.push(i * col + j);          // cannot *row, produce error, the large
                while (!stk.isEmpty()) {
                    int tmp = stk.pop();
                    int x = tmp / col;
                    int y = tmp % col;
                    if (x < 0 || y < 0 || x >= row || y >= col ||
                        !(board[x][y] == 'O'))
                        continue;
                    board[x][y] = '#';
                    stk.push((x-1) * col + y);
                    stk.push((x+1) * col + y);
                    stk.push(x * col + (y - 1));
                    stk.push(x * col + (y + 1));
                }
            }
        }
    }

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (board[i][j] == 'O')
                board[i][j] = 'X';
            if (board[i][j] == '#')
                board[i][j] = 'O';
        }
    }
}
```

10.1.2 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.

时间复杂度 O(n), 空间复杂度 O(n).

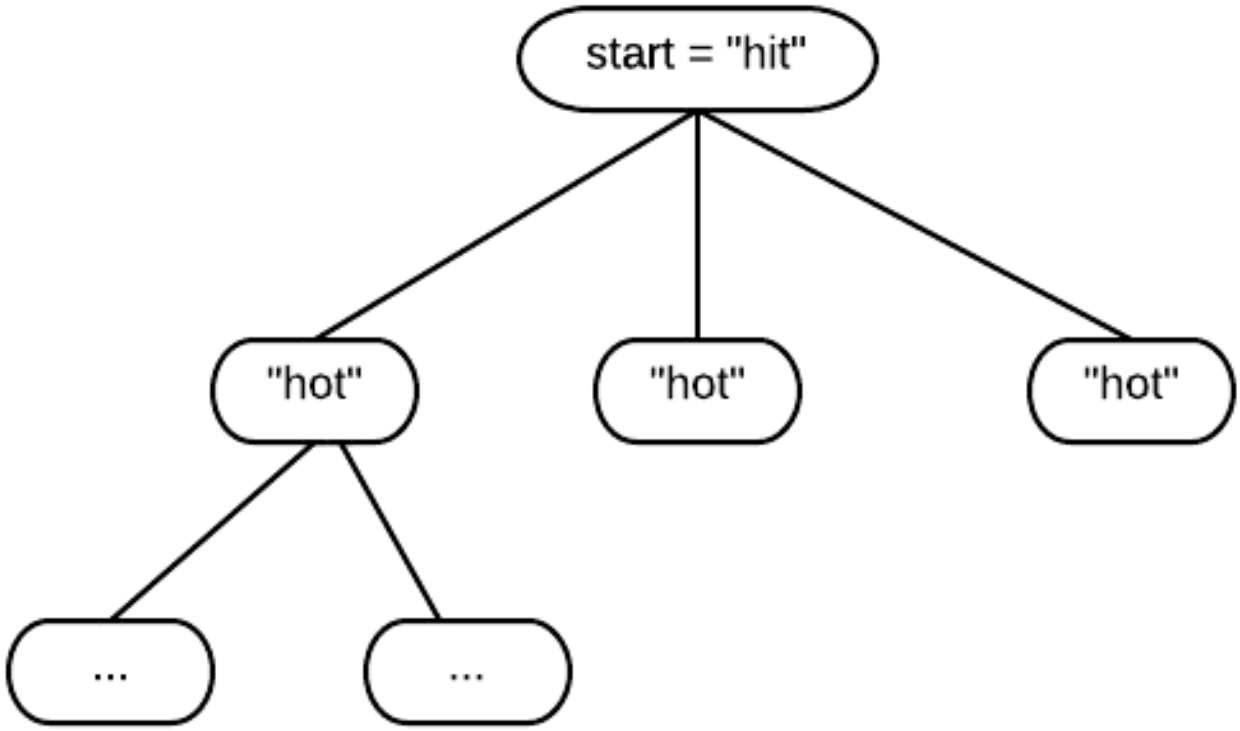


Figure 10.1: Word Ladder

分析：这种题，肯定是每次改变单词的一个字母，然后逐渐搜索，很多人一开始就想到用 dfs，其实像这种求最短路径、树最小深度问题 bfs 最适合，可以参考我的这篇博客 bfs（层序遍历）求二叉树的最小深度。本题 bfs 要注意的问题：

和当前单词相邻的单词是：对当前单词改变一个字母且在字典中存在的单词, 找到一个单词的相邻单词，加入 bfs 队列后，要从字典中删除，因为不删除的话会造成类似于 hog->hot->hog 的死循环。而删除对求最短路径没有影响，因为我们第一次找到该单词肯定是最短路径，即使后面其他单词也可能转化得到它，路径肯定不会比当前的路径短（如果要输出所有最短路径，则不能立即从字典中删除，具体见下一题）

bfs 队列中用 NULL 来标识层与层的间隔，每次碰到层的结尾，遍历深度 +1

我们利用和求二叉树最小深度层序遍历的方法来进行 bfs，代码如下：

```
public int ladderLength(String start, String end, Set<String> dict) {
    if (start.length() != end.length()) return 0;
    if (start.compareTo("") == 0) return 1;
    List<String> newStart = new ArrayList<String>();
    List<String> nextStart = new ArrayList<String>();
    newStart.add(start);
    int distance = 1;
    for (int index = 0; index < newStart.size(); index++) {
        String curr = newStart.get(index);
```

```
        for (int i = 0; i < curr.length(); i++) {
            for (char j = 'a'; j <= 'z'; j++) {
                if (curr.charAt(i) == j) continue;
                String tmp = curr.substring(0, i) + j + curr.substring(i + 1);
                if (tmp.equals(end)) return distance + 1;
                if (dict.contains(tmp)) {
                    nextStart.add(tmp);
                    dict.remove(tmp);
                }
            }
        }
        if (index == newStart.size() - 1) {
            newStart.addAll(new ArrayList<String>(nextStart));
            nextStart = new ArrayList<String>();
            distance++;
        }
    }
    return 0;
}
```

应该还有更多更好的方法，改天再深入去写。

10.2 Hard

10.2.1 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.

跟 Word Ladder 比, 这题是求路径本身, 不是路径长度, 也是 BFS, 略微麻烦点。
这题跟普通的广搜有很大的不同, 就是要输出所有路径, 因此在记录前驱和判重地方与普通广搜不同。

时间复杂度 O(n), 空间复杂度 O(n)。

这道题是 LeetCode 中 AC 率最低的题目，确实是比较难。一方面是因为对时间有比较严格的要求（容易超时），另一方面是它有很多细节需要实现。思路上和 Word Ladder 是比较类似的，但是因为是要求出所有路径，仅仅保存路径长度是不够的，而且这里还有更多的问题，那就是为了得到所有路径，不是每个结点访问一次就可以标记为 visited 了，因为有些访问过的结点也会是别的路径上的结点，所以访问的集合要进行回溯（也就是标记回未访问）。所以时间上不再是一次广度优先搜索的复杂度了，取决于结果路径的数量。同样空间上也是相当高的复杂度，因为我们要保存过程中满足的中间路径到某个数据结构中，以便最后可以获取路径，这里我们维护一个 HashMap，把一个结点前驱结点都进行保存。

在 LeetCode 中用 Java 实现上述算法非常容易超时。为了提高算法效率，需要注意一下两点：

1）在替换 String 的某一位的字符时，先转换成 char 数组再操作；

2）如果按照正常的方法从 start 找 end，然后根据这个来构造路径，代价会比较高，因为保存前驱结点容易，而保存后驱结点则比较困难。所以我们在广度优先搜索时反过来先从 end 找 start，最后再根据生成的前驱结点映射从 start 往 end 构造路径，这样算法效率会有明显提高。

```
/* Graph of example:
 *                 |--- dot --- dog ---|
 * hit --- hot -- |     |       |     |--- cog
 *                 |--- lot --- log ---|
 */
// set: visited   // current level used already
// set: unvisited // leftover for choosing for next level to avoid repeat, dict + end
// List<List<String>> res; // result paths
// List<String, Queue<String>> adjMap; // adjacent directed graph,
// List<string, List<String>> works 2 ~! clone graph

private List<String> getNextLadder(String currLadder, Set<String> unVisited) {
    ArrayList<String> nextLadders = new ArrayList<String>();
    StringBuffer replace = new StringBuffer(currLadder); // mutate from here
    for(int i = 0; i < currLadder.length(); i++){
        char old = replace.charAt(i);
        for(char ch = 'a'; ch <= 'z'; ch++){
            if (currLadder.charAt(i) != ch)
                replace.setCharAt(i, ch);
            else continue;
            String replaced = replace.toString();
            if (unVisited.contains(replaced))
                nextLadders.add(replaced);
        }
        replace.setCharAt(i, old);
    }
    return nextLadders;
}

private void getLadders(String start, String end, LinkedList<String> path,
                        List<List<String>> res,
                        HashMap<String, Queue<String>> adjMap) {
    if (end.equals(start))      // end condition, len ?
        res.add(new ArrayList<String> (path));
    else {                      // else if (len > 0) {
        Queue<String> adjs = adjMap.get(end);
```

```
        for (String lad : adjs) {
            path.addFirst(lad); // path.add(0, lad); // equivallent
            getLadders(start, lad, path, res, adjMap);
            path.removeFirst(); // path.remove(0);   // remove ele at idx 0
        }
    }
}

public List<List<String>> findLadders(String start, String end, HashSet<String> dict) {
    HashMap<String, Queue<String>> adjMap = new HashMap<String, Queue<String>>();
    int currLen = 0;
    boolean found = false;
    List<List<String>> res = new ArrayList<List<String>>(); // results
    Queue<String> queue = new LinkedList<String>();          // queue for BFS
    Set<String> unv = new HashSet<String>(dict);             // unvstd words, delete f h
    Set<String> vst = new HashSet<String>();                 // check vstd during same l
    unv.add(end);       // dict may not contain end, so we add it~!@
    queue.offer(start); // search & build from start
    int currLev = 1;
    int nextLev = 0;
    for(String word : unv)    // all strings in dict + start
        adjMap.put(word, new LinkedList<String>());  // just like clone graph
    unv.remove(start);
    while (!queue.isEmpty()) {
        String currLadder = queue.poll();  // curr word, mutate, check dicts, add or no
        for(String nextLadder : getNextLadder(currLadder, unv)){
            if (vst.add(nextLadder)) {  // add to vst set, return true, succeed,
                nextLev++;              // vst set didn't contain it before
                queue.offer(nextLadder);
            }
            adjMap.get(nextLadder).offer(currLadder);
            if(nextLadder.equals(end) && !found) { // control if start-->end, No here
                found = true;
                currLen += 2;
            }
        }
        if(--currLev == 0){
            if(found) break;
            unv.removeAll(vst);  // remove all used string from cur level
            currLev = nextLev;   // to avoid next level repeat & infinite loop
            nextLev = 0;
            currLen++;
        }
    }
    if (found) {
        LinkedList<String> path = new LinkedList<String>();
        path.addFirst(end);     // add at beginning of list
        getLadders(start, end, path, res, adjMap);
    }
    return res;
}
```

此题改天再回来优化。。。。

10.3 other Covered

- Binary Tree Level Order Traversal 4.2.4
- Binary Tree Level Order Traversal II 4.2.5
- Clone Graph 18.1.1
- Binary Tree Zigzag Level Order Traversal 4.2.6
- Minimum Depth of Binary Tree 4.5.1

10.4 小结

10.4.1 适用场景

输入数据: 没什么特征, 不像深搜, 需要有“递归”的性质。如果是树或者图, 概率更大。
状态转换图: 树或者图。
求解目标: 求最短。

10.4.2 思考的步骤

1. 求路径长度, 还是路径本身 (或动作序列)?

   (a) 如果是求路径长度, 则状态里面要存路径长度

   (b) 如果是求路径本身或动作序列

   i. 要用一棵树存储宽搜过程中的路径

   ii. 是否可以预估状态个数的上限? 能够预估状态总数, 则开一个大数组, 用树的双亲表示法; 如果不能预估状态总数, 则要使用一棵通用的树。这一步也是第 4 步的必要不充分条件。

2. 如何表示状态? 即一个状态需要存储哪些些必要的数据, 才能够完整提供如何扩展到下一步状态的所有信息。一般记录当前位置或整体局面。

3. 如何扩展状态? 这一步跟第 2 步相关。状态里记录的数据不同, 扩展方法就不同。对于固定不变的数据结构 (一般题目直接给出, 作为输入数据), 如二叉树, 图等, 扩展方法很简单, 直接往下一层走, 对于隐式图, 要先在第 1 步里想清楚状态所带的数据, 想清楚了这点, 那如何扩展就很简单了。

4. 关于判重, 状态是否存在完美哈希方案? 即将状态一一映射到整数, 互相之间不会冲突。

   (a) 如果不存在, 则需要使用通用的哈希表 (自己实现或用标准库, 例如 unordered_set) 来判重; 自己实现哈希表的话, 如果能够预估状态个数的上限, 则可以开两个数组, head 和 next, 表示哈希表, 参考第 §?? 节方案 2。

   (b) 如果存在, 则可以开一个大布尔数组, 作为哈希表来判重, 且此时可以精确计算出状态总数, 而不仅仅是预估上限。

5. 目标状态是否已知? 如果题目已经给出了目标状态, 可以带来很大便利, 这时候可以从起始状态出发, 正向广搜; 也可以从目标状态出发, 逆向广搜; 也可以同时出发, 双向广搜。

Chapter 11

深度优先搜索 Depth First Search

11.1 Medium

11.1.1 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"]
  ]
```

在每一步都可以判断中间结果是否为合法结果, 用回溯法。
一个长度为 n 的字符串, 有 n+1 个地方可以砍断, 每个地方可断可不断, 前后两个隔板默认已经使用, 因此复杂度为 $O(2^{n-1})$.

1. 深搜: 时间复杂度 $O(2^n)$, 空间复杂度 O(n)

```
public void dfs(String s, List<String> path, List<List<String>> res) {
    if (s.length() == 0) {
        res.add(new ArrayList<String>(path));
        return;
    }
    for (int i = 0; i < s.length(); i++) {
        int bgn = 0;
        int end = i;
        while (bgn < end) {
            if (s.charAt(bgn) == s.charAt(end)) {
                bgn ++;
                end --;
            } else break;
        }
        if (bgn >= end) {
            path.add(s.substring(0, i + 1));
            dfs(s.substring(i + 1), path, res);
```

```
                path.remove(path.size() - 1);
            }
        }
        return;
    }

    public List<List<String>> partition(String s) {
        List<List<String>> res = new ArrayList<List<String>>();
        List<String> path = new ArrayList<String>();
        dfs(s, path, res);
        return res;
    }
```

2. 深搜 2: 时间复杂度 O(2^n), 空间复杂度 O(n)

   另一种写法, 更加简洁。这种写法也在 Combination Sum, Combination Sum II 中出现过。

3. 动态规划：时间复杂度 O(n^2), 空间复杂度 O(1)

   需要先消化一下别人的解法。。。。

11.1.2 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)
```

必须要走到底部才能判断解是否合法, 深搜。

需要优化自己的代码，写得太丑。。。

11.1.3 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:

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

小括号串是一个递归结构, 跟单链表、二叉树等递归结构一样, 首先想到用递归。

一步步构造字符串。当左括号出现次数 < n 时, 就可以放置新的左括号。当右括号出现次数小于左括号出现次数时, 就可以放置新的右括号。

1. 时间复杂度 O(TODO), 空间复杂度 O(n)

```
public void generate(List<String> res, int n, StringBuffer one, int left, int right
    StringBuffer tmp = new StringBuffer(one);
    if (left == n) {
        for (int i = 0; i < n - right; i++)
            tmp.append(")");
        res.add(tmp.toString());
        return;
    }
```

```
        generate(res, n, new StringBuffer(one).append("("), left + 1, right);
        if (left > right)
            generate(res, n, new StringBuffer(one).append(")"), left, right + 1);
    }

    public List<String> generateParenthesis(int n) {
        List<String> res = new ArrayList<String>();
        if (n > 0)
            generate(res, n, new StringBuffer(), 0, 0);
        return res;
    }
```

反过来写。

```
    public void generate(List<String> res, int n, StringBuffer tmp, int left, int right
        if (left < 0 || right < left) return;
        if (left == 0 && right == 0) {
            res.add(tmp.toString());
            return;
        } else {
            if (left > 0)
                generate(res, n, new StringBuffer(tmp).append("("), left - 1, right);
            if (right > left)
                generate(res, n, new StringBuffer(tmp).append(")"), left, right - 1);
        }
    }

    public List<String> generateParenthesis(int n) {
        List<String> res = new ArrayList<String>();
        if (n > 0)
            generate(res, n, new StringBuffer(), n, n);
        return res;
    }
```

不太确定这是真正正确的操作 StringBuffer 的途径还。

2. 另一种写法：

11.1.4 Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers added 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]
```

时间复杂度 O(n!), 空间复杂度 O(n).

```
public void dfs(int [] candidates, int bgn, int target,
                List<Integer> path, List<List<Integer>> res) {
    if (target == 0) {                              // 找到一个合法解
        res.add(new ArrayList<Integer> (path));
        return;
    }
    for (int i = bgn; i < candidates.length; i++) {    // 扩展状态
        if (candidates[i] > target) return;
        path.add(candidates[i]);       // 执行扩展动作
        dfs(candidates, i, target - candidates[i], path, res); // unlimited
        path.remove(path.size() - 1); // 撤销动作
    }
    return;
}

public List<List<Integer>> combinationSum(int[] candidates, int target) {
    Arrays.sort(candidates);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs(candidates, 0, target, path, res);
    return res;
}
```

11.1.5 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 added 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]
```

时间复杂度 O(n!), 空间复杂度 O(n).

```
public void dfs(int [] candidates, int bgn, int target,
                List<Integer> path, List<List<Integer>> res) {
    if (target == 0) {
        res.add(new ArrayList<Integer> (path));
        return;
    }
    int previous = -1;
    for (int i = bgn; i < candidates.length; i++) {
        //if (i > 0 && candidates[i] == candidates[i - 1]) cnt++;
        //else cnt = 1;
        //if (cnt == 1 || (path.size() >= cnt - 1
        //                 && path.get(path.size() - cnt + 1) == candidates[i])) {
        // 如果上一轮循环没有选 candidates[i则本次循环就不能再选], candidates[i],
        // 确保 candidates[i] 最多只用一次
        if (previous == candidates[i]) continue;
        if (candidates[i] > target) return;
        previous = candidates[i];
        path.add(candidates[i]);                                  // , cnt
        dfs(candidates, i + 1, target - candidates[i], path, res); // use once
        path.remove(path.size() - 1);
        // }
    }
    return;
}

public List<List<Integer>> combinationSum2(int[] candidates, int target) {
    Arrays.sort(candidates);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs(candidates, 0, target, path, res);
    return res;
}
```

11.2 Hard

11.2.1 N-Queens

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

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.."]
]
```

11.2.2 N-Queens II

Follow up for N-Queens problem.
Now, instead outputting board configurations, return the total number of distinct solutions.

11.2.3 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
时间复杂度 O(9^4), 空间复杂度 O(1).

```
public boolean isValidSudoku(char[][] bd, int x, int y) {
    for (int i = 0; i < 9; i++)
        if (i != x && bd[x][y] == bd[i][y]) return false;
    for (int j = 0; j < 9; j++)
        if (j != y && bd[x][y] == bd[x][j]) return false;
    for (int i = (x / 3) * 3; i < (x / 3 + 1) * 3; i++)
        for (int j = (y / 3) * 3; j < (y / 3 + 1) * 3; j++)
            if (i != x && j != y && bd[x][y] == bd[i][j])
                return false;
    return true;
}

public boolean solveSudoku(char[][] board) {
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            if (board[i][j] == '.') {
                for (int k = 0; k < 9; k++) {
                    board[i][j] = (char)('1' + k);
                    if (isValidSudoku(board, i, j) && solveSudoku(board))
                        return true;
                    board[i][j] = '.';
                }
                return false;
            }
    return true;
}
```

相当于是暴力法，基本上测试了所有的可能性，然后就每种可能性作有效性判断。(277 ms)

我希望执行的速度能够稍微快一点儿，所以每一步都执行了最优的一步，这样每一步都不用有效性判断，因为选的就是最好的，可速度还是更慢了。。。（589 ms）

```
public boolean helper(char[][] bd, List<Integer> list, int fst, List<Character> avb) {
    avb = new ArrayList<Character>();
    fst = getFirst(bd, list, avb);
    int val = list.get(fst);
    int x = list.get(fst) / 9;
    int y = list.get(fst) % 9;
    if (avb.size() == 0) return false;
    list.remove(fst);
    for (int k = 0; k < avb.size(); k++) {
        char c = avb.get(k);
        bd[x][y] = c;
        if (list.size() == 0) return true;
        if (helper(bd, new ArrayList<Integer>(list), fst, avb)) return true;
        bd[x][y] = '.';
    }
    list.add(new Integer(val));
    return false;
}

public int getFirst(char[][] bd, List<Integer> list, List<Character> res) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    Set<Character> tmp = new HashSet<Character>();  // for each pos
    Set<Character> bst = new HashSet<Character>();  // for each pos
    Set<Character> avb = new HashSet<Character>();
    int x, y; // row, col
    int result = 0;
    for (int i = 0; i < list.size(); i++) {
        tmp.clear();
        x = list.get(i) / 9;
        y = list.get(i) % 9;
        for (int j = 0; j < 9; j++) {
            if (bd[x][j] != '.') tmp.add(bd[x][j]);
            if (bd[j][y] != '.') tmp.add(bd[j][y]);
        }
        for (int j = 0; j < 3; j++)
            for (int k = 0; k < 3; k++)
                if (bd[j + (x / 3) * 3][k + (y / 3) * 3] != '.')
                    tmp.add(bd[j + (x / 3) * 3][k + (y / 3) * 3]);
        if (tmp.size() == 8) {
            for (char a = '1'; a <= '9'; a++)
                if (!tmp.contains(a)) res.add(a);
            return i;
        }
        if (tmp.size() > result) {
            result = i;
            bst = new HashSet<Character>(tmp);
        }
    }
```

```
    for (char i = '1'; i <= '9'; i++) {
        if (!bst.contains(i)) res.add(i);
    }
    return result;
}

public void solveSudoku(char[][] board) {
    List<Integer> list = new ArrayList<Integer>();
    List<Character> avb = new ArrayList<Character>();
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            if (board[i][j] == '.') list.add(i * 9 + j);
    int first = getFirst(board, list, avb);
    helper(board, list, first, avb);
}
```

看来解题还是简明地满足最基本的要求会比较好。

11.2.4 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.
```

深搜, 递归: 时间复杂度 $O(n^2*m^2)$, 空间复杂度 $O(n^2)$.

```
public boolean dfs(char[][] board, String word,
                   int i, int j, int idx, boolean [][] visited) {
    int m = board.length;
    int n = board[0].length;
    if (idx == word.length()) return true;   // 收敛条件

    if (i < 0 || i >= m || j < 0 || j >= n)
        return false;                       // 越界终止条件,
    if (visited[i][j]) return false;   // 已经访问过剪枝,
    if (word.charAt(idx) != board[i][j])
        return false;                        // 不相等剪枝,

    visited[i][j] = true;
    if (dfs(board, word, i - 1, j, idx + 1, visited)    // 上
        || dfs(board, word, i + 1, j, idx + 1, visited) // 下
        || dfs(board, word, i, j + 1, idx + 1, visited) // 左
```

```
            || dfs(board, word, i, j - 1, idx + 1, visited))// 右
            return true;
        visited[i][j] = false;
        return false;
    }

    public boolean exist(char[][] board, String word) {
        int m = board.length;
        int n = board[0].length;
        boolean [][] visited = new boolean[m][n];
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                if (dfs(board, word, i, j, 0, visited) == true) return true;
        return false;
    }
```

11.3 other Covered

- Same Tree 4.2.8
- Symmetric Tree 4.2.9
- Balanced Binary Tree 4.2.10
- Validate Binary Search Tree 4.4.1
- Convert Sorted List to Binary Search Tree 3.2.9
- Convert Sorted Array to Binary Search Tree 4.4.2
- Construct Binary Tree from Preorder and Inorder Traversal 4.3.1
- Construct Binary Tree from Inorder and Postorder Traversal 4.3.2
- Flatten Binary Tree to Linked List 4.2.11
- Populating Next Right Pointers in Each Node 4.5.6
- Populating Next Right Pointers in Each Node II 4.2.12
- Sum Root to Leaf Numbers 4.5.7
- Path Sum 4.5.3
- Path Sum II 4.5.4
- Maximum Depth of Binary Tree 4.5.2
- Recover Binary Search Tree 4.2.7
- Clone Graph 18.1.1
- Binary Tree Maximum Path Sum 4.5.5

11.4 小结

11.4.1 适用场景

输入数据: 如果是递归数据结构, 如单链表, 二叉树, 集合, 则百分之百可以用深搜; 如果是非递归数据结构, 如一维数组, 二维数组, 字符串, 图, 则概率小一些。

状态转换图: 树或者图。

求解目标: 必须要走到最深 (例如对于树, 必须要走到叶子节点) 才能得到一个解, 这种情况适合用深搜。

11.4.2 思考的步骤

1. 是求路径条数, 还是路径本身 (或动作序列)? 深搜最常见的三个问题, 求可行解的总数, 求一个可行解, 求所有可行解。

   (a) 如果是求路径本身, 则要用一个数组 path[] 存储路径。跟宽搜不同, 宽搜虽然最终求的也是一条路径, 但是需要存储扩展过程中的所有路径, 在没找到答案之前所有路径都不能放弃; 而深搜, 在搜索过程中始终只有一条路径, 因此用一个数组就足够了。

   (b) 如果是路径条数, 则不需要存储路径。

2. 只要求一个解, 还是要求所有解? 如果只要求一个解, 那找到一个就可以返回; 如果要求所有解, 找到了一个后, 还要继续扩展, 直到遍历完。广搜一般只要求一个解, 因而不需要考虑这个问题 (广搜当然也可以求所有解, 这时需要扩展到所有叶子节点, 相当于在内存中存储整个状态转换图, 非常占内存, 因此广搜不适合解这类问题)。

3. 如何表示状态? 即一个状态需要存储哪些些必要的数据, 才能够完整提供如何扩展到下一步状态的所有信息。跟广搜不同, 深搜的惯用写法, 不是把数据记录在状态 struct 里, 而是添加函数参数 (有时为了节省递归堆栈, 用全局变量), struct 里的字段与函数参数一一对应。

4. 如何扩展状态? 这一步跟上一步相关。状态里记录的数据不同, 扩展方法就不同。对于固定不变的数据结构 (一般题目直接给出, 作为输入数据), 如二叉树, 图等, 扩展方法很简单, 直接往下一层走, 对于隐式图, 要先在第 1 步里想清楚状态所带的数据, 想清楚了这点, 那如何扩展就很简单了。

5. 关于判重 (a) 如果状态转换图是一棵树, 则不需要判重, 因为在遍历过程中不可能重复。(b) 如果状态转换图是一个图, 则需要判重, 方法跟广搜相同, 见第 §9.4 节。这里跟第 8 步中的加缓存是相同的, 如果有重叠子问题, 则需要判重, 此时加缓存自然也是有效果的。

6. 终止条件是什么? 终止条件是指到了不能扩展的末端节点。对于树, 是叶子节点, 对于图或隐式图, 是出度为 0 的节点。

7. 收敛条件是什么? 收敛条件是指找到了一个合法解的时刻。如果是正向深搜 (父状态处理完了才进行递归, 即父状态不依赖子状态, 递归语句一定是在最后, 尾递归), 则是指是否达到目标状态; 如果是逆向深搜 (处理父状态时需要先知道子状态的结果, 此时递归语句不在最后), 则是指是否到达初始状态。

   由于很多时候终止条件和收敛条件是是合二为一的, 因此很多人不区分这两种条件。仔细区分这两种条件, 还是很有必要的。

   为了判断是否到了收敛条件, 要在函数接口里用一个参数记录当前的位置 (或距离目标还有多远)。如果是求一个解, 直接返回这个解; 如果是求所有解, 要在这里收集解, 即把第一步中表示路径的数组 path[] 复制到解集合里。

8. 如何加速?

   (a) 剪枝。深搜一定要好好考虑怎么剪枝, 成本小收益大, 加几行代码, 就能大大加速。这里没有通用方法, 只能具体问题具体分析, 要充分观察, 充分利用各种信息来剪枝, 在中间节点提前返回。

   (b) 缓存。如果子问题的解会被重复利用, 可以考虑使用缓存。

   i. 前提条件: 子问题的解会被重复利用, 即子问题之间的依赖关系是有向无环图 (DAG)。如果依赖关系是树状的 (例如树, 单链表) , 没必要加缓存, 因为子问题只会一层层往下, 用一次就再也不会用到, 加了缓存也没什么加速效果。

   ii. 具体实现: 可以用数组或 HashMap。维度简单的, 用数组; 维度复杂的, 用 HashMap, C++ 有 map, C++ 11 以后有 unordered$_{map}$, 比 map 快。

   拿到一个题目, 当感觉它适合用深搜解决时, 在心里面把上面 8 个问题默默回答一遍, 代码基本上就能写出来了。对于树, 不需要回答第 5 和第 8 个问题。如果读者对上面的经验总结看不懂或感觉“不实用”, 很正常, 因为这些经验总结是笔者做了很多深搜题后总结出来的, 从思维的发展过程看, “经验总结”要晚于感性认识, 所以这时候建议读者先做做后面的题目, 积累一定的感性认识后, 在回过头来看这一节的总结, 相信会和笔者有共鸣。

11.4.3 代码模板

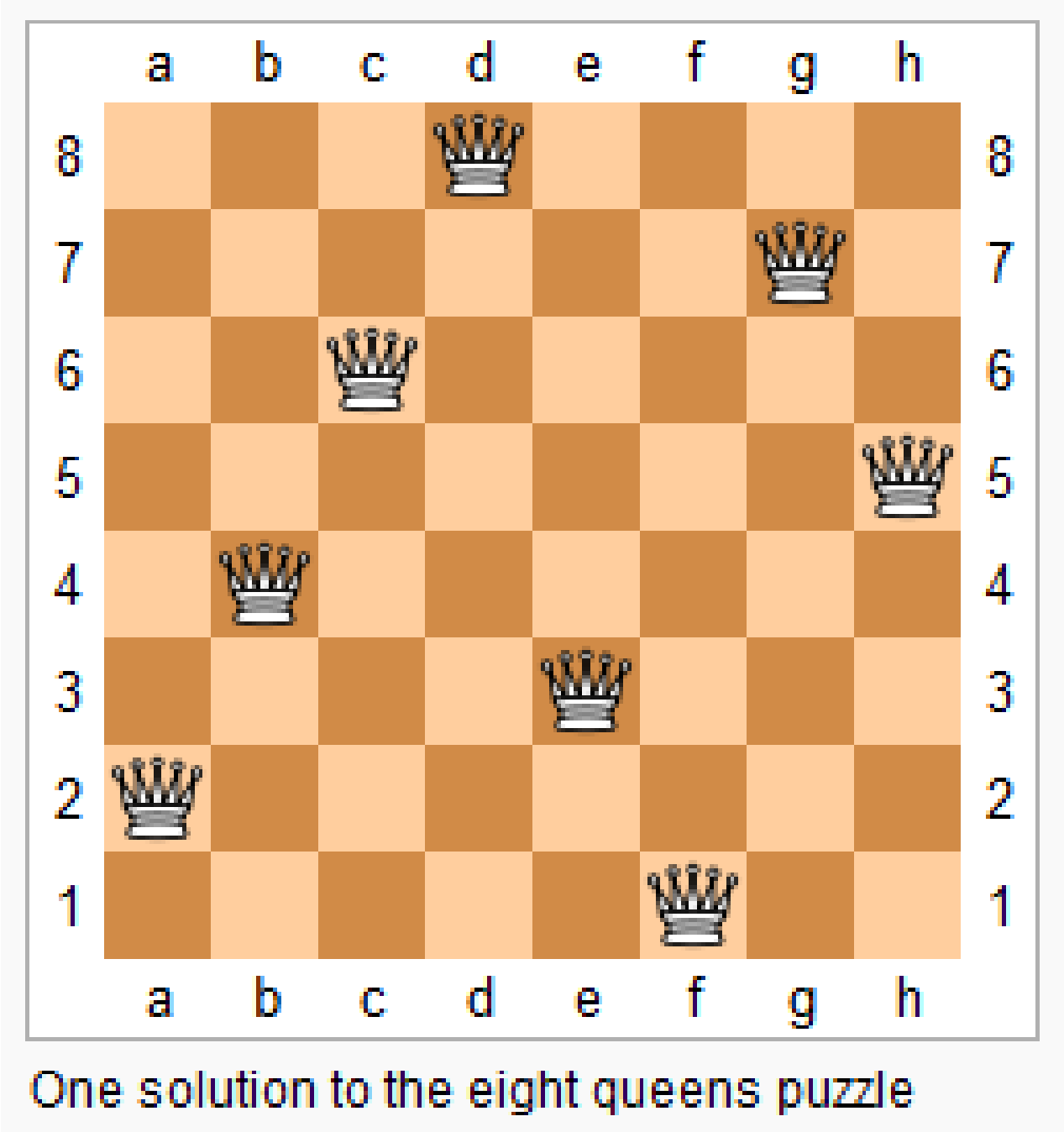


Figure 11.1: N-Queens

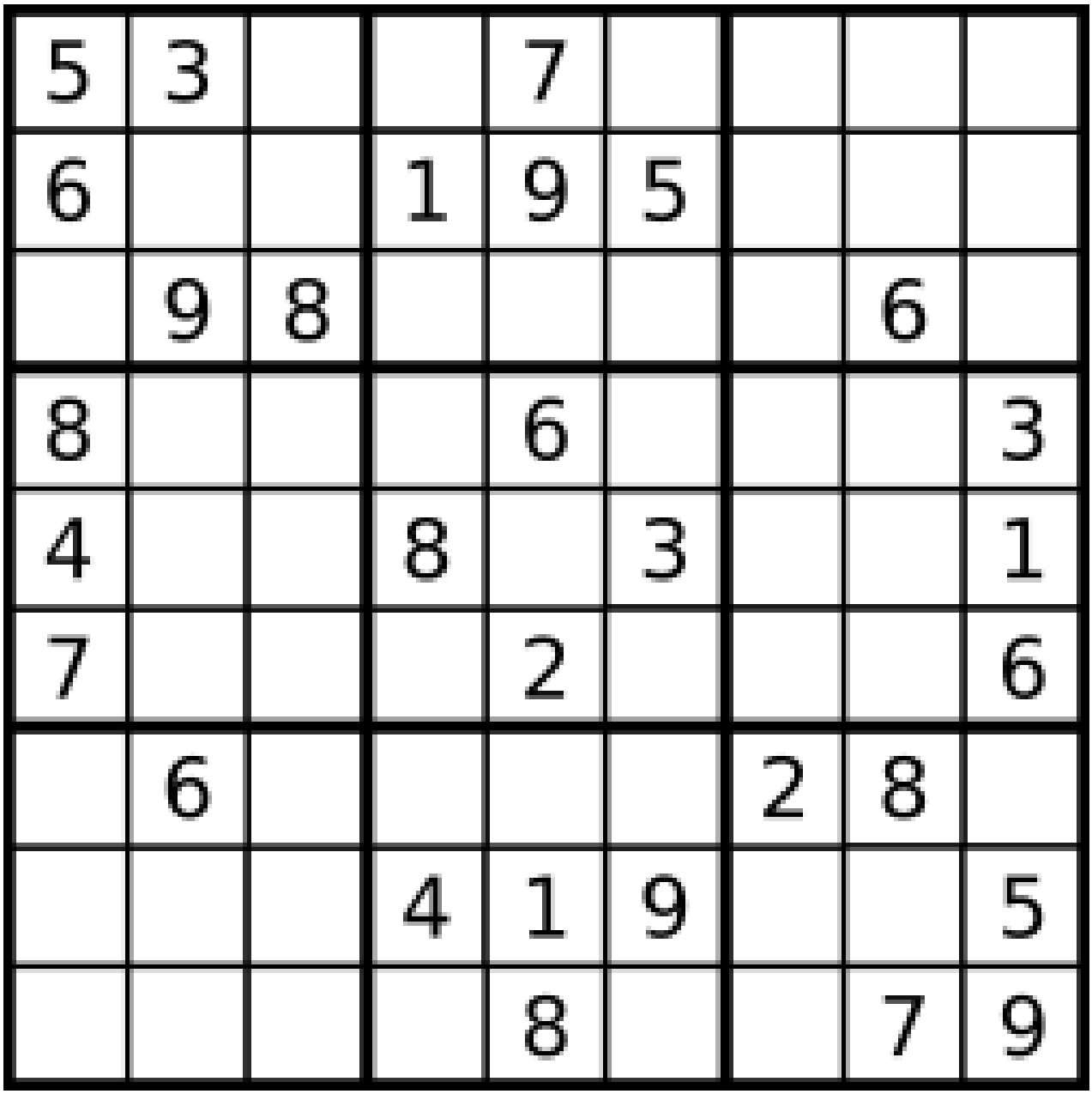


Figure 11.2: Sudoku Solver 1

Figure 11.3: Sudoku Solver 2

Chapter 12

分治法 Divide and Conqure

12.1 Medium

12.1.1 Pow(x, n)

Implement pow(x, n).
Tags: Math, Binary Search
二分法, $x^n = x^{n/2} \times x^{n/2} \times x^{n\%2}$
时间复杂度 O(logn), 空间复杂度 O(1).

```
public double myPower(double x, int n) {
    if (n == 0) return 1;
    double val = myPower(x, n / 2);
    if (n % 2 == 0) return val * val;
    else return val * val * x;
}

public double pow(double x, int n) {
    if (n < 0) return 1.0 / myPower(x, -n);
    else return myPower(x, n);
}
```

12.1.2 Sqrt(x)

Implement int sqrt(int x).
Compute and return the square root of x.
Tags: Math, Binary Search
二分查找
时间复杂度 O(logn), 空间复杂度 O(1).

```
public int sqrt(int x) {
    int bgn = 0, end = 0;
    int mid = 0;
    int lastMid = 0;
    if (x < 2) return x;
    // end = x / 2;          // which one is better ?
    end = x / 2 < (int)Math.sqrt(Integer.MAX_VALUE) ?
        x / 2 + 1 : (int)Math.sqrt(Integer.MAX_VALUE);
```

```
    while (bgn <= end) {
        mid = bgn + (end - bgn) / 2;
        if (x / mid == mid)  // 不要用 x > mid * mid, 会溢出
            return mid;
        else if (x / mid > mid) {
            bgn = mid + 1;
            lastMid = mid;
        } else  // x / mid < mid
            end = mid - 1;
    }
    return lastMid;
}
```

12.2 other Covered

- Majority Element 1.4.1
- Maximum Subarray `Maximum Subarray`
- Merge K sorted List `Merge K sorted List`

Chapter 13

贪心法 Greedy Search

13.1 Medium

13.1.1 Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:

The solution is guaranteed to be unique.

13.1.2 Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

贪心法, 分别找到价格最低和最高的一天, 低进高出, 注意最低的一天要在最高的一天之前。把原始价格序列变成差分序列, 本题也可以做是最大 m 子段和, m = 1。

时间复杂度 O(n), 空间复杂度 O(1).

```
public int maxProfit(int[] prices) {
    int lowest = Integer.MAX_VALUE;
    int maxProfit = 0;
    for (int i = 0; i < prices.length; i++) {
        if (prices[i] < lowest) lowest = prices[i];
        else maxProfit = Math.max(maxProfit, prices[i] - lowest);
    }
    return maxProfit;
}
```

13.1.3 Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

贪心法, 低进高出, 把所有正的价格差价相加起来。把原始价格序列变成差分序列, 本题也可以做是最大 m 子段和, m = 数组长度。

```
public int maxProfit(int[] prices) {
    int result = 0;
    for (int i = 0; i < prices.length-1; i++)
        result += (prices[i] < prices[i+1]) ? prices[i + 1] - prices[i] : 0;
    return result;
}
```

13.1.4 Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you are able to reach the last index.
For example:
A = [2, 3, 1, 1, 4], return true.
A = [3, 2, 1, 0, 4], return false.
由于每层最多可以跳 A[i] 步, 也可以跳 0 或 1 步, 因此如果能到达最高层, 则说明每一层都可以到达。有了这个条件, 说明可以用贪心法。
思路一: 正向, 从 0 出发, 一层一层网上跳, 看最后能不能超过最高层, 能超过, 说明能到达, 否则不能到达。
思路二: 逆向, 从最高层下楼梯, 一层一层下降, 看最后能不能下降到第 0 层。
思路三: 如果不敢用贪心, 可以用动规, 设状态为 f[i], 表示从第 0 层出发, 走到 A[i] 时剩余的最大步数, 则状态转移方程为:

```
f[i] = max(f[i - 1], A[i - 1]) - 1, i > 0
```

1. 思路 1, 时间复杂度 O(n), 空间复杂度 O(1)

```
public boolean canJump(int[] a) {
    int reach = 1;      // 最右能跳到哪里
    for (int i = 0; i < reach && reach < a.length; i++)
        reach = Math.max(reach, i + 1 + a[i]);
    return reach >= a.length;
}
```

2. 思路 2, 时间复杂度 O(n), 空间复杂度 O(1)

```
public boolean canJump(int[] a) {
    if (a.length == 0) return true;
    // 逆向下楼梯, 最左能下降到第几层
    int leftMost = a.length - 1;
    for (int i = a.length - 2; i >= 0; i--)
        if (i + a[i] >= leftMost)
            leftMost = i;
    return leftMost == 0;
}
```

3. 思路三, 动规, 时间复杂度 O(n), 空间复杂度 O(n)

```
public boolean canJump(int[] a) {
    int f[] = new int[a.length];
    for (int i = 1; i < a.length; i++) {
        f[i] = Math.max(f[i - 1], a[i - 1]) - 1;
        if (f[i] < 0) return false;
    }
    return f[a.length - 1] >= 0;
}
```

13.2 Hard

13.2.1 Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.
For example:
Given array A = [2, 3, 1, 1, 4]
The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

1. 思路 1：时间复杂度 O(n), 空间复杂度 O(1)
2. 思路 2：时间复杂度 O(n), 空间复杂度 O(1)

13.2.2 Candy

There are N children standing in a line. Each child is assigned a rating value.
You are giving candies to these children subjected to the following requirements:
Each child must have at least one candy.
Children with a higher rating get more candies than their neighbors.
What is the minimum candies you must give?

13.3 other Covered

- Wildcard Matching 14.3.10

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

Most solutions are DP with runtime O(n) and O(1) space, 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 ((0, 1), (1, 1)) ^ (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

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

这里 Note 的含义是: 如果数值过大会溢出的。–《靖空间》

方法论总结：遇上新颖没见过的题目：

- 查看其特征；
- 搜索大脑有什么熟悉的题目，可以套的，或者相似的可以退到出来的；
- 罗列出来这些题目：全排列，水槽，TwoSum，等等；
- 搜索大脑里面有什么熟悉的算法；
- 罗列出来：动态规划法，递归，回溯，二叉树遍历，贪心法，递归树，分治法，观察特征计算法，等等，肯定有可以套的算法的！
- 最后选定算法解题；
- 没有优化的算法，可以使用暴力法，先解决再说！

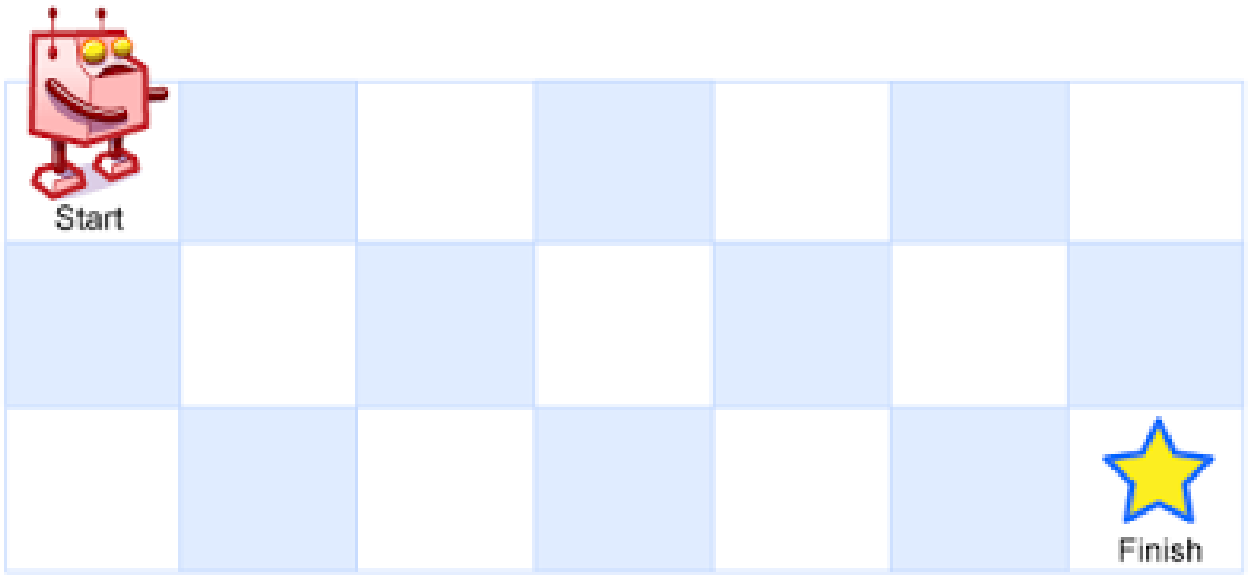


Figure 14.1: Unique Path

最后实在没招了，想办法让人提示，主要是提示用什么算法，能给个思路更好了，O(∩)O~

1. 深搜，递归法

   深搜, 小集合可以过, 大集合会超时；时间复杂度 $O(n^4)$, 空间复杂度 $O(n)$；

```
public int uniquePaths(int m, int n) {
    if (m < 1 || n < 1) return 0;   // 终止条件
    if (m == 1 && n == 1) return 1; // 收敛条件
    return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
}
```

2. 备忘录法: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

   给前面的深搜, 加个缓存, 就可以过大集合了。即备忘录法。

```
// 深搜 + 缓存即备忘录法,
// 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
    private int [][] f;     // 缓存

    public int dfs(int x, int y) {
        if (x < 1 || y < 1) return 0;    // 数据非法终止条件,
        if (x == 1 && y == 1) return 1; // 回到起点收敛条件,
        return getOrUpdate(x - 1, y) + getOrUpdate(x, y - 1);
    }

    public int getOrUpdate(int x, int y) {
        if (f[x][y] > 0) return f[x][y];
        else return f[x][y] = dfs(x, y);
    }

    public int uniquePaths(int m, int n) {
        // 0 行和 0 列未使用
```

```
            f = new int[m + 1][n + 1]; // 缓存
            return dfs(m, n);
        }
    }
```

3. 动态规划

   算法 1 的递归解法中，其实我们计算了很多重复的子问题，比如计算 uniquePaths(4, 5) 和 uniquePaths(5, 3) 时都要计算子问题 uniquePaths(3,2)，再者由于 uniquePaths(m, n) = uniquePaths(n, m)，这也使得许多子问题被重复计算了。

   要保存子问题的状态，这样很自然的就想到了动态规划方法，设 dp[i][j] = uniquePaths(i, j)，那么动态规划方程为：

   动态方程: dp[i][j] = dp[i-1][j] + dp[i][j-1]

   边界条件：dp[i][1] = 1, dp[1][j] = 1

```
public int uniquePaths(int m, int n) {
    if (m == 0 || n == 0) return 1;
    else if (m == 1 && n == 1) return 1;

    int [][] dp = new int[m][n];
    for (int i = 0; i < n; i++)
        dp[0][i] = 1;
    for (int i = 0; i < m; i++)
        dp[i][0] = 1;
    dp[0][0] = 0;
    for (int i = 1; i < m; i++)
        for (int j = 1; j < n; j++)
            dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
    return dp[m-1][n-1];
}
```

   滚动数组

```
// 动规滚动数组,
// 时间复杂度 O(n^2), 空间复杂度 O(n)
int uniquePaths(int m, int n) {
    vector<int> f(n, 0);
    f[0] = 1;
    for (int i = 0; i < m; i++) {
        for (int j = 1; j < n; j++) {
            // 左边的 f[j], 表示更新后的 f[j], 与公式中的 f[i[[j] 对应
            // 右边的 f[j], 表示老的 f[j], 与公式中的 f[i-1][j] 对应
            f[j] = f[j - 1] + f[j];
        }
    }
    return f[n - 1];
}
```

4. 组合数（C_{m+n-2}^{m-1}）

一个 m 行, n 列的矩阵, 机器人从左上走到右下总共需要的步数是 m + n - 2, 其中向下走的步数是 m - 1, 因此问题变成了在 m + n - 2 个操作中, 选择 m - 1 个时间点向下走, 选择方式有多少种。即 C_{m+n-2}^{m-1}。这里需要注意的是求组合数时防止乘法溢出。

```
public int combination(int a, int b) {
    if (b > (a >>> 1)) b = a - b;
    long res = 1;
    for (int i = 1; i <= b ; i++)
        res = res * (a - i + 1) / i;
    return (int)res;
}

public int uniquePaths(int m, int n) {
    return combination(m + n - 2, m - 1);
}
```

14.2.2 Unique Path with Obstacles

Follow up for "Unique Paths":
Now consider if some obstacles are added to the grids. How many unique paths would there be?
An obstacle and empty space is marked as 1 and 0 respectively in the grid.
For example, There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.
Note: m and n will be at most 100.

1. 备忘录法

```
// LeetCode, Unique Paths II
// 深搜 + 缓存, 即备忘录法
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();
        // 0 行和 0 列未使用
        this->f = vector<vector<int> >(m + 1, vector<int>(n + 1, 0));
        return dfs(obstacleGrid, m, n);
    }

private:
    vector<vector<int> > f;                    // 缓存
    int dfs(const vector<vector<int> > &obstacleGrid,
            int x, int y) {
        if (x < 1 || y < 1) return 0;          // 数据非法终止条件,
```

```
        if (obstacleGrid[x-1][y-1]) return 0; // (x,y) 是障碍
        if (x == 1 and y == 1) return 1;       // 回到起点收敛条件,
        return getOrUpdate(obstacleGrid, x - 1, y) +
                getOrUpdate(obstacleGrid, x, y - 1);
    }
    int getOrUpdate(const vector<vector<int> > &obstacleGrid,
                    int x, int y) {
        if (f[x][y] > 0) return f[x][y];
        else return f[x][y] = dfs(obstacleGrid, x, y);
    }
};
```

2. 动态规划

   与上一题类似, 但要特别注意第一列的障碍。在上一题中, 第一列全部是 1, 但是在这一题中不同, 第一列如果某一行有障碍物, 那么后面的行应该为 0。

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    int n = obstacleGrid[0].length;
    if (m == 0 || n == 0) return 0;
    else if (obstacleGrid[0][0] == 1 || obstacleGrid[m - 1][n - 1] == 1) return 0;
    else if (m == 1 || n == 1) return obstacleGrid[m-1][n-1] == 1 ? 0 : 1;

    int [][] arr = new int[m][n];
    for (int i = 0; i < n; i++)
        if (obstacleGrid[0][i] == 1) {
            for (int j = i; j < n; j++)
                arr[0][j] = 0;
            i = n;
        } else arr[0][i] = 1;

    for (int i = 0; i < m; i++)
        if (obstacleGrid[i][0] != 1)
            arr[i][0] = 1;
        else
            for (int j = i; j < m; j++) {
                arr[j][0] = 0;
                i = m;
            }

    arr[0][0] = 0;
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (obstacleGrid[i][j] == 1)
                arr[i][j] = 0;
            else
                arr[i][j] = arr[i][j - 1] + arr[i - 1][j];
        }
    }
    return arr[m-1][n-1];
}
```

14.2.3 Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

跟 Unique Paths (见 §10.2) 很类似。设状态为 f[i][j], 表示从起点 (0, 0) 到达 (i, j) 的最小路径和, 则状态转移方程为:

转移方程: f[i][j] = min(f[i-1][j], f[i][j-1]) + grid[i][j]

1. 备忘录法

   代码应该可以再优化一下。

```
private int [][] f;

private int getOrUpdate(int [][] grid, int x, int y) {
    if (x < 0 || y < 0) return Integer.MAX_VALUE;
    if (f[x][y] >= 0) return f[x][y];
    else {
        f[x][y] = dfs(grid, x, y);
        return f[x][y];
    }
}

private int dfs(int [][] grid, int x, int y) {
    if (x < 0 || y < 0) return Integer.MAX_VALUE;
    if (x == 0 && y == 0) return grid[0][0];
    return Math.min(getOrUpdate(grid, x - 1, y),
                    getOrUpdate(grid, x, y - 1)) + grid[x][y];
}

public int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    f = new int[m][n];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            f[i][j] = -1;
    return dfs(grid, m - 1, n - 1);
}
```

2. 动态规划

```
public int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    if (m == 0 || n == 0) return 0;

    int [][] f = new int[m][n];
    f[0][0] = grid[0][0];
    for (int i = 1; i < n; i++)
        f[0][i] = f[0][i - 1] + grid[0][i];
    for (int i = 1; i < m; i++)
```

```
            f[i][0] = f[i - 1][0] + grid[i][0];

        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                f[i][j] = Math.min(f[i][j - 1], f[i - 1][j]) + grid[i][j];
        return f[m-1][n-1];
    }
```

3. 动态规划 + 滚动数组

14.2.4 Unique Binary Search Tree

Given n, how many structurally unique BST's (binary search trees) that store values 1⋯n?
For example, Given n = 3, there are a total of 5 unique BST's.

```
1          3     3      2      1
 \        /     /      / \      \
  3      2     1      1   3      2
 /      /       \                 \
2      1         2                 3
```

如果把上例的顺序改一下, 就可以看出规律了。《水中的鱼》

```
1            1                  2                  3            3
 \            \                / \                /            /
  3            2              1   3              2            1
 /              \                                /              \
2                3                              1                2
```

比如，以 1 为根的树有几个，完全取决于有二个元素的子树有几种。同理，2 为根的子树取决于一个元素的子树有几个。以 3 为根的情况，则与 1 相同。
定义 Count[i] 为以 [0,i] 能产生的 Unique Binary Tree 的数目，
如果数组为空，毫无疑问，只有一种 BST，即空树，Count[0] =1
如果数组仅有一个元素 {1}，只有一种 BST，单个节点, Count[1] = 1
如果数组有两个元素 {1,2}，那么有如下两种可能

```
1               2
 \             /
  2           1
```

```
Count[2] = Count[0] * Count[1]    ( 1 为根的情况 )
         + Count[1] * Count[0]    ( 2 为根的情况 )
```

再看一遍三个元素的数组，可以发现 BST 的取值方式如下：

```
Count[3] = Count[0]*Count[2]   ( 1 为根的情况 )
         + Count[1]*Count[1]   ( 2 为根的情况 )
         + Count[2]*Count[0]   ( 3 为根的情况 )
```

所以，由此观察，当数组为 1, 2, 3, ⋯, n 时, 基于以下原则的构建的 BST 树具有唯一性: 以 i 为根节点的树, 其左子树由 [1, i-1] 构成, 其右子树由 [i+1, n] 构成。
可以得出 Count 的递推公式为
Count(i) = $\sum_{k=1}^{i}$ Count(k - 1) * Count(i - k)
问题至此划归为一维动态规划。

```
public int numTrees(int n) {
    int [] cnt = new int[n + 1];
    cnt[0] = 1; // empty element
    cnt[1] = 1; // one element
    for (int i = 2; i <= n ; i++)
        for (int j = 0; j < i; j++)
            cnt[i] += cnt[j]*cnt[i - 1 - j];
    return cnt[n];
}
```

14.2.5 Unique Binary Search Tree II

Given n, generate all structurally unique BST's (binary search trees) that store values 1⋯n.
For example,
Given n = 3, your program should return all 5 unique BST's shown below.

```
1          3     3      2      1
 \        /     /      / \      \
  3     2     1      1   3      2
 /     /       \                 \
2     1         2                 3
```

```
public List<TreeNode> helper(int bgn, int end) {
    List<TreeNode> res = new ArrayList<TreeNode>();
    if (bgn > end) {
        res.add(null);  // important
        return res;
    }

    List<TreeNode> left = null;
    List<TreeNode> right = null;
    TreeNode root = null;
    for (int i = bgn; i <= end ; i++) {
        left = helper(bgn, i - 1);
        right = helper(i + 1, end);
        for (int j = 0; j < left.size(); j++) {
            for (int k = 0; k < right.size(); k++) {
                root = new TreeNode(i);
                root.left = left.get(j);
                root.right = right.get(k);
                res.add(root);
                root = null;
            }
        }
        left = null;
        right = null;
    }
    return res;
}

public List<TreeNode> generateTrees(int n) {
```

```
    return helper(1, n);
}
```

14.2.6 Maximum Sum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [-2, 1, -3, 4, -1, 2, 1, -5, 4],
the contiguous subarray [4, -1, 2, 1] has the largest sum = 6.
click to show more practice.

More practice:

If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Tags: Divide and Conquer, Array, Dynamic Programming

分析:

最大连续子序列和, 非常经典的题。

当我们从头到尾遍历这个数组的时候, 对于数组里的一个整数, 它有几种选择呢? 它只有两种选择: 1、加入之前的 SubArray; 2. 自己另起一个 SubArray。那什么时候会出现这两种情况呢?

- 如果之前 SubArray 的总体和大于 0 的话, 我们认为其对后续结果是有贡献的。这种情况下我们选择加入之前的 SubArray
- 如果之前 SubArray 的总体和为 0 或者小于 0 的话, 我们认为其对后续结果是没有贡献, 甚至是有害的 (小于 0 时)。这种情况下我们选择以这个数字开始, 另起一个 SubArray。

设状态为 f[j], 表示以 S[j] 结尾的最大连续子序列和, 则状态转移方程如下:

```
f[j] = max (f[j-1] + S[j], S[j]) , 其中 1 ≤ j ≤ n
target = max (f[j]) , 其中 1 ≤ j ≤ n
```

解释如下:

情况一, S[j] 不独立, 与前面的某些数组成一个连续子序列, 则最大连续子序列和为 f[j-1] + S[j]。

情况二, S[j] 独立划分成为一段, 即连续子序列仅包含一个数 S[j], 则最大连续子序列和为 S[j]。

其他思路:

- 思路 2: 直接在 i 到 j 之间暴力枚举, 复杂度是 $O(n^3)$
- 思路 3: 处理后枚举, 连续子序列的和等于两个前缀和之差, 复杂度 $O(n^2)$。
- 思路 4: 分治法, 把序列分为两段, 分别求最大连续子序列和, 然后归并, 复杂度 O(n*(log(n)))
- 思路 5: 把思路 2 $O(n^2)$ 的代码稍作处理, 得到 O(n) 的算法
- 思路 6: 当成 M = 1 的最大 M 子段和

1. 动态规划：时间复杂度 O(n), 空间复杂度 O(1)

```
public int maxSubArray(int[] a) {
    int result = Integer.MIN_VALUE, f = 0;
    for (int i = 0; i < a.length; i++) {
        f = Math.max(f + a[i], a[i]);
        result = Math.max(result, f);
    }
    return result;
}
```

2. 思路 5: 时间复杂度 O(n), 空间复杂度 O(n)

```
public int maxSubArray(int[] a) {
    int result, curMin;
    int [] sum = new int[a.length + 1];
    sum[0] = 0;
    result = Integer.MIN_VALUE;
    curMin = sum[0];
    for (int i = 1; i <= a.length; i++)
        sum[i] = sum[i - 1] + a[i - 1];
    for (int i = 1; i <= a.length; i++) {
        result = Math.max(result, sum[i] - curMin);
        curMin = Math.min(curMin, sum[i]);
    }
    return result;
}
```

14.2.7 Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2, 3, -2, 4],
the contiguous subarray [2, 3] has the largest product = 6.

这道题跟14.2.6模型上和思路上都比较类似，还是用一维动态规划中的“局部最优和全局最优法”。

这里的区别是维护一个局部最优不足以求得后面的全局最优，这是由于乘法的性质不像加法那样，累加结果只要是正的一定是递增，乘法中有可能现在看起来小的一个负数，后面跟另一个负数相乘就会得到最大的乘积。

不过事实上也没有麻烦很多，我们只需要在维护一个局部最大的同时，在维护一个局部最小，这样如果下一个元素遇到负数时，就有可能与这个最小相乘得到当前最大的乘积和，这也是利用乘法的性质得到的。

来自《Code Ganker 征服代码》

```
public int maxProduct(int[] a) {
    int n = a.length;
    int maxLocal = a[0];
    int minLocal = a[0];
    int res = a[0];
    for (int i = 1; i < n; i++) {
        int tmp = maxLocal;
        maxLocal = Math.max(Math.max(maxLocal * a[i], a[i]),
                            Math.max(minLocal * a[i], a[i]));
        minLocal = Math.min(Math.min(a[i], a[i] * tmp),
                            Math.min(a[i], a[i] * minLocal));
        res = Math.max(res, maxLocal);
    }
    return res;
}
```

14.2.8 Decode Ways, bug~ !!

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.
For example,
Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).
The number of ways decoding "12" is 2.
参考 Climb stairs, 14.1.1
最原始型的 follow、控制 logic:

```
public int numDecodings(String s) {
    if (s == null || s.length() == 0) return 0;
    else if (s.charAt(0) == '0') return 0;
    else if (s.length() == 1) return 1;
    else if (s.length() >= 2 && s.charAt(0) - '0' > 2 && s.charAt(1) == '0') return 0;

    int j = 0; // idx for s
    char c;    // char at pos j
    int [] res = new int[s.length()];
    res[0] = 1;
    res[1] = 1;
    if (s.charAt(1) != '0'
        && ( (s.charAt(0) == '1'
              || (s.charAt(0) == '2') && s.charAt(1) - '0' < 7)))
        res[1] = 2;

    for (int i = 2; i < s.length(); i++) {
        c = s.charAt(i);
        if (c != '0')
            res[i] += res[i - 1]; // separate dijit // except '0', NO, 10, 20 ok

        switch (c) {
        case '0':
            if (s.charAt(i - 1) == '1' || s.charAt(i - 1) == '2')
                if (s.charAt(i - 2) == '1' || s.charAt(i - 2) == '2')
                    res[i] = res[i - 2];     // 110 = 1 10, ... 11 0, NO~!
                else
                    res[i] = res[i - 1];
            else return 0;
            break;
        case '7':     // 17, 18, 19
        case '8':
        case '9':
            if (s.charAt(i - 1) == '1')
                res[i] += res[i - 2];
            break;
        default:     // 1, 2, 3, 4, 5, 6: Individual, 1-pre, 2-pre, 11, 21, 12, 22 ...
            if (s.charAt(i - 1) == '1' || s.charAt(i - 1) == '2')
                res[i] += res[i - 2];
            break;
```

```
        }
    }
    return res[s.length() - 1];
}
```

1. 动态规划, 时间复杂度 O(n), 空间复杂度 O(1), java v m bug ~~!!!!

   模拟 pdf c++ 版大牛精华版代码：

```
// my Java version code has bug here, fix it later ~~~
public int numDecodings(String s) {
    if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;
    int prev = 0;
    int curr = 1;
    int tmp = 0;
    for (int i = 1; i <= s.length() ; i++) {
        if (s.charAt(i - 1) == '0') curr = 0;
        if (i < 2 || s.charAt(i - 2) != '1' ||
            (s.charAt(i - 2) == '2' && s.charAt(i - 1) <= '6'))
            prev = 0;
        tmp = curr;
        curr = prev + curr;
        prev = tmp;
    }
    return curr;
}
```

14.2.9 Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note: Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

1. 自顶向下：时间复杂度是 $O(n^2)$，空间复杂度是 O(n)

   这是一道动态规划的题目，求一个三角形二维数组从顶到低端的最小路径和。思路是维护到某一个元素的最小路径和，那么在某一个元素 i，j 的最小路径和就是它上层对应的相邻两个元素的最小路径和加上自己的值，递推式是

   sum[i][j] = min(sum[i-1][j-1],sum[i-1][j]) + triangle[i][j]。

   最后扫描一遍最后一层的路径和，取出最小的即可。每个元素需要维护一次，总共有 1+2+…+n = n*(n+1)/2 个元素，所以时间复杂度是 $O(n^2)$。而空间上每次只需维护一层即可（因为当前层只用到上一层的元素），所以空间复杂度是 O(n)。

来自《Code Ganker 征服代码》

```
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
    if(triangle == null || triangle.size() == 0)
        return 0;
    if(triangle.size() == 1)
        return triangle.get(0).get(0);

    int[] dp = new int[triangle.size()];
    dp[0] = triangle.get(0).get(0);
    for(int i = 1; i < triangle.size(); i++) {
        dp[i] = dp[i - 1] + triangle.get(i).get(i);
        for(int j = i - 1 ; j >= 1; j--)
            dp[j] = (dp[j] < dp[j-1] ? dp[j] : dp[j - 1]) + triangle.get(i).get(j);
        dp[0] = dp[0] + triangle.get(i).get(0);
    }

    int minimum = dp[0];
    for(int i = 1; i < dp.length; i++)
        if (dp[i] < minimum)
            minimum = dp[i];
    return minimum;
}
```

上述代码实现时要注意每层第一个和最后一个元素特殊处理一下。

2. 自底向上：

换个角度考虑一下，如果这道题不自顶向下进行动态规划，而是放过来自底向上来规划，递归式只是变成下一层对应的相邻两个元素的最小路径和加上自己的值，原理和上面的方法是相同的，这样考虑到优势在于不需要最后对于所有路径找最小的，而且第一个元素和最后元素也不需要单独处理了，所以代码简洁了很多。

```
public int minimumTotal(List<List<Integer>> triangle) {
    List<Integer> res = new ArrayList<Integer>(triangle.get(triangle.size() - 1));
    for (int i = triangle.size() - 2; i >= 0 ; i--)
        for (int j = 0; j <= i ; j++)
            res.set(j, Math.min(res.get(j), res.get(j + 1))
                    + triangle.get(i).get(j));
    return res.get(0);
}
```

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".
```

设状态为 f(i), 表示 s[0,i] 是否可以分词, 则状态转移方程为

转移方程: f (i) = any$_{of}$ (f(j) && s[j+1, i] $\in$ dict), 0 $\leqslant$ j < i

1. 深搜: 时间复杂度 O(2^n), 空间复杂度 O(n)

   OJ 超时。

```
public boolean helper(String s, int i, Set<String> dict) {
    int n = s.length();
    if (i >= n) return false;
    int j = i;
    while (j < n) {
        while (j < n && !dict.contains(s.substring(i, j + 1))) j++;
        if (j == n) return false;
        if (helper(s, j + 1, dict) == true) return true;
        j++;
    }
    return false;
}

public boolean wordBreak(String s, Set<String> dict) {
    if (dict.contains(s)) return true;
    if (s == null || s.length() == 0) return false;
    return helper(s, 0, dict);
}
```

2. 动态规划:

```
public boolean wordBreak(String s, Set<String> dict) {
    if (dict.contains(s)) return true;
    boolean [] dp = new boolean[s.length()];
    for (int i = 0; i < s.length(); i++)
        dp[i] = false;

    for (int i = 0; i < s.length(); i++)
        for (int j = i; j >= 0; j--) {
            String tmp = new String(s.substring(j, i+1));
            if (dict.contains(tmp))
                if ((j == 0) || (j >= 1 && dp[j-1])) {
                    dp[i] = true;
                    break;
                }
        }
    return dp[s.length()-1];
}
```

14.3 Hard

14.3.1 Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

设状态 f(i), 表示区间 [0, i](0 ⩽ i ⩽ n 1) 的最大利润, 状态 g(i), 表示区间 [i, n 1](0 ⩽ i ⩽ n 1) 的最大利润, 则最终答案为 max{ f(i) + g(i) }, 0 ⩽ i ⩽ n 1。
允许在一天内买进又卖出, 相当于不交易, 因为题目的规定是最多两次, 而不是一定要两次。
将原数组变成差分数组, 本题也可以看做是最大 m 子段和, m = 2.

```
public int maxProfit(int[] prices) {
    if (prices.length < 2) return 0;
    int [] past = new int[prices.length];
    int [] futu = new int[prices.length];
    int result = 0;
    for (int i = 1, valley = prices[0]; i < prices.length; i++) {
        valley = Math.min(valley, prices[i]);
        past[i] = Math.max(past[i - 1], prices[i]-valley);
    }
    for (int i = prices.length-2, peak = prices[prices.length-1]; i >= 0 ; i--) {
        peak = Math.max(peak, prices[i]);
        futu[i] = Math.max(futu[i+1], peak-prices[i]);
        result = Math.max(result, past[i] + futu[i]);
    }
    return result;
}
```

14.3.2 Coins in a Line

14.3.3 Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

Tags: Array Hash Table Stack Dynamic Programming

1. 栈：O(n) 这是一道非常综合的题目，要求在 0-1 矩阵中找出面积最大的全 1 矩阵。刚看到这道题会比较无从下手，brute force 就是对于每个矩阵都看一下，总共有 m(m+1)/2*n(n+1)/2 个子矩阵（原理跟字符串子串类似，字符串的子串数有 n(n+1)/2，只是这里是二维情形，所以是两个相乘），复杂度相当高，肯定不是面试官想要的答案，就不继续想下去了。

   这道题的解法灵感来自于 Largest Rectangle in Histogram 这道题，假设我们把矩阵沿着某一行切下来，然后把切的行作为底面，将自底面往上的矩阵看成一个直方图（histogram）。直方图的中每个项的高度就是从底面行开始往上 1 的数量。根据 Largest Rectangle in Histogram 我们就可以求出当前行作为矩阵下边缘的一个最大矩阵。接下来如果对每一行都做一次 Largest Rectangle in Histogram，从其中选出最大的矩阵，那么它就是整个矩阵中面积最大的子矩阵。算法的基本思路已经出来了，剩下的就是一些节省时间空间的问题了。

   我们如何计算某一行为底面时直方图的高度呢？如果重新计算，那么每次需要的计算数量就是当前行数乘以列数。然而在这里我们会发现一些动态规划的踪迹，如果我们知道上一行直方图的高度，我们只需要看新加进来的行（底面）上对应的列元素是不是 0，如果是，则高度是 0，否则则是上一行直方图的高度加 1。利用历史信息，我们就可以在线行时间内完成对高度的更新。我们知道，Largest Rectangle in Histogram 的算法复杂度是 O(n)。所以完成对一行为底边的矩阵求解复杂度是 O(n+n)=O(n)。接下来对每一行都做一次，那么算法总时间复杂度是 O(m*n)。

   空间上，我们只需要保存上一行直方图的高度 O(n)，加上 Largest Rectangle in Histogram 中所使用的空间 O(n)，所以总空间复杂度还是 O(n)。

```
private int largestRectangleArea(int[] height) {
    Stack<Integer> s = new Stack<Integer>();
    int [] hnew = new int[height.length + 1];
    hnew = Arrays.copyOf(height, height.length + 1);
    int res = 0;
    int tmp;
    int left;
    for (int i = 0; i <= height.length; i++) {
        if (s.isEmpty() || hnew[i] > hnew[s.peek()]) s.push(i);
        else {
            tmp = s.pop();
            res = Math.max(res, hnew[tmp] *
                           (s.isEmpty() ? i : i - s.peek() - 1));
            i--;
        }
    }
    return res;
}

public int maximalRectangle(char[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
    int [] h = new int[matrix[0].length];
    int res = 0;     // record max area
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++)
            h[j] = matrix[i][j] == '0' ? 0 : h[j] + 1;
        res = Math.max(res, largestRectangleArea(h));
    }
    return res;
}
```

2. 动态规划：

一般一个题目我首先会想想怎么暴力解决，比如这一题，可以枚举出所有的矩形，求出其中的面积最大者，那么怎么枚举呢，如果分别枚举矩形的宽度和高度，这样还得枚举矩形的位置，复杂度至少为 $O(n^4)$ (计算复杂度是我们把 matrix 的行、列长度都泛化为 n，下同)，我们可以枚举矩形左上角的位置，那么知道了矩形左上角的位置，怎么计算以某一点为左上角的矩形的最大面积呢？举例如下，下面的矩阵我们以 (0,0) 为矩形的左上角：

```
1 1 1 1 0 0
1 1 1 0 1 1
1 0 1 0 1 1
0 1 1 1 1 1
1 1 1 1 1 1
```

矩形高度是 1 时, 宽度为第一行中从第一个位置起连续的 1 的个数, 为 4, 面积为 4 * 1=4

矩形高度是 2 时, 第二行从第一个位置起连续 1 的个数是 3, 宽度为 min(3,4)=3, 面积为 3*2=6

矩形高度为 3 时, 第三行从第一个位置起连续 1 的个数是 1, 宽度为 min(1,3)=1, 面积为 1*3=3

矩形高度为 4 时, 第四行从第一个位置起连续 1 的个数是 0, 宽度为 min(0,1)=0, 面积为 0*4=0

后面的行就不用计算了, 因为上一行计算的宽度是 0, 下面所有宽度都是 0

因此以 (0,0) 为左上角的矩形的最大面积是 6，计算以某一点为左上角的矩形的最大面积复杂度是 O(n)。

注意到上面我们用到了信息“从某一行某个位置开始连续的 1 的个数”，这个我们可以通过动态规划求得：设 dp[i][j] 是从点 (i,j) 开始，这一行连续 1 的个数，动态规划方程如下：

初始条件：

- dp[i][n-1] = (matrix[i][n-1] == '1') （n 是 matrix 的列数）
- dp[i][j] = (matrix[i][j] == '1') ? 1 + dp[i][j + 1] : 0 (从方程看出我们应该从每一行的后往前计算)

计算 dp 复杂度 $O(n^2)$，枚举左上角位置以及计算以该位置为左上角的矩形最大面积复杂度是 $O(n^2*n)=O(n^3)$，总的复杂度是 $O(n^3)$

这个算法还可以优化，枚举到某个点时我们可以假设该点右下方全是 1，得到一个假设最大面积，如果这个面积比当前计算好的面积还要小，该点就可以直接跳过；在上面计算以某点为左上角的矩形面积时，也可以剪枝，剪枝方法同上。《JustDoIT》

```
public int maximalRectangle(char[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
    int m = matrix.length;
    int n = matrix[0].length;
    int [][] dp = new int[m][n];
    int res = 0;
    int width = 0;
    for (int i = 0; i < m; i++)
        dp[i][n - 1] = matrix[i][n - 1] == '1' ? 1 : 0;
    for (int i = 0; i < m; i++)
        for (int j = n - 2; j >= 0; j--)
            if (matrix[i][j] == '1')
                dp[i][j] = dp[i][j + 1] + 1;
    for (int i = 0; i < m; i++) // 以每个点作为矩形的左上角计算所得的最大矩形面积
        for (int j = 0; j < n; j++) {
            if ((n - j) * (m - i) <= res)
                break;          // 剪枝，column-j 是最大宽度，row-i 是最大高度
            width = dp[i][j];
            for (int k = i; k < m && width > 0; k++) {
                if (width * (m - i) <= res)
                    break;    // 剪枝,row-i 是以点 (i,j) 为左上角的矩形的最大高度
                if (width > dp[k][j])
                    width = dp[k][j]; //矩形宽度要取从第 i 行到第 k 行宽度的最小值
                res = Math.max(res, width * (k - i + 1));
            }
        }
    return res;
}
```

14.3.4 Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:
a) Insert a character
b) Delete a character
c) Replace a character
设状态为 f[i][j], 表示 A[0, i] 和 B[0, j] 之间的最小编辑距离。设 A[0, i] 的形式是 str1c, B[0, j] 的形式是 str2d,
一. 如果 c == d, 则 f[i][j] = f[i-1][j-1];
二. 如果 c != d,
(a) 如果将 c 替换成 d, 则 f[i][j] = f[i-1][j-1] + 1;
(b) 如果在 c 后面添加一个 d, 则 f[i][j] = f[i][j-1] + 1;
(c) 如果将 c 删除, 则 f[i][j] = f[i-1][j] + 1;

1. 动态规划：时间复杂度 O(n*m), 空间复杂度 O(n*m)。

```
public int minDistance(String s, String t) {
    int m = s.length();
    int n = t.length();
    int [][] res = new int[m + 1][n + 1];
    for (int i = 0; i <= m; i++)
        res[i][0] = i;
    for (int i = 0; i <= n; i++)
        res[0][i] = i;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (s.charAt(i - 1) == t.charAt(j - 1))
                res[i][j] = res[i - 1][j - 1];
            else
                res[i][j] = Math.min(Math.min(res[i - 1][j], res[i][j - 1]),
                                     Math.min(res[i - 1][j],
                                              res[i - 1][j - 1])) + 1;
    return res[m][n];
}
```

2. 动态规划 + 滚动数组：

14.3.5 Distinct Subsequence

Given a string S and a string T, count the number of distinct subsequences of T in S.
A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, ”ACE” is a subsequence of ”ABCDE” while ”AEC” is not).
Here is an example:
S = ”rabbbit”, T = ”rabbit”
Return 3.
设状态为 f(i, j), 表示 T[0, j] 在 S[0, i] 里出现的次数。首先, 无论 S[i] 和 T[j] 是否相等,
若不使用 S[i], 则 f(i, j) = f(i-1, j);
若 S[i] == T[j], 则可以使用 S[i], 则 f(i, j) = f(i-1, j) + f(i-1, j-1)。

```
public int numDistinct(String s, String t) {
    if (s == null || s.length() == 0) return 0;
```

```
    int m = s.length();
    int n = t.length();
    int [][] res = new int[m + 1][n + 1];
    for (int i = 0; i <= m; i++)
        res[i][0] = 1;
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s.charAt(i - 1) == t.charAt(j - 1))
                res[i][j] = res[i - 1][j] + res[i - 1][j - 1];
            else
                res[i][j] = res[i - 1][j];
        }
    }
    return res[m][n];
}
```

滚动数组：时间复杂度 O(m*n), 空间复杂度 O(n)

```
public int numDistinct(String s, String t) {
    if (s == null || s.length() == 0) return 0;
    int m = s.length();
    int n = t.length();
    int [] res = new int[n + 1];
    res[0] = 1;
    for (int i = 0; i < m; i++)
        for (int j = n - 1; j >= 0; j--)
            res[j + 1] += s.charAt(i) == t.charAt(j) ? res[j] : 0;
    return res[n];
}
```

14.3.6 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.
```

1. 思路 1：leetcode online

   忘记从线上哪个贴子看的了，根据回文可能有的两种特性，奇数长度回文和偶数长度回文，进行一维 dp 的判断。

```
    public int minCut(String s) {
        int n = s.length();
        int [] dp = new int[n + 1];
        for (int i = 0; i <= n ; i++) // initiate
            dp[i] = i - 1;
        for (int i = 0; i < n; i++) {
            // s[i - j] == s[i + j], aba
            for (int j = 0; i - j >= 0 && i + j <= n - 1 && s.charAt(i - j) == s.charAt
                dp[i + j + 1] = Math.min(dp[i - j] + 1, dp[i + j + 1]);
```

```
            // s[i - j] == s[i + 1 + j], abba
            for (int j = 0; i - j >= 0 && i + 1 + j <= n - 1 && s.charAt(i - j) == s.ch
                dp[i + j + 2] = Math.min(dp[i - j] + 1, dp[i + j + 2]);
        }
        return dp[n];
    }
```

2. 动态规划：时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

   定义状态 f(i,j) 表示区间 [i,j] 之间最小的 cut 数, 则状态转移方程为

   $f(i, j) = \min \{f(i, k) + f(k + 1, j)\}$, $i \leqslant k \leqslant j$, $0 \leqslant i \leqslant j < n$

   这是一个二维函数, 实际写代码比较麻烦。

   所以要转换成一维 DP。如果每次, 从 i 往右扫描, 每找到一个回文就算一次 DP 的话, 就可以转换为 f(i)= 区间 [i, n-1] 之间最小的 cut 数, n 为字符串长度, 则状态转移方程为

   $f(i) = \min \{f(j + 1) + 1\}$, $i \leqslant j < n$

   一个问题出现了, 就是如何判断 [i, j] 是否是回文? 每次都从 i 到 j 比较一遍? 太浪费了, 这里也是一个 DP 问题。

   定义状态 P[i][j] = true if [i, j] 为回文, 那么

   P[i][j] = str[i] == str[j] && P[i+1][j-1]

```
public int minCut(String s) {
    int n = s.length();
    int [] f = new int[n + 1];
    boolean [][] p = new boolean[n][n];
    for (int i = 0; i <= n; i++)
        f[i] = n - 1 - i;  // 最后一个 f[n] = -1
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i; j < n; j++) {
            if (s.charAt(i) == s.charAt(j) &&
                (j - i < 2 || p[i + 1][j - 1] == true)) {
                p[i][j] = true;
                f[i] = Math.min(f[i], f[j + 1] + 1);
            }
        }
    }
    return f[0];
}
```

14.3.7 Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.
For example, Given:

```
s1 = "aabcc",
s2 = "dbbca",
When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.
```

1. 递归: 会超时, 仅用来帮助理解

   今天写不出来代码，明天再写 ~

2. 动态规划：二维动规, 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

   设状态 f[i][j], 表示 s1[0, i] 和 s2[0, j], 匹配 s3[0, i+j]。如果 s1 的最后一个字符等于 s3 的最后一个字符, 则 f[i][j]=f[i-1][j]; 如果 s2 的最后一个字符等于 s3 的最后一个字符, 则 f[i][j]=f[i][j-1]。

   因此状态转移方程如下:

```
f[i][j] = (s1[i - 1] == s3 [i + j - 1] && f[i - 1][j])
       || (s2[j - 1] == s3 [i + j - 1] && f[i][j - 1]);
```

```
public boolean isInterleave(String s1, String s2, String s3) {
    int m = s1.length();
    int n = s2.length();
    if (s3.length() != m + n) return false;
    boolean [][] f = new boolean[m + 1][n + 1];
    f[0][0] = true;
    for (int j = 1; j <= m ; j++)
        f[j][0] = f[j - 1][0] && s1.charAt(j - 1) == s3.charAt(j - 1);
    for (int j = 1; j <= n ; j++)
        f[0][j] = f[0][j - 1] && s2.charAt(j - 1) == s3.charAt(j - 1);
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n ; j++) {
            f[i][j] = ((f[i - 1][j] && s1.charAt(i - 1) == s3.charAt(i + j - 1)) ||
                       (f[i][j - 1] && (s2.charAt(j - 1) == s3.charAt(i + j - 1))));
        }
    }
    return f[m][n];
}
```

3. 二维动规 + 滚动数组：时间复杂度 $O(n^2)$, 空间复杂度 $O(n)$

```
public boolean isInterleave(String s1, String s2, String s3) {
    int m = s1.length();
    int n = s2.length();
    if (s3.length() != m + n) return false;
    if (m < n) return isInterleave(s2, s1, s3);
    boolean [] f = new boolean[n + 1];
    for (int i = 0; i < n + 1; i++)
        f[i] = true;
    for (int i = 1; i < n + 1 ; i++)
        f[i] = s2.charAt(i - 1) == s3.charAt(i - 1) && f[i - 1];
    for (int i = 1; i < m + 1; i++) {
        f[0] = s1.charAt(i - 1) == s3.charAt(i - 1) && f[0];
        for (int j = 1; j < n + 1; j++)
            f[j] = (s1.charAt(i - 1) == s3.charAt(i + j - 1) && f[j])
                || (s2.charAt(j - 1) == s3.charAt(i + j - 1) && f[j - 1]);
    }
    return f[n];
}
```

14.3.8 Scramble String

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta  e
       / \
      t   a
```

We say that "rgtae" is a scrambled string of "great".

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

首先想到的是递归 (即深搜), 对两个 string 进行分割, 然后比较四对字符串。代码虽然简单, 但是复杂度比较高。有两种加速策略, 一种是剪枝, 提前返回; 一种是加缓存, 缓存中间结果, 即 memorization(翻译为记忆化搜索)。

这两种思路应该都非常容易理解，有大量工作要做，遇到已经干过的 quit 不干了，剪枝；或者保存了前面工作的结果，遇见干过的，直接调结果，缓存？

剪枝可以五花八门, 要充分观察, 充分利用信息, 找到能让节点提前返回的条件。例如, 判断两个字符串是否互为 scamble, 至少要求每个字符在两个字符串中出现的次数要相等, 如果不相等则返回 false。

加缓存, 可以用数组或 HashMap。本题维数较高, 用 HashMap, map 和 unordered$_{map}$ 均可。既然可以用记忆化搜索, 这题也一定可以用动规。设状态为 f[n][i][j], 表示长度为 n, 起点为 s1[i] 和起点为 s2[j] 两个字符串是否互为 scramble, 则状态转移方程为

```
f[n][i][j] = (f[k][i][j] && f[n-k][i+k][j+k])
           || (f[k][i][j+n-k] && f[n-k][i+k][j])
```

1. 递归：时间复杂度 O(n^6), 空间复杂度 O(1)

2. 动态规划：时间复杂度 O(n^3), 空间复杂度 O(n^3)

   代码的变量名称循环方式还是看起来很奇怪，改天再优化一下 ~

```
public boolean isScramble(String s1, String s2) {
    int n = s1.length();
    // f[k][i][j], 表示长度各为 k, 起点为 s1[i] 和
    // 起点为 s2[j] 两个字符串是否互为 scramble
    boolean [][][] f = new boolean[n + 1][n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            f[1][i][j] = s1.charAt(i) == s2.charAt(j);
    for (int k = 2; k <= n; k++) {
        for (int i = 0; i <= n - k; i++) {
            for (int j = 0; j <= n - k; j++) {
                f[k][i][j] = false;
                for (int x = 1; x < k && !f[k][i][j]; x++) {
                    f[k][i][j] = ( (f[x][i][j] && f[k - x][x + i][x + j])
                                   || (f[k - x] [x + i][j] && f[x][i][k - x + j]));
                }
            }
        }
    }
    return f[n][0][0];
}
```

3. 递归 + 剪枝：时间复杂度 O(n^6), 空间复杂度 O(1)

4. 备忘录法: 递归 +map 做 cache, 时间复杂度 O(n^3), 空间复杂度 O(n^3)

5. 备忘录法:

   定制一个哈希函数, 递归 + (c++) unordered$_{map}$ 做 cache, 比 map 快, 时间复杂度 O(n^3), 空间复杂度 O(n^3).

   Java 是一样的吗，改天试一下。。。这三个题 need every effort to work on them…

   http://www.cnblogs.com/TenosDoIt/p/3452004.html

   http://blog.unieagle.net/2012/10/23/leetcode%E9%A2%98%E7%9B%AE%EF%BC%9Ascramble-string%EF%BC%8C%E4%B8%89%E7%BB%B4%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92/

14.3.9 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

这是一道很有挑战的题（三个最难的之一，本楼层居中，还有其上其下 neighbor 楼。）。这是 LeetCode 里仅有的两个我觉得面试时应该用 c++ 写的题目，尤其是递归法，假如面试时实在想不出 DP 思路的话。所以这里代码直接贴 c++ 版的。。。

1. 递归：时间复杂度 O(n), 空间复杂度 O(1)

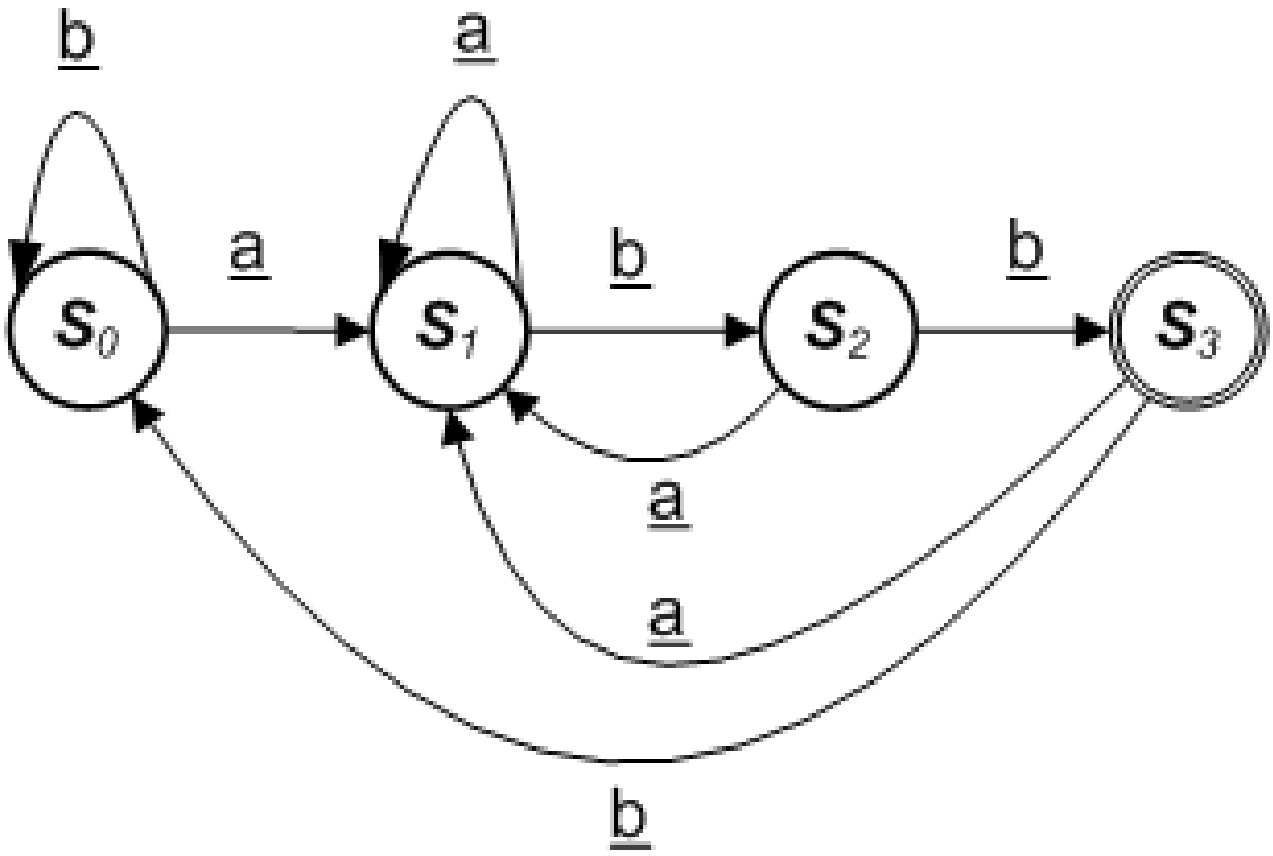


Figure 14.2: Regular Expression Matching

来自 http://leetcode.com/2011/09/regular-expression-matching.html

```
bool isMatch(const char *s, const char *p) {
    if (*p == '\0') return *s == '\0';
    if (*(p + 1) != '*') {
        if (*p == *s || (*p == '.' && *s != '\0'))
            return isMatch(s + 1, p + 1);
        else return false;
    } else { // next char is '*'
```

```
        while (*p == *s || (*p == '.' && *s != '\0')) {
            if (isMatch(s, p + 2)) // can optimize
                return true;
            s++;                   // but easier to understand
        }
    }
    return isMatch(s, p + 2);
}
```

搜到两种 Java 的解法，改天自己得再好好研究下。。。来自《压栈底的弱渣》

暂时只想出来了递归解法。其实就是自动机的实现原理。

对于两个字符串：string，pattern 有两种情况：

当 pattern[1] 不为“*”的时候（或 pattern 长度为 1），直接比较 patter[0] 和 string[0] 是否相等。若不等，则直接返回 false；若相等，则接着比较他们后面的字符，即跳到自动机的下一个 state。

当 pattern[1] 为“*”的时候，情况就稍微复杂一点了。

若 pattern[0] != string[0]，或 string 长度为 0 时，应跳过 pattern[0] 和 pattern[1] 继续比较后面的字符，因为“*”为 optional symbol。

若 pattern[0] == string[0]，我们既可以跳过 string[0]，也可以跳过 pattern[0] 和 pattern[1]。

```
public boolean isMatch(String s, String p) {
    if (p.length() == 0) return s.length() == 0;
    if (p.length() == 1 || p.charAt(1)!='*') {
        if (s.length() > 0 &&
            (p.charAt(0) == '.' || s.charAt(0) == p.charAt(0)))
            return isMatch(s.substring(1), p.substring(1));
        else return false;
    } else {
        if (s.length() == 0) return isMatch(s, p.substring(2));
        if (p.charAt(0) == '.' || s.charAt(0) == p.charAt(0))
            return isMatch(s.substring(1), p) || isMatch(s, p.substring(2));
        else return isMatch(s, p.substring(2));
    }
}
```

在 Java 1.7 以后，String.substring() 这个方法会重新拷贝一个新的字符串，这样在递归过程中内存效率很低, 而且会消耗大量时间用来对象初始化。因而可以考虑在递归的时候传 index 进去。

改进代码如下：

```
public boolean isMatchHelper(String s, String p, int is, int ip){
    if (ip == p.length()) return is  ==  s.length();
    if (ip == p.length()-1 || p.charAt(ip + 1)!='*'){ // no *
        if ( is < s.length() &&
             (p.charAt(ip) == '.' || s.charAt(is) == p.charAt(ip)) )
            return isMatchHelper(s, p, is + 1, ip + 1);
        else return false;
    } else {                                          // with *
```

```
            if (is == s.length())
                return isMatchHelper(s,p,is,ip + 2);       // empty
            if (p.charAt(ip) == '.' || s.charAt(is) == p.charAt(ip)) //match
                return isMatchHelper(s, p, is + 1, ip)
                    || isMatchHelper(s, p, is, ip + 2);  // two choices for continue
            else return isMatchHelper(s, p, is, ip + 2); // not match
        }
    }

    public boolean isMatch(String s, String p) {
        return isMatchHelper(s, p, 0, 0);
    }
```

2. 动态规划：

   仅次于 Scramble String 的最复杂的动规。。。

   思路改天再写，…

```
    public boolean isMatch(String s, String p) {
        if (p.length() == 0 && s.length() == 0) return true;
        if (p.length() == 0 && s.length() > 0) return false;
        int m = s.length();
        int n = p.length();
        boolean [][] dp = new boolean[m + 1][n + 1];
        dp[0][0] = true;
        for (int i = 1; i <= m ; i++)
            dp[i][0] = false;
        for (int i = 1; i <= n ; i++)
            if (i >= 2)
                dp[0][i] = (p.charAt(i - 1) == '*') ? dp[0][i - 2] : false;
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (p.charAt(j - 1) == '.')
                    dp[i][j] = dp[i - 1][j - 1];
                else if (p.charAt(j - 1) == '*')
                    dp[i][j] = ( dp[i][j - 1] || dp[i][j - 2] ||
                                 (dp[i - 1][j] &&
                                  ((s.charAt(i - 1) == p.charAt(j - 2)  || p.charAt(j -
                else if (s.charAt(i - 1) == p.charAt(j - 1))
                    dp[i][j] = dp[i - 1][j - 1];
            }
        }
        return dp[m][n];
    }
```

14.3.10 Wildcard 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
跟上一题很类似。
主要是'*' 的匹配问题。p 每遇到一个'*', 就保留住当前'*' 的坐标和 s 的坐标, 然后 s 从前往后扫描, 如果不成功, 则 s++, 重新扫描。
除 Scramble String 外最复杂的 DP, 因为'*' 可以什么也不用配。

1. 递归版: 时间复杂度 O(n!*m!), 空间复杂度 O(n)

   递归版, 会超时, 用于帮助理解题意.

   ```
   bool isMatch(const char *s, const char *p) {
       if (*p == '*') {
           while (*p == '*') ++p;
           if (*p == '\0') return true;
           while (*s != '\0' && !isMatch(s, p)) ++s;
           return *s != '\0';
       } else if (*p == '\0' || *s == '\0') return *p == *s;
       else if (*p == *s || *p == '?') return isMatch(++s, ++p);
       else return false;
   }
   ```

2. 迭代版：时间复杂度 O(n*m), 空间复杂度 O(1)

3. 动态规划：

   直接使用递归会超出内存或者超时，因为递归会重复计算很多相同的内容，可以使用动态规划利用中间结果来避免重复计算。

   用 dp[i][j] 表示 s 串的前 i 个字符和 p 串的前 j 个字符是否匹配，则计算

   p[i][j] 时，可以根据 p 的第 j 个字符分别讨论

   p[j-1] 为'*'，s 可以选择使用这个 *，转化为 dp[i-1][j]；或者不使用这个'*'，变为 dp[i][j-1]。

   p[j-1] 为'?'，此时 p[j-1] 必定要匹配 s[i-1]。

   p[j-1] 为普通字符，如果 p[j-1]==s[i-1]，匹配成功，问题转化为 dp[i-1][j-1]，否则匹配失败，dp[i][j] = false。

   ```
   if (p[j-1] == '*') then dp[i][j] = dp[i-1][j] || dp[i][j-1]
   if (p[j-1] == '?') then dp[i][j] = dp[i-1][j-1]
   if (p[j-1] == s[i-1])   dp[i][j] = dp[i-1][j-1] else dp[i][j]=false//普通字符
   ```

当 i 或 j 为 0 时作为初始化部分，根据定义，应有：

dp[0][0] = true。

dp[i][0] = false，因为有长度的 s 肯定无法和空串 p 匹配。

dp[0][i]，只有 p[i-1] 为 * 且 dp[0][i-1] 为真是才为真。

```
public boolean isMatch(String s, String p) {
    int minLen = 0;
    for (int i = 0; i < p.length(); i++)
        if (p.charAt(i) != '*') ++minLen;
    if (s.length() < minLen) return false;
    int m = s.length();
    int n = p.length();
    boolean [][] dp = new boolean[m + 1][n + 1];
    dp[0][0] = true;
    for (int i = 1; i <= m ; i++)
        dp[i][0] = false;
    for (int i = 1; i <= n ; i++)
        dp[0][i] = (p.charAt(i - 1) == '*' && dp[0][i - 1]) ? true : false;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n ; j++) {
            if (p.charAt(j - 1) == '*')  // user * or not
                dp[i][j] = (dp[i - 1][j] || dp[i][j - 1]);
            else if (s.charAt(i - 1) == p.charAt(j - 1) || p.charAt(j - 1) == '?')
                dp[i][j] = dp[i - 1][j - 1];
        }
    return dp[m][n];
}
```

14.4 others Covered

- Best Time to Buy and Sell Stock 13.1.2
- Best Time to Buy and Sell Stock II 13.1.3
- Longest Valid Parentheses 5.3.1
- Word Break II 10.2.1
- Maximum Rectangle `Maximum Rectangle`

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

Chapter 16

Backtracing and Recursion

16.1 Medium

16.1.1 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 kth permutation sequence.

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

16.1.2 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.

格雷码 (Gray Code) 的定义请参考 wikipedia http://en.wikipedia.org/wiki/Gray_code。

自然二进制码转换为格雷码:g 0 = b 0 , g i = b i ⊕ b i 1

保留自然二进制码的最高位作为格雷码的最高位, 格雷码次高位为二进制码的高位与次高位异或, 其余各位与次高位的求法类似。例如, 将自然二进制码 1001, 转换为格雷码的过程是: 保留最高位; 然后将第 1 位的 1 和第 2 位的 0 异或, 得到 1, 作为格雷码的第 2 位; 将第 2 位的 0 和第 3 位的 0 异或, 得到 0, 作为格雷码的第 3 位; 将第 3 位的 0 和第 4 位的 1 异或, 得到 1, 作为格雷码的第 4 位, 最终, 格雷码为 1101。

格雷码转换为自然二进制码:b 0 = g 0 , b i = g i ⊕ b i 1

保留格雷码的最高位作为自然二进制码的最高位, 次高位为自然二进制高位与格雷码次高位异或, 其余各位与次高位的求法类似。例如, 将格雷码 1000 转换为自然二进制码的过程是: 保留最高位 1, 作为自然二进制码的最高位; 然后将自然二进制码的第 1 位 1 和格雷码的第 2 位 0 异或, 得到 1, 作为自然二进制码的第 2 位; 将自然二进制码的第 2 位 1 和格雷码的第 3 位 0 异或, 得到 1, 作为自然二进制码的第 3 位; 将自然二进制码的第 3 位 1 和格雷码的第 4 位 0 异或, 得到 1, 作为自然二进制码的第 4 位, 最终, 自然二进制码为 1111。

格雷码有数学公式, 整数 n 的格雷码是 n ⊕ (n/2)。

这题要求生成 n 比特的所有格雷码。

最简单的方法, 利用数学公式, 对从 0 2 n 1 的所有整数, 转化为格雷码。

n 比特的格雷码, 可以递归地从 n 1 比特的格雷码生成。如图 §2-5 所示。

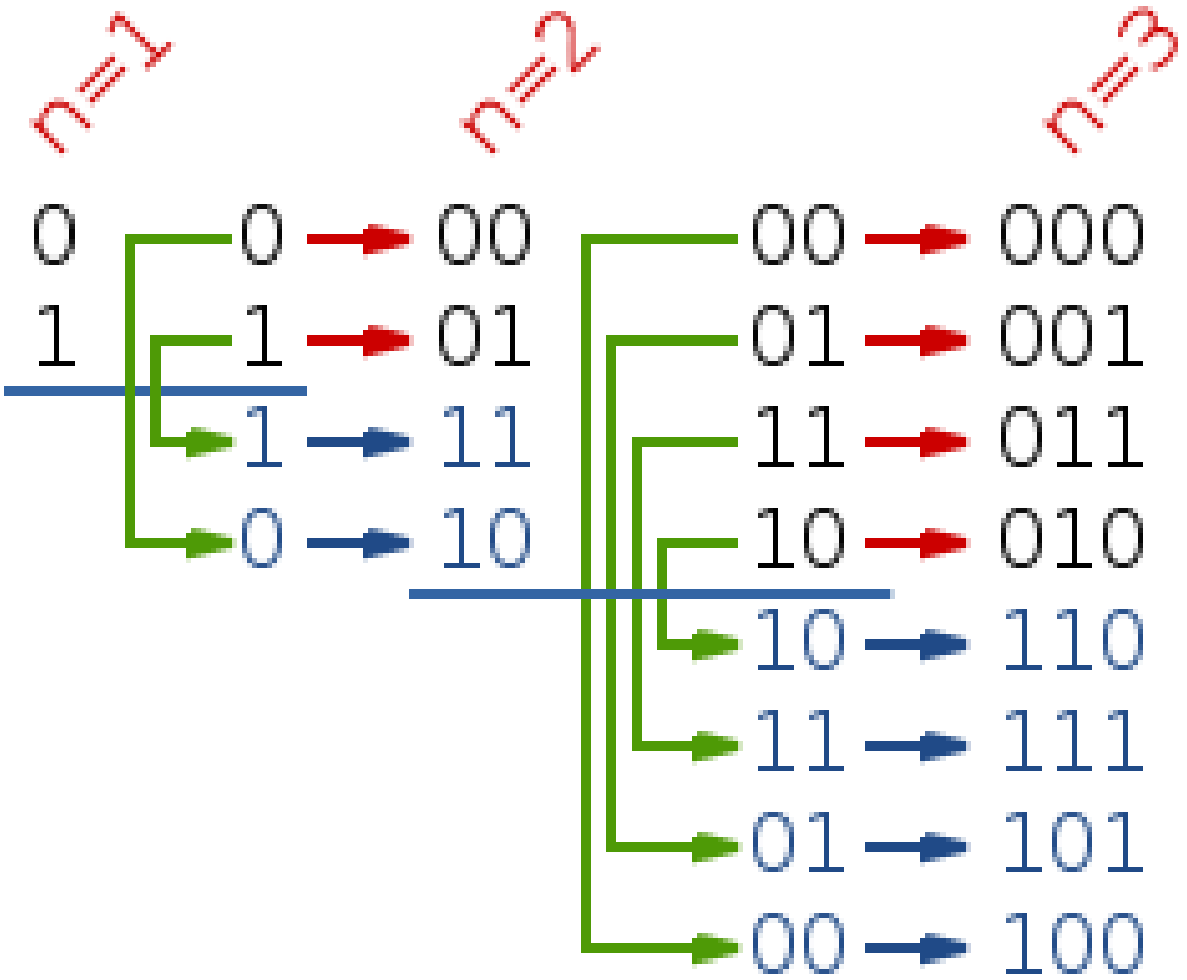


Figure 16.1: Gray Code

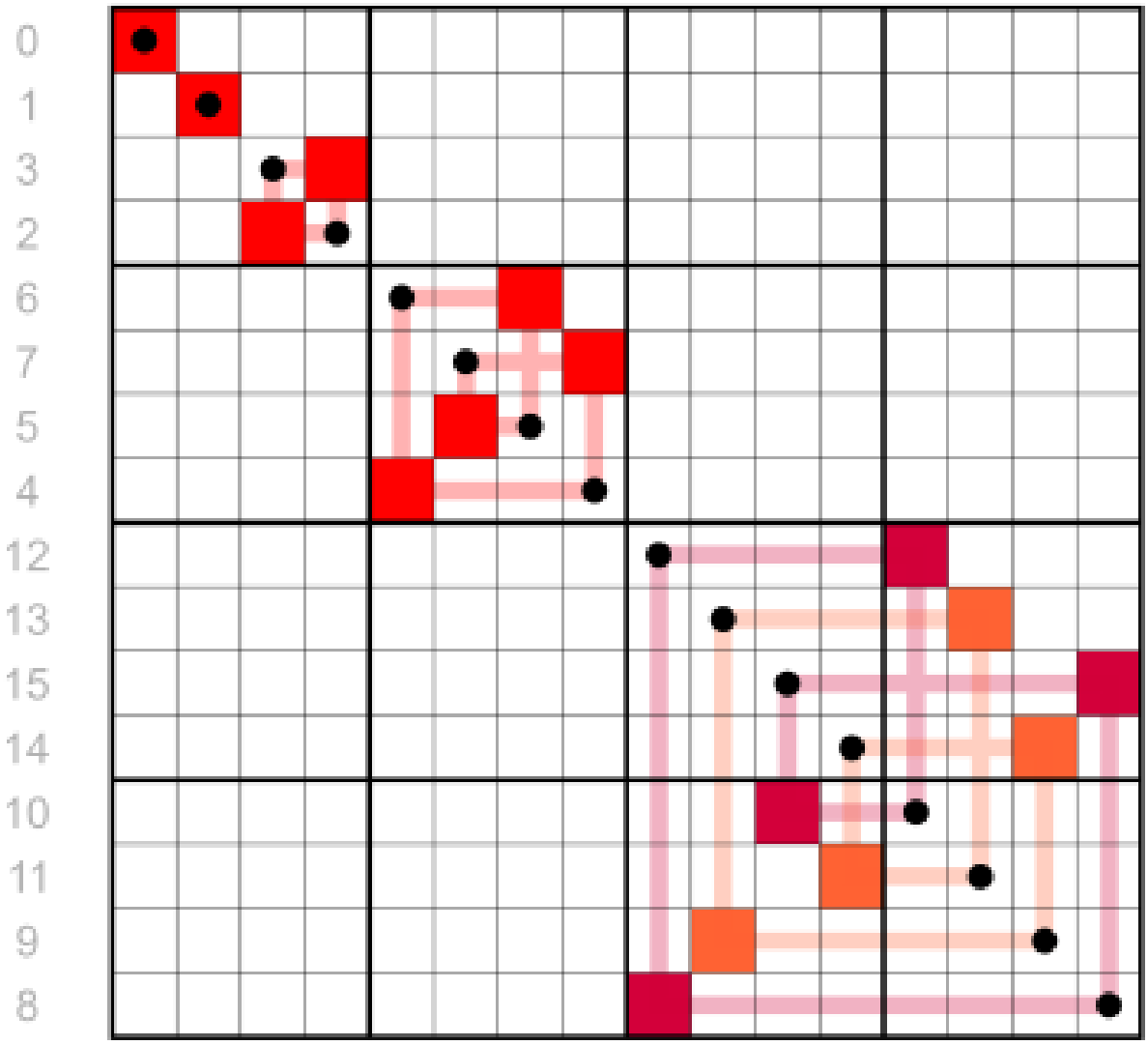


Figure 16.2: Gray Code 2

16.2 Hard

16.2.1 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

16.3 other Covered

- Subsets 9.1.1
- Subsets II 9.1.2
- Combinationas 9.1.4
- Letter Combinationas of Phone Number 9.1.5
- Permutation 9.1.3
- Permutation II 9.2.1
- Palindrome Partitioning 11.1.1
- Restore IP Addresses 11.1.2
- N-Queens 11.2.1
- N-Queens II 11.2.2
- Combination Sum 11.1.4
- Combination Sum II 11.1.5
- Generate Parentheses 11.1.3
- Sudoku Solver 6.3.3
- Word Search 11.2.4
- Word Ladder II 10.2.1
- Regular Expression Matching 14.3.9
- Wild Card Matching `Wild Card Matching`

Chapter 17

Bit Manipulation

17.1 Easy

17.1.1 Majority Element

Given an array of size n, find the majority element. The majority element is the element that appears more than n/2 times.

You may assume that the array is non-empty and the majority element always exist in the array.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

Tags: Divide and Conquer, Array, Bit Manipulation

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

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

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Tags: Hash Table, Bit Manipulation

17.2.3 Single Number II

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

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

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.

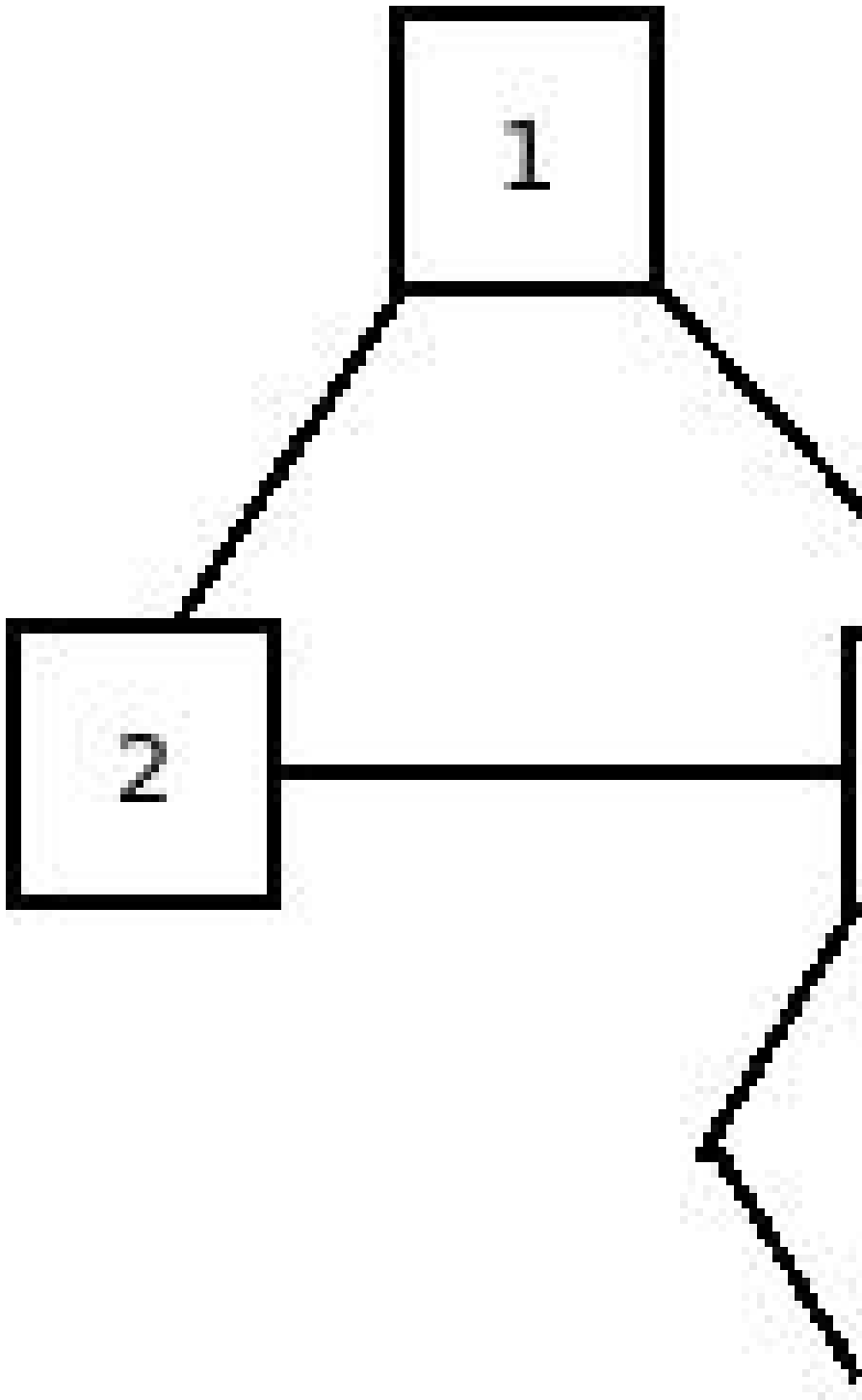
As an example, consider the serialized graph {0,1,2#1,2#2,2}.

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<Undirected
 * };
```

```
 */
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<UndirectedGraph
    Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGraphNode>();
    q.add(node);                        // added first node, need add its all Neighbo
    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 co
            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:

18.1.2 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).

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

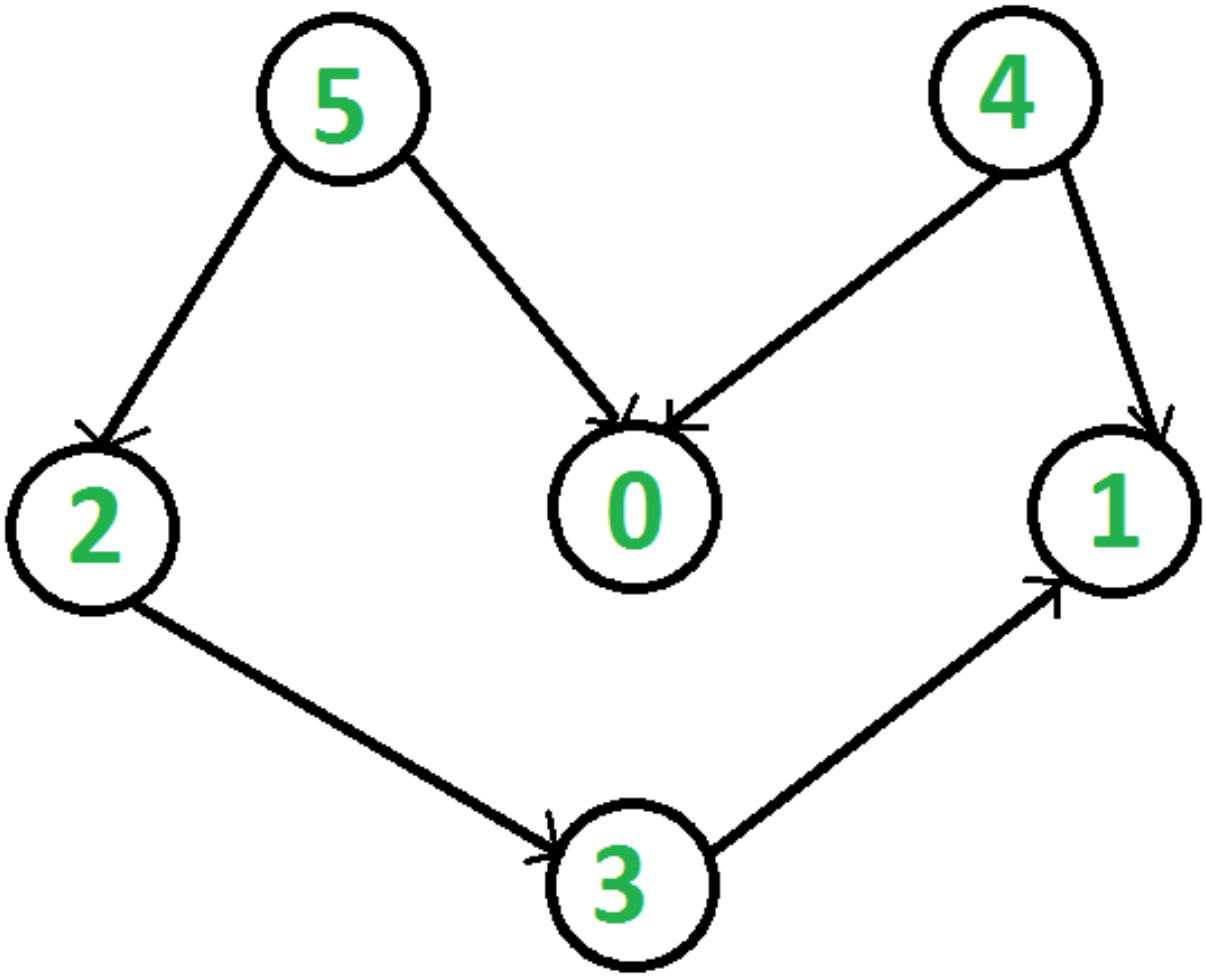


Figure 18.1: Topological Sorting

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.

18.2 other Covered

- Word Ladder 10.1.2
- Word Ladder II

10.2.1

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 frequence
            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 frequence
          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, connection
        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

20.1 Medium

20.1.1 Pascal's Triangle

20.1.2 Pascal's Triangle II

20.1.3 Two Sum

20.1.4 Two Sum II - Input array is sorted

20.1.5 Insert Interval

20.1.6 Merge Intervals

20.1.7 Spiral Matrix

20.1.8 Spiral Matrix II

20.1.9 Multiply Strings

20.1.10 Substring with Concatenation of All Words

20.1.11 ZigZag Conversion

20.1.12 Text Justification

20.2 other Covered

- Majority Element
- Container With Most Water
- Minimum Path Sum
- Missing Ranges
- Find Minimum in Rotated Sorted Array
- Triangle

- Find Peak Element
- Find Minimum in Rotated Sorted Array II
- Jump Game II
- Word Ladder II

Chapter 21

Math

21.1 Easy

21.1.1 Add Binary

21.1.2 String to Integer (atoi)

21.1.3 Palindrome Number

21.1.4 Factorial Trailing Zeroes

21.1.5 Excel Sheet Column Title

21.1.6 Excel Sheet Column Number

21.1.7 Reverse Integer

21.2 Medium

21.2.1 Multiply Strings

21.2.2 Divide Two Integers

21.2.3 Fraction to Recurring Decimal

21.2.4 Permutation Sequence

21.2.5 Next Permutation

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

21.4 other Covered

- Plus One 1.1.3
- Roman to Integers `Roman to Integers`
- Integer to Roman 2.2.2
- Pow(x, n) 12.1.1
- Sqrt(x) 12.1.2