

LeetCode Summary

deepwaterooo

January 28, 2015

Contents

1	数组 Array	5
1.1	Easy	6
1.1.1	Remove Element	6
1.1.2	Remove Duplicates from Sorted Array	6
1.1.3	Plus One	6
1.1.4	Pascal's Triangle	6
1.1.5	Pascal's Triangle II	6
1.1.6	Merge Sorted Array	6
1.1.7	Majority Element	6
1.2	Medium	6
1.2.1	Two Sum	6
1.2.2	3Sum Closest	6
1.2.3	Container With Most Water	6
1.2.4	Set Matrix Zeroes	6
1.2.5	Minimum Path Sum	6
1.2.6	Search Insert Position	6
1.2.7	Unique Paths	6
1.2.8	Search in Rotated Sorted Array II	6
1.2.9	Search for a Range	6
1.2.10	Search a 2D Matrix	6
1.2.11	Rotate Image	6
1.2.12	3Sum	6
1.2.13	4Sum	6
1.2.14	Remove Duplicates from Sorted Array II	6
1.2.15	Unique Paths II	6
1.2.16	Next Permutation	6
1.2.17	Missing Ranges	6
1.2.18	Find Minimum in Rotated Sorted Array	6
1.2.19	Combination Sum II	6
1.2.20	Two Sum II - Input array is sorted	6
1.2.21	Triangle	6
1.2.22	Maximum Subarray	6
1.2.23	Maximum Product Subarray	6
1.2.24	Word Search	6
1.2.25	Combination Sum	6
1.2.26	Subsets	6
1.2.27	Subsets II	6
1.2.28	Jump Game	6

1.2.29	Spiral Matrix	6
1.2.30	Spiral Matrix II	6
1.2.31	Sort Colors	6
1.2.32	Find Peak Element	6
1.3	Hard	6
1.3.1	Trapping Rain Water	6
1.3.2	Find Minimum in Rotated Sorted Array II	6
1.3.3	Insert Interval	6
1.3.4	Jump Game II	6
1.3.5	Longest Consecutive Sequence	6
1.3.6	First Missing Positive	6
1.3.7	Median of Two Sorted Arrays	6
1.3.8	Merge Intervals	6
1.3.9	Word Ladder II	6
1.3.10	Best Time to Buy and Sell Stock III	6
1.3.11	Search in Rotated Sorted Array	6
1.4	Others	6
2	字符串 String	9
2.1	Easy	10
2.1.1	Valid Parentheses	10
2.1.2	Add Binary	10
2.1.3	Compare Version Numbers	10
2.1.4	Count and Say	10
2.1.5	Longest Common Prefix	10
2.1.6	Roman to Integer	10
2.1.7	Valid Palindrome	10
2.1.8	Read N Characters Given Read4	10
2.1.9	Implement strStr()	10
2.1.10	ZigZag Conversion	10
2.1.11	String to Integer (atoi)	10
2.1.12	Length of Last Word	10
2.2	Medium	10
2.2.1	Restore IP Addresses	10
2.2.2	Anagrams	10
2.2.3	Integer to Roman	10
2.2.4	Multiply Strings	10
2.2.5	One Edit Distance	10
2.2.6	Longest Palindromic Substring	10
2.2.7	Letter Combinations of a Phone Number	10
2.2.8	Decode Ways	10
2.2.9	Longest Substring Without Repeating Characters	10
2.2.10	Generate Parentheses	10
2.2.11	Reverse Words in a String	10
2.2.12	Simplify Path	10
2.3	Hard	10
2.3.1	Edit Distance	10
2.3.2	Minimum Window Substring	10

2.3.3	Longest Substring with At Most Two Distinct Characters	10
2.3.4	Read N Characters Given Read4 II - Call multiple times	10
2.3.5	Regular Expression Matching	10
2.3.6	Interleaving String	10
2.3.7	Wildcard Matching	10
2.3.8	Scramble String	10
2.3.9	Distinct Subsequences	10
2.3.10	Substring with Concatenation of All Words	10
2.3.11	Text Justification	10
2.3.12	Valid Number	10
2.3.13	Word Ladder II	10
2.4	other Covered	10
3	Linked List	11
3.1	Easy	12
3.1.1	Remove Nth Node From End of List	12
3.1.2	Remove Duplicates from Sorted List	12
3.1.3	Merge Two Sorted Lists	12
3.1.4	Intersection of Two Linked Lists	12
3.2	Medium	12
3.2.1	Reverse Linked List II	12
3.2.2	Reorder List	12
3.2.3	Convert Sorted List to Binary Search Tree	12
3.2.4	Rotate List	12
3.2.5	Remove Duplicates from Sorted List II	12
3.2.6	Sort List	12
3.2.7	Insertion Sort List	12
3.2.8	Swap Nodes in Pairs	12
3.2.9	Linked List Cycle	12
3.2.10	Linked List Cycle II	12
3.2.11	Add Two Numbers	12
3.2.12	Partition List	12
3.3	Hard	12
3.3.1	Copy List with Random Pointer	12
3.3.2	Merge k Sorted Lists	12
3.3.3	Reverse Nodes in k-Group	12
4	树 Binary Tree, Binary Search Tree	13
4.1	二叉树的遍历	13
4.1.1	Binary Tree Preorder Traversal	13
4.1.2	Binary Tree Inorder Traversal	13
4.1.3	Binary Tree Postorder Traversal	14
4.1.4	Binary Tree Level Order Traversal	14
4.1.5	Binary Tree Level Order Traversal II	15
4.1.6	Binary Tree Zigzag Level Order Traversal	15
4.1.7	Recover Binary Search Tree	16
4.1.8	Same Tree	16
4.1.9	Symmetric Tree	16

4.1.10	Balanced Binary Tree	16
4.1.11	Flatten Binary Tree to Linked List	17
4.1.12	Populating Next Right Pointers in Each Node II	17
4.2	二叉树的构建	18
4.2.1	Construct Binary Tree from Preorder and Inorder Traversal	18
4.2.2	Construct Binary Tree from Inorder and Postorder Traversal	18
4.3	二叉树查找	18
4.3.1	Validate Binary Search Tree	18
4.3.2	Convert Sorted Array to Binary Search Tree	18
4.3.3	Convert Sorted List to Binary Search Tree	19
4.4	二叉树递归	19
4.4.1	Minimum Depth of Binary Tree	19
4.4.2	Maximum Depth of Binary Tree	19
4.4.3	Path Sum	19
4.4.4	Path Sum II	20
4.4.5	Binary Tree Maximum Path Sum	20
4.4.6	Populating Next Right Pointers in Each Node	21
4.4.7	Sum Root to Leaf Numbers	21
4.4.8	Binary Tree Upside Down	22
4.4.9	Binary Search Tree Iterator	22
4.5	others Covered	22
5	栈 Stack	23
5.1	Easy	23
5.1.1	Valid Parentheses	23
5.2	Medium	23
5.2.1	Evaluate Reverse Polish Notation	23
5.2.2	Simplify Path	23
5.3	Hard	24
5.3.1	Longest Valid Parentheses	24
5.3.2	Trapping Rain Water	26
5.3.3	Largest Rectangle in Histogram	27
5.3.4	Maximal Rectangle	29
5.4	Heap	31
5.4.1	Merge k sorted List	31
5.5	others Covered	32
6	Hash Table	35
6.1	Easy	36
6.1.1	Valid Sudoku	36
6.1.2	Two Sum III	36
6.2	Medium	36
6.2.1	Two Sum	36
6.2.2	4Sum	36
6.2.3	Binary Tree Inorder Traversal	36
6.2.4	Fraction to Recurring Decimal	36
6.2.5	Single Number	36
6.2.6	Anagrams	36

6.2.7	Longest Substring Without Repeating Characters	36
6.3	Hard	36
6.3.1	Minimum Window Substring	36
6.3.2	Copy List with Random Pointer	36
6.3.3	Sudoku Solver	36
6.3.4	Max Points on a Line	36
6.3.5	Substring with Concatenation of All Words	36
6.3.6	Longest Substring with At Most Two Distinct Characters	36
6.4	other Covered	36
7	广度优先搜索 Breadth First Search	37
7.1	Easy	37
7.1.1	Binary Tree Level Order Traversal	37
7.1.2	Binary Tree Level Order Traversal II	37
7.2	Medium	37
7.2.1	Word Ladder	37
7.2.2	Surrounded Regions	38
7.2.3	Clone Graph	38
7.2.4	Binary Tree Zigzag Level Order Traversal	38
7.2.5	Minimum Depth of Binary Tree	38
7.3	Hard	38
7.3.1	Word Ladder II	38
8	深度优先搜索 Depth First Search	39
8.1	Easy	40
8.1.1	Path Sum	40
8.1.2	Balanced Binary Tree	40
8.1.3	Same Tree	40
8.1.4	Symmetric Tree	40
8.1.5	Maximum Depth of Binary Tree	40
8.2	Medium	40
8.2.1	Sum Root to Leaf Numbers	40
8.2.2	Convert Sorted List to Binary Search Tree	40
8.2.3	Convert Sorted Array to Binary Search Tree	40
8.2.4	Construct Binary Tree from Preorder and Inorder Traversal	40
8.2.5	Construct Binary Tree from Inorder and Postorder Traversal	40
8.2.6	Clone Graph	40
8.2.7	Flatten Binary Tree to Linked List	40
8.2.8	Populating Next Right Pointers in Each Node	40
8.2.9	Validate Binary Search Tree	40
8.2.10	Path Sum II	40
8.3	Hard	40
8.3.1	Recover Binary Search Tree	40
8.3.2	Populating Next Right Pointers in Each Node II	40
8.3.3	Binary Tree Maximum Path Sum	40

9 排序	41
9.1 Medium	41
9.1.1 Sort Colors	41
9.1.2 Sort List	41
9.1.3 Largest Number	41
9.2 Hard	41
9.2.1 Insertion Sort List	41
9.2.2 Maximum Gap	41
9.2.3 Merge Intervals	41
9.2.4 Insert Interval	41
10 查找 Binary Search	43
10.1 Medium	43
10.1.1 Sqrt(x)	43
10.1.2 Divide Two Integers	43
10.1.3 Search in Rotated Sorted Array II	43
10.1.4 Two Sum II - Input array is sorted	43
10.1.5 Pow(x, n)	43
10.1.6 Search for a Range	43
10.1.7 Search a 2D Matrix	43
10.1.8 Find Minimum in Rotated Sorted Array	43
10.1.9 Search Insert Position	43
10.1.10 Find Peak Element	43
10.2 Hard	43
10.2.1 Median of Two Sorted Arrays	43
10.2.2 Dungeon Game	43
10.2.3 Find Minimum in Rotated Sorted Array II	43
10.2.4 Search in Rotated Sorted Array	43
11 暴力枚举法	45
11.1	45
11.1.1	45
12 分治法 Divide and Conquer	47
12.1 Easy	47
12.1.1 Majority Element	47
12.2 Medium	47
12.2.1 Maximum Subarray	47
12.3 Hard	47
12.3.1 Merge K sorted List	47
12.3.2 Median of Two Sorted Array	48
13 贪心法 Greedy Search	49
13.1 Medium	49
13.1.1 Gas Station	49
13.1.2 Best Time to Buy and Sell Stock	49
13.1.3 Best Time to Buy and Sell Stock II	49
13.1.4 Jump Game	50

13.2	Hard	50
13.2.1	Jump Game II	50
13.2.2	Candy	50
13.2.3	Wildcard Matching	50
14	动态归划 Dynamic Programming	53
14.1	Easy	53
14.1.1	Climbing Stairs	53
14.2	Medium	54
14.2.1	Unique Path	54
14.2.2	Unique Path with Obstacles	57
14.2.3	Minimum Path Sum	59
14.2.4	Unique Binary Search Tree	60
14.2.5	Unique Binary Search Tree II	61
14.2.6	Maximum Sum Subarray	62
14.2.7	Maximum Product Subarray	63
14.2.8	Decode Ways, bug~ !!	63
14.2.9	Triangle	65
14.2.10	Word Break	66
14.3	Hard	67
14.3.1	Best Time to Buy and Sell Stock III	67
14.3.2	Coins in a Line	68
14.3.3	Edit Distance	68
14.3.4	Distinct Subsequence	69
14.3.5	Palindrome Partitioning II	70
14.3.6	Interleaving String	70
14.3.7	Scramble String	70
14.3.8	Regular Expression Matching	71
14.3.9	Wildcard Matching	71
14.4	others Covered	72
15	Two Pointers and Sliding Window	73
15.1	Easy	74
15.1.1	Valid Palindrome	74
15.1.2	Remove Nth Node From End of List	74
15.1.3	Remove Element	74
15.1.4	Remove Duplicates from Sorted Array	74
15.1.5	Merge Sorted Array	74
15.1.6	Implement strStr()	74
15.2	Medium	74
15.2.1	3Sum	74
15.2.2	4Sum	74
15.2.3	Container With Most Water	74
15.2.4	Remove Duplicates from Sorted Array II	74
15.2.5	Partition List	74
15.2.6	Two Sum II - Input array is sorted	74
15.2.7	Linked List Cycle II	74
15.2.8	Longest Substring Without Repeating Characters	74

15.2.9	3Sum Closest	74
15.2.10	Linked List Cycle	74
15.2.11	Sort Colors	74
15.2.12	Rotate List	74
15.3	Hard	74
15.3.1	Trapping Rain Water	74
15.3.2	Longest Substring with At Most Two Distinct Characters	74
15.3.3	Substring with Concatenation of All Words	74
15.3.4	Minimum Window Substring	74
16	Backtracing and Recursion	75
16.1	排列	75
16.1.1	Permutation	75
16.1.2	Permutation II	75
16.1.3	Permutation Sequence	75
16.2	组合	76
16.2.1	Combinationas	76
16.2.2	Combination Sum	76
16.2.3	Combination Sum II	76
16.3	Subsets	77
16.3.1	Subsets: Bit Manipulation	77
16.3.2	Subsets II	77
16.4	others with Recursion	78
16.4.1	Letter Combinationas of Phone Number	78
16.4.2	Restore IP Addresses	78
16.4.3	Generate Parentheses	79
16.4.4	Gray Code	79
16.4.5	Word Search	79
16.4.6	Palindrome Partitioning	79
16.4.7	N-Queens	80
16.4.8	N-Queens II	80
16.4.9	Sudoku Solver	80
16.4.10	Regular Expression Matching	80
16.4.11	Wild Card Matching	81
16.4.12	Word Break II	81
16.4.13	Word Ladder II	82
17	Bit Manipulation	87
17.1	Easy	87
17.1.1	Majority Element	87
17.2	Medium	87
17.2.1	Subsets: Bit Manipulation	87
17.2.2	Single Number	88
17.2.3	Single Number II	88

18 图 Graphics	89
18.1 Medium	89
18.1.1 Clone Graph	89
18.2 Word Ladder, Word Ladder II: Backtracing	91
18.2.1 Word Ladder	91
18.2.2 Word Ladder II	92
18.2.3 Check whether the graph is bigraph	92
19 Data Structure	95
19.1 Easy	95
19.1.1 Two Sum III	95
19.1.2 Min Stack	95
19.2 Hard	96
19.2.1 LRU Cache	96
20 细节实现题	99
21 Math	101
21.1 Easy	102
21.1.1 Add Binary	102
21.1.2 Roman to Integers	102
21.1.3 String to Integer (atoi)	102
21.1.4 Palindrome Number	102
21.1.5 Plus One	102
21.1.6 Factorial Trailing Zeroes	102
21.1.7 Excel Sheet Column Title	102
21.1.8 Excel Sheet Column Number	102
21.1.9 Reverse Integer	102
21.2 Medium	102
21.2.1 Multiply Strings	102
21.2.2 Sqrt(x)	102
21.2.3 Divide Two Integers	102
21.2.4 Pow(x, n)	102
21.2.5 Fraction to Recurring Decimal	102
21.2.6 Permutation Sequence	102
21.2.7 Integer to Roman	102
21.2.8 Next Permutation: Math	102
21.3 Hard	103
21.3.1 Valid Number	103
21.3.2 Max Points on a Line	103

Chapter 1

数组 Array

1.1 Easy

1.1.1 Remove Element

1.1.2 Remove Duplicates from Sorted Array

1.1.3 Plus One

1.1.4 Pascal's Triangle

1.1.5 Pascal's Triangle II

1.1.6 Merge Sorted Array

1.1.7 Majority Element

1.2 Medium

1.2.1 Two Sum

1.2.2 3Sum Closest

1.2.3 Container With Most Water

1.2.4 Set Matrix Zeroes

1.2.5 Minimum Path Sum

1.2.6 Search Insert Position

1.2.7 Unique Paths

1.2.8 Search in Rotated Sorted Array II

1.2.9 Search for a Range

1.2.10 Search a 2D Matrix

1.2.11 Rotate Image

1.2.12 3Sum

- Construct Binary Tree from Preorder and Inorder Traversal
- 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~~

Chapter 2

字符串 **String**

2.1 Easy

2.1.1 Valid Parentheses

2.1.2 Add Binary

2.1.3 Compare Version Numbers

2.1.4 Count and Say

2.1.5 Longest Common Prefix

2.1.6 Roman to Integer

2.1.7 Valid Palindrome

2.1.8 Read N Characters Given Read4

2.1.9 Implement strStr()

2.1.10 ZigZag Conversion

2.1.11 String to Integer (atoi)

2.1.12 Length of Last Word

2.2 Medium

2.2.1 Restore IP Addresses

2.2.2 Anagrams

2.2.3 Integer to Roman

2.2.4 Multiply Strings

2.2.5 One Edit Distance

2.2.6 Longest Palindromic Substring

2.2.7 Letter Combinations of a Phone Number

Chapter 3

Linked List

3.1 Easy

3.1.1 Remove Nth Node From End of List

3.1.2 Remove Duplicates from Sorted List

3.1.3 Merge Two Sorted Lists

3.1.4 Intersection of Two Linked Lists

3.2 Medium

3.2.1 Reverse Linked List II

3.2.2 Reorder List

3.2.3 Convert Sorted List to Binary Search Tree

3.2.4 Rotate List

3.2.5 Remove Duplicates from Sorted List II

3.2.6 Sort List

3.2.7 Insertion Sort List

3.2.8 Swap Nodes in Pairs

3.2.9 Linked List Cycle

3.2.10 Linked List Cycle II

3.2.11 Add Two Numbers

3.2.12 Partition List

3.3 Hard

3.3.1 Copy List with Random Pointer

3.3.2 Merge k Sorted Lists

Chapter 4

树 Binary Tree, Binary Search Tree

4.1 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)。

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

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

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

4.1.4 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]
]

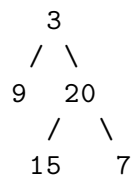
```

1. 递归版
2. 迭代版

4.1.5 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},



return its bottom-up level order traversal as:

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

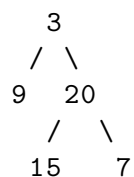
1. 递归版
2. 迭代版

4.1.6 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},



return its zigzag level order traversal as:

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

1. 递归版
2. 迭代版

4.1.7 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。

4.1.8 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 $O(\log n)$
2. 迭代版: 时间复杂度 $O(n)$, 空间复杂度 $O(\log n)$

4.1.9 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 $O(\log n)$
2. 迭代版: 时间复杂度 $O(n)$, 空间复杂度 $O(\log n)$

4.1.10 Balanced Binary Tree

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

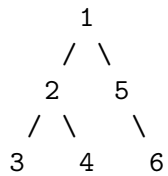
For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

时间复杂度 $O(n)$, 空间复杂度 $O(\log n)$

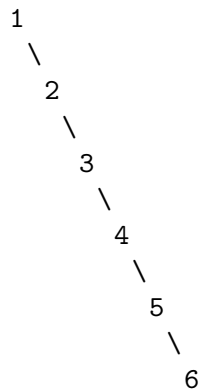
4.1.11 Flatten Binary Tree to Linked List

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

For example, Given



The flattened tree should look like:



[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 $O(\log n)$
2. 递归版 2: 时间复杂度 $O(n)$, 空间复杂度 $O(\log n)$
3. 迭代版: 时间复杂度 $O(n)$, 空间复杂度 $O(\log n)$

4.1.12 Populating Next Right Pointers in Each Node II

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

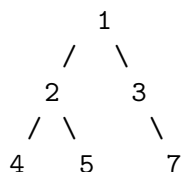
What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space.

For example,

Given the following binary tree,



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 $O(1)$

4.2 二叉树的构建

4.2.1 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.

4.2.2 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.

4.3 二叉树查找

4.3.1 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 $O(\log(n))$

4.3.2 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。

4.3.3 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

4.4 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。

4.4.1 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 $O(\log(n))$

4.4.2 Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

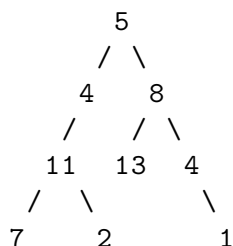
The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

4.4.3 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。

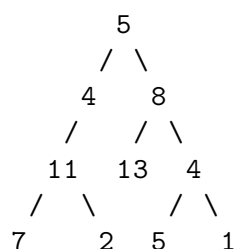
时间复杂度 $O(n)$, 空间复杂度 $O(\log(n))$

4.4.4 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,



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。

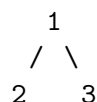
时间复杂度 $O(n)$, 空间复杂度 $O(\log(n))$

4.4.5 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,



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 $O(\log(n))$

4.4.6 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 $O(\log(n))$

4.4.7 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.

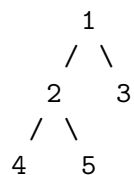
时间复杂度 $O(n)$, 空间复杂度 $O(\log(n))$

4.4.8 Binary Tree Upside Down

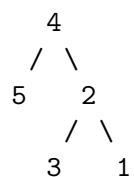
Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example:

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



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



4.4.9 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 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)、如果栈顶元素是'(': 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 longestValidParentheses(String s) {
    Stack<Integer> k = new Stack<Integer>();
    int res = 0;
    k.push(-1);    // 为了处理边界条件, 在栈底添加 -1
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '(')
            k.push(i);
        else { // ')'
            if (k.isEmpty())
                k.push(i);
            else {
                int tmp = k.peek();
                if (tmp >= 0 && s.charAt(tmp) == '(') { // -1
                    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i
    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
        }
    }
    return res;
}
```

```

        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.

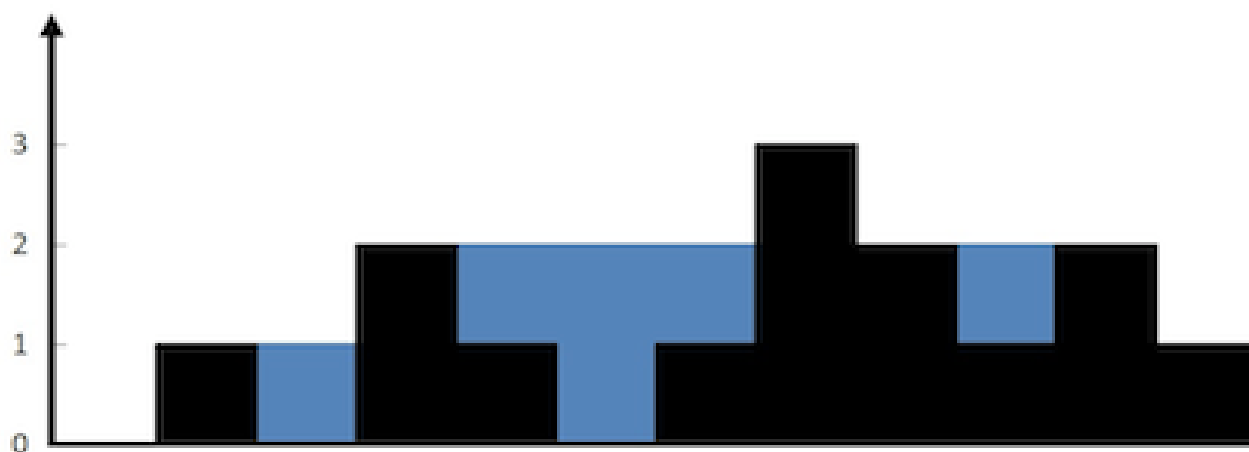


Figure 5.1: Trapping Rain Water

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 $\min(\max_{left}, \max_{right}) - \text{height}$ 。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 $O(n^2)$ 这样大数据会超时。

枚举的过程中可以优化一下: 如果 $\text{height}[i+1] > \text{height}[i]$, 那么区间 $[k \cdots i]$ 内的最大矩形肯定不会超过区间 $[k \cdots i+1]$ 内的最大矩形, 比如上例中的区间 $[0 \cdots 3]$ 内的矩形要大于 $[0 \cdots 2]$ 内的矩形, 因为 $\text{height}^1 > \text{height}^2$ 。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
```

¹DEFINITION NOT FOUND.

²DEFINITION NOT FOUND.

```

        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；

- 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`。

以上取 `i` 从 0 到 `n-1`。

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]; // 数组末尾插入元素dummy0
    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>

5.3.4 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:

- $dp[i][n-1] = (\text{matrix}[i][n-1] == '1') \quad (n \text{ 是 matrix 的列数})$
- $dp[i][j] = (\text{matrix}[i][j] == '1') ? 1 + dp[i][j + 1] : 0$ (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;
}
```

5.4 Heap

5.4.1 Merge k sorted List

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

Tags: Divide and Conquer, Linked List, Heap

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

```

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

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

```

5.5 others Covered

- Binary Tree Preorder Traversal [4.1.1](#)
- Binary Tree Inorder Traversal [4.1.2](#)
- Binary Tree Postorder Traversal [4.1.3](#)
- Binary Search Tree Iterator [4.4.9](#)
- Binary Tree Zigzag Level Order Traversal [4.1.6](#)
- Min Stack [19.1.2](#)

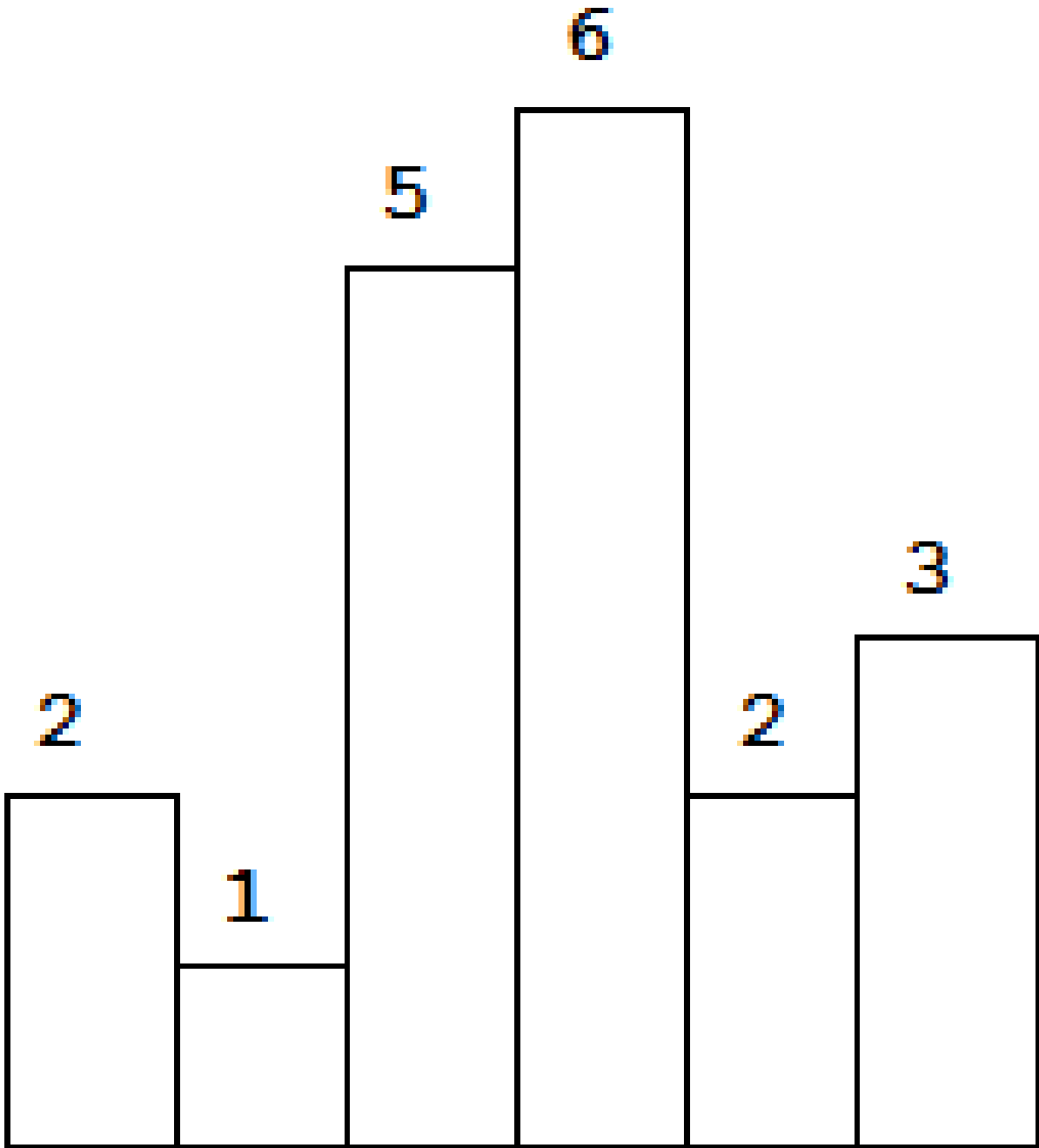


Figure 5.2: Largest Rectangle in Histogram 1

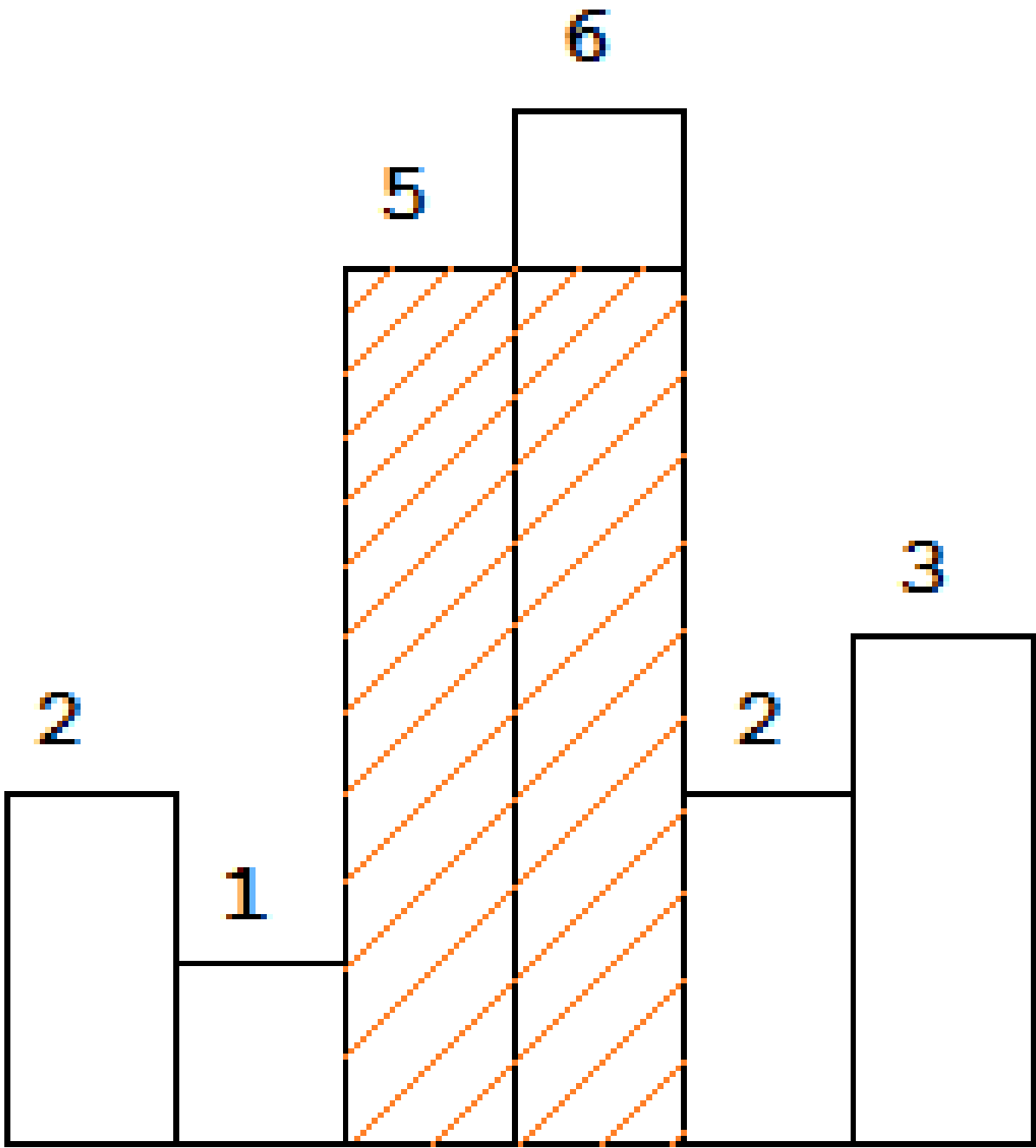


Figure 5.3: Largest Rectangle in Histogram 2

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 [5.3.4](#)

Chapter 7

广度优先搜索 Breadth First Search

7.1 Easy

7.1.1 Binary Tree Level Order Traversal

7.1.2 Binary Tree Level Order Traversal II

7.2 Medium

7.2.1 Word Ladder

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

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

For example,

Given:

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

Note:

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

7.2.2 Surrounded Regions

7.2.3 Clone Graph

7.2.4 Binary Tree Zigzag Level Order Traversal

7.2.5 Minimum Depth of Binary Tree

7.3 Hard

7.3.1 Word Ladder II

Chapter 8

深度优先搜索 Depth First Search

8.1 Easy

8.1.1 Path Sum

8.1.2 Balanced Binary Tree

8.1.3 Same Tree

8.1.4 Symmetric Tree

8.1.5 Maximum Depth of Binary Tree

8.2 Medium

8.2.1 Sum Root to Leaf Numbers

8.2.2 Convert Sorted List to Binary Search Tree

8.2.3 Convert Sorted Array to Binary Search Tree

8.2.4 Construct Binary Tree from Preorder and Inorder Traversal

8.2.5 Construct Binary Tree from Inorder and Postorder Traversal

8.2.6 Clone Graph

8.2.7 Flatten Binary Tree to Linked List

8.2.8 Populating Next Right Pointers in Each Node

8.2.9 Validate Binary Search Tree

8.2.10 Path Sum II

8.3 Hard

8.3.1 Recover Binary Search Tree

8.3.2 Populating Next Right Pointers in Each Node II

8.3.3 Binary Tree Maximum Path Sum

Chapter 9

排序

9.1 Medium

9.1.1 Sort Colors

9.1.2 Sort List

9.1.3 Largest Number

9.2 Hard

9.2.1 Insertion Sort List

9.2.2 Maximum Gap

9.2.3 Merge Intervals

9.2.4 Insert Interval

Chapter 10

查找 Binary Search

10.1 Medium

10.1.1 Sqrt(x)

10.1.2 Divide Two Integers

10.1.3 Search in Rotated Sorted Array II

10.1.4 Two Sum II - Input array is sorted

10.1.5 Pow(x, n)

10.1.6 Search for a Range

10.1.7 Search a 2D Matrix

10.1.8 Find Minimum in Rotated Sorted Array

10.1.9 Search Insert Position

10.1.10 Find Peak Element

10.2 Hard

10.2.1 Median of Two Sorted Arrays

10.2.2 Dungeon Game

10.2.3 Find Minimum in Rotated Sorted Array II

10.2.4 Search in Rotated Sorted Array

Chapter 11

暴力枚举法

11.1

11.1.1

Chapter 12

分治法 Divide and Conquer

12.1 Easy

12.1.1 Majority Element

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

12.2 Medium

12.2.1 Maximum 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

12.3 Hard

12.3.1 Merge K sorted List

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

Tags: Divide and Conquer Linked, List, Heap

12.3.2 Median of Two Sorted Array

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be $O(\log (m+n))$.

Tags: Divide and Conquer, Array, Binary Search

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 i th 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 i th 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, $m = \text{数组长度}$ 。

```
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.

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.)

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?

13.2.3 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

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 $m \times n$ 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 3×7 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!

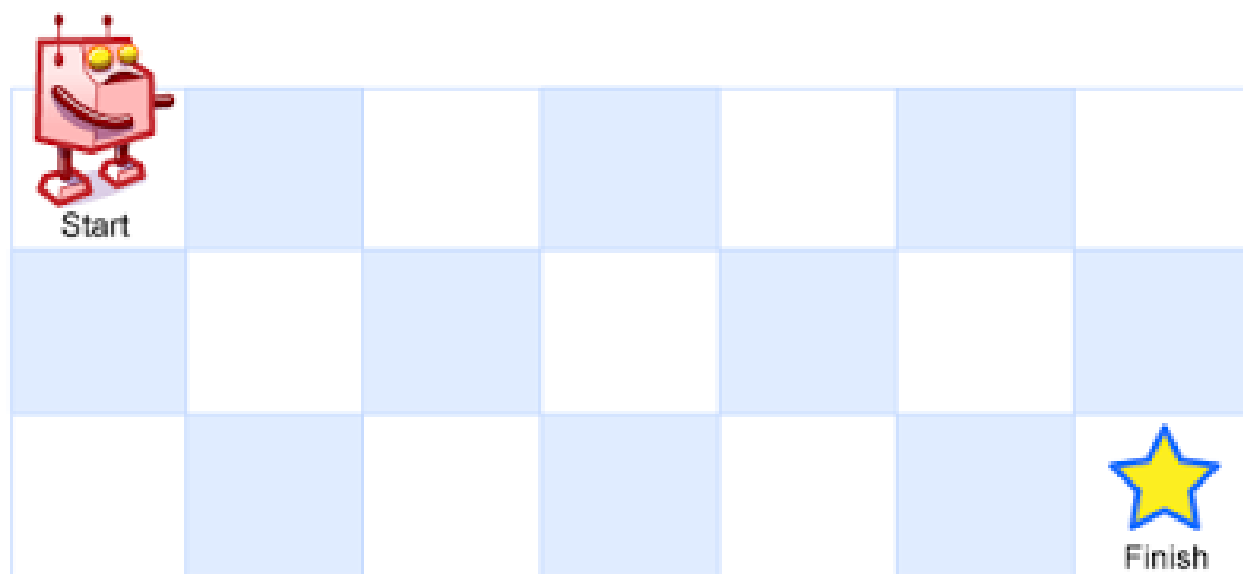


Figure 14.1: Unique Path

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， $O(n^4)$ ~

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 $dp[i][j] = \text{uniquePaths}(i, j)$ ，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 $m \times n$ 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: $f[i][j] = \min(f[i-1][j], f[i][j-1]) + \text{grid}[i][j]$

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];
}

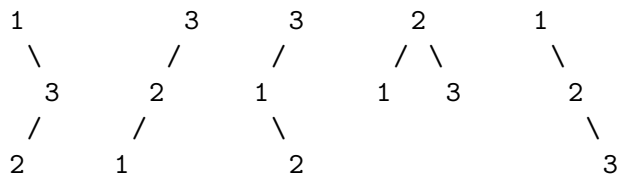
```

3. 动态规划 + 滚动数组

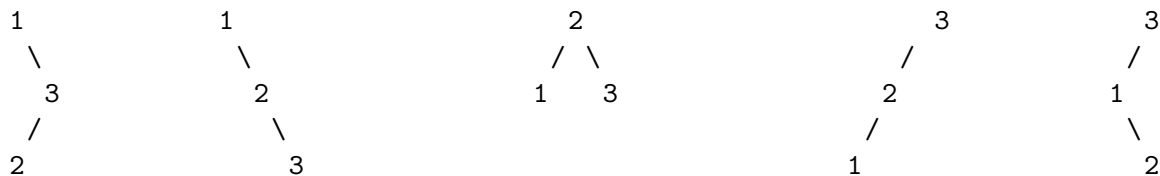
14.2.4 Unique Binary Search Tree

Given n , how many structurally unique BST's (binary search trees) that store values $1 \cdots n$?

For example, Given $n = 3$, there are a total of 5 unique BST's.



如果把上例的顺序改一下, 就可以看出规律了。《水中的鱼》



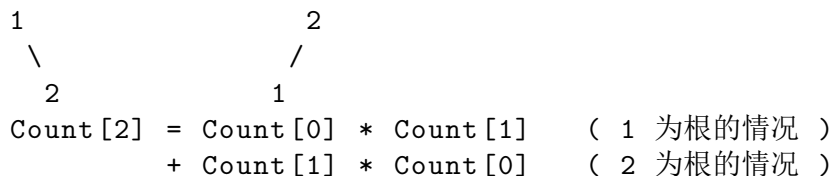
比如, 以 1 为根的树有几个, 完全取决于有二个元素的子树有几种。同理, 2 为根的子树取决于一个元素的子树有几个。以 3 为根的情况, 则与 1 相同。

定义 $\text{Count}[i]$ 为以 $[0, i]$ 能产生的 Unique Binary Tree 的数目,

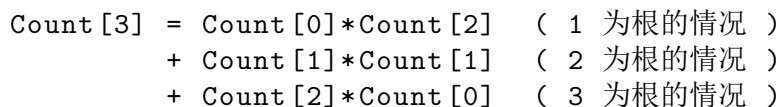
如果数组为空, 毫无疑问, 只有一种 BST, 即空树, $\text{Count}[0] = 1$

如果数组仅有一个元素 $\{1\}$, 只有一种 BST, 单个节点, $\text{Count}[1] = 1$

如果数组有两个元素 $\{1, 2\}$, 那么有如下两种可能



再看一遍三个元素的数组, 可以发现 BST 的取值方式如下:



所以, 由此观察, 当数组为 $1, 2, 3, \dots, n$ 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

$$\text{Count}(i) = \sum_{k=1}^i \text{Count}(k-1) * \text{Count}(i-k)$$

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];
}

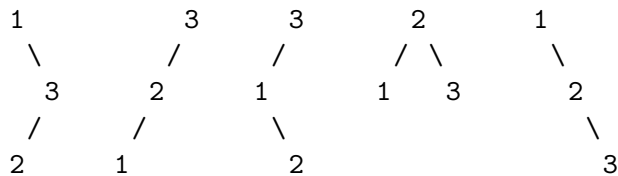
```

14.2.5 Unique Binary Search Tree II

Given n , generate all structurally unique BST's (binary search trees) that store values $1 \cdots n$.

For example,

Given $n = 3$, your program should return all 5 unique BST's shown below.



```

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  $1 \leq j \leq n$ 
target = max (f[j]) , 其中  $1 \leq j \leq n$ 

```

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 $O(n \cdot (\log(n)))$
- 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 digit // 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

$$\text{sum}[i][j] = \min(\text{sum}[i-1][j-1], \text{sum}[i-1][j]) + \text{triangle}[i][j].$$

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 $1+2+\dots+n = n*(n+1)/2$ 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: $f(i) = \text{any}_{of} (f(j) \ \&\& \ s[j+1, i] \in \text{dict}), 0 \leq j < i$

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 i th 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 $[0, i](0 \leq i \leq n-1)$ 的最大利润, 状态 $g(i)$, 表示区间 $[i, n-1](0 \leq i \leq n-1)$ 的最大利润, 则最终答案为 $\max\{f(i) + g(i)\}, 0 \leq i \leq n-1$ 。

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 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:

14.3.4 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];
}

```

14.3.5 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 $s = \text{"aab"}$,

Return 1 since the palindrome partitioning $[\text{"aa"}, \text{"b"}]$ could be produced using 1 cut.

14.3.6 Interleaving String

Given s_1 , s_2 , s_3 , find whether s_3 is formed by the interleaving of s_1 and s_2 .

For example, Given:

$s_1 = \text{"aabcc"}$,

$s_2 = \text{"dbbca"}$,

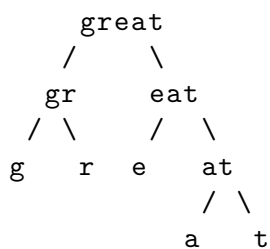
When $s_3 = \text{"aadbcbcbac"}$, return true.

When $s_3 = \text{"aadbbaacc"}$, return false.

14.3.7 Scramble String

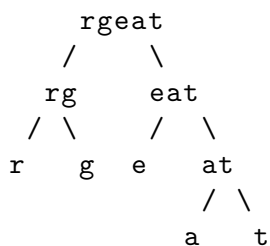
Given a string s_1 , we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of $s_1 = \text{"great"}$:



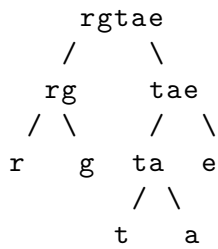
To scramble the string, we may choose any non-leaf node and swap its two children.

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



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".



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.

14.3.8 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

14.3.9 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

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 [1.3.9](#)
- 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 排列

16.1.1 Permutation

Given a collection of numbers, return all possible permutations.

For example,

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

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

16.1.2 Permutation II

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

For example,

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

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

16.1.3 Permutation Sequence

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

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

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

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

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

16.2 组合

16.2.1 Combinationas

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

For example,

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

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

16.2.2 Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers 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 (a_1, a_2, \cdots, a_k) must be in non-descending order. (ie, $a_1 \leq a_2 \leq \cdots \leq a_k$).
- The solution set must not contain duplicate combinations.

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

A solution set is:

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

16.2.3 Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers 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 (a_1, a_2, \cdots, a_k) must be in non-descending order. (ie, $a_1 \leq a_2 \leq \cdots \leq a_k$).
- The solution set must not contain duplicate combinations.

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

A solution set is:

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

16.3 Subsets

16.3.1 Subsets: Bit Manipulation

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

Note:

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

For example,

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

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

Tags: Array Backtracking, Bit Manipulation

16.3.2 Subsets II

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

Note:

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

For example,

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

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

16.4 others with Recursion

16.4.1 Letter Combinationas of Phone Number

Given a digit string, return all possible letter combinations that the number could represent. A mapping of digit to letters (just like on the telephone buttons) is given below.



Figure 16.1: Letter Combinationas of Phone Number

Input:Digit string "23"

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

Note:

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

16.4.2 Restore IP Addresses

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

For example:

Given "25525511135",

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

16.4.3 Generate Parentheses

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

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

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

16.4.4 Gray Code

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

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

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

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

Note:

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

16.4.5 Word Search

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

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

For example,

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

16.4.6 Palindrome Partitioning

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

Return all possible palindrome partitioning of s .

For example,

```

given s = "aab",
Return
[
    ["aa", "b"],
    ["a", "a", "b"]
]

```

16.4.7 N-Queens

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

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

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

For example,

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

```

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

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

```

16.4.8 N-Queens II

Follow up for N-Queens problem.

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

16.4.9 Sudoku Solver

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

Empty cells are indicated by the character '.'.

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

A sudoku puzzle...

...and its solution numbers marked in red.

Tags: Backtracking, Hash Table

16.4.10 Regular Expression Matching

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

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

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

The function prototype should be:

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

Some examples:

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

Tags: Dynamic Programming, Backtracking, String

16.4.11 Wild Card Matching

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

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

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

The function prototype should be:

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

Some examples:

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

Tags: Dynamic Programming, Backtracking, Greedy, String

16.4.12 Word Break II

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

Return all such possible sentences.

For example, given

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

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

Tags: Dynamic Programming Backtracking

16.4.13 Word Ladder II

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

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

For example,

Given:

```
start = "hit"
```

```
end = "cog"
```

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

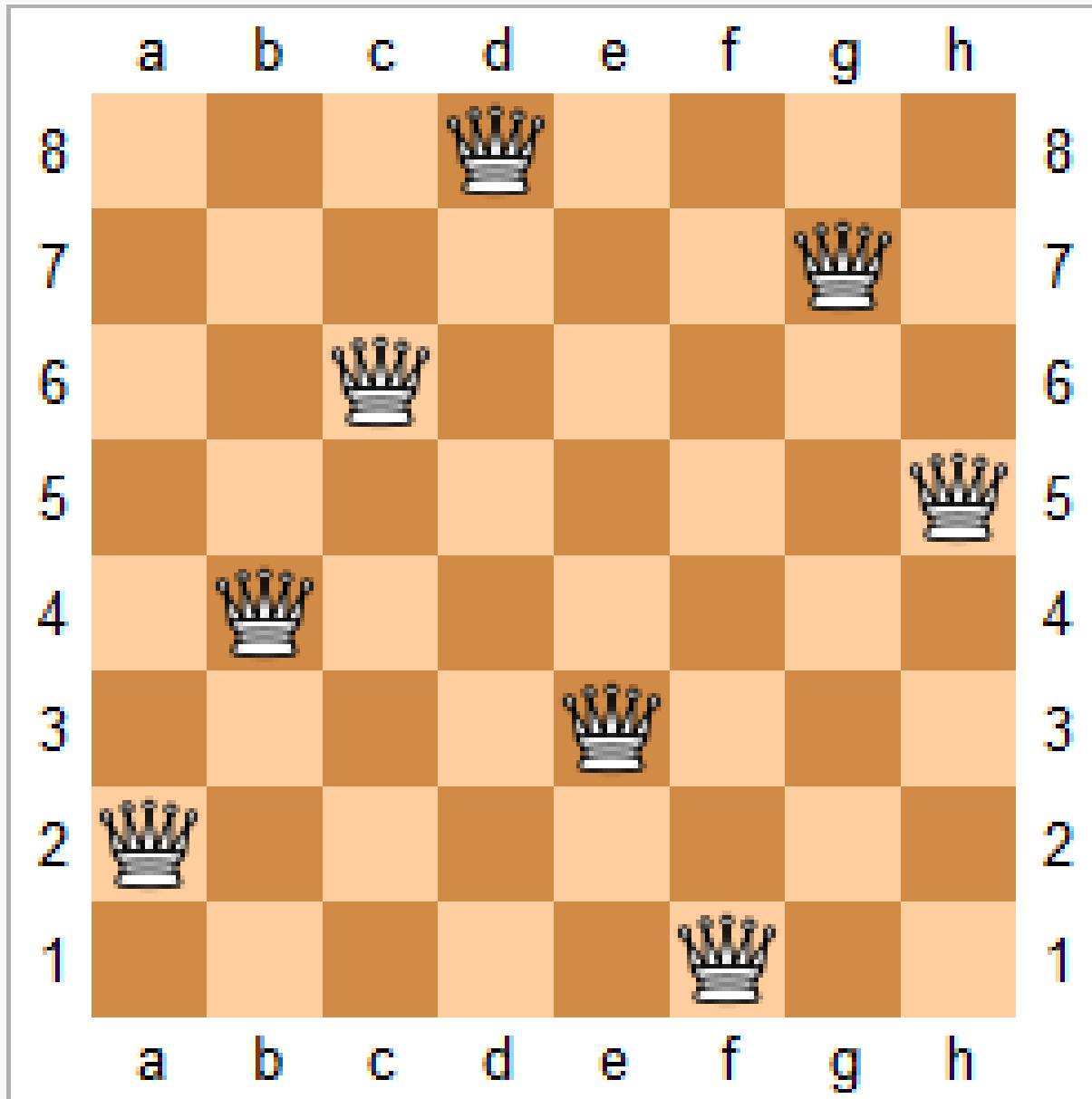
Return

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

Note:

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

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



One solution to the eight queens puzzle

Figure 16.2: N-Queens

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

Figure 16.3: Sudoku Solver 1

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

Figure 16.4: Sudoku Solver 2

Chapter 17

Bit Manipulation

17.1 Easy

17.1.1 Majority Element

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:

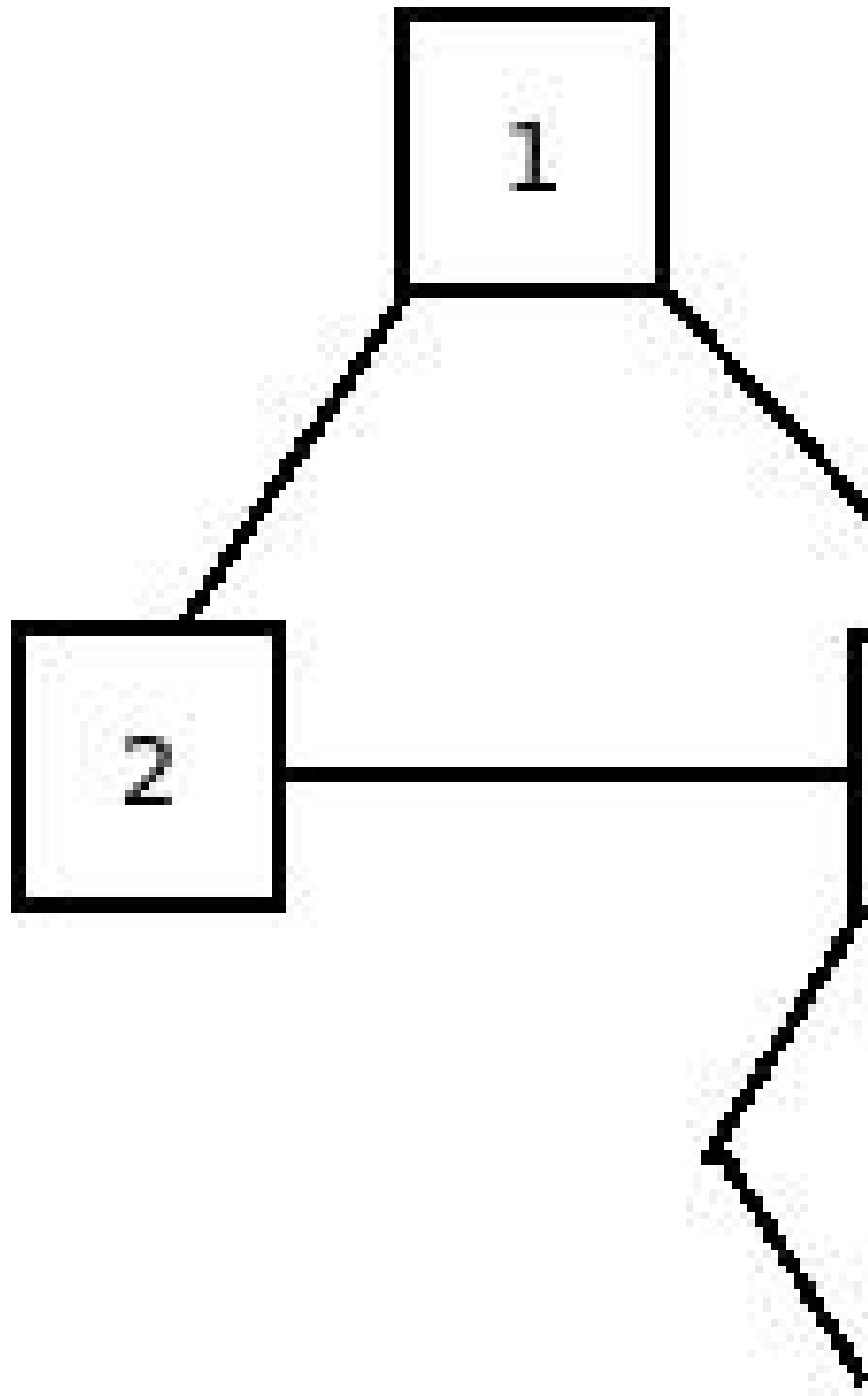
Nodes are labeled uniquely.

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<UndirectedGraphNode, UndirectedGraphNode>();
        Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGraphNode>();
        q.add(node); // added first node, need add its all Neighbors
        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 clones
                if (!map.containsKey(aNbr)) {
                    UndirectedGraphNode acpNbr = new UndirectedGraphNode(aNbr.label);
                    map.put(aNbr, acpNbr);
                    map.get(curr).neighbors.add(acpNbr);
                    q.add(aNbr);
                } else
                    map.get(curr).neighbors.add(map.get(aNbr));
            }
        }
        return res;
    }
}

```

3. BFS:

18.2 Word Ladder, Word Ladder II: Backtracing

18.2.1 Word Ladder

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

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

For example,

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

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

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

Note:

- Return 0 if there is no such transformation sequence.

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

18.2.2 Word Ladder II

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

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

For example,

Given:

start = "hit", end = "cog"

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

Return

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

Note:

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

18.2.3 Check whether the graph is bigraph

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

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

2. Topological Sorting vs Depth First Traversal (DFS): In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the above graph is "5 2 3 1 0 4", but it is not a topological sorting.
3. Algorithm to find Topological Sorting: We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note

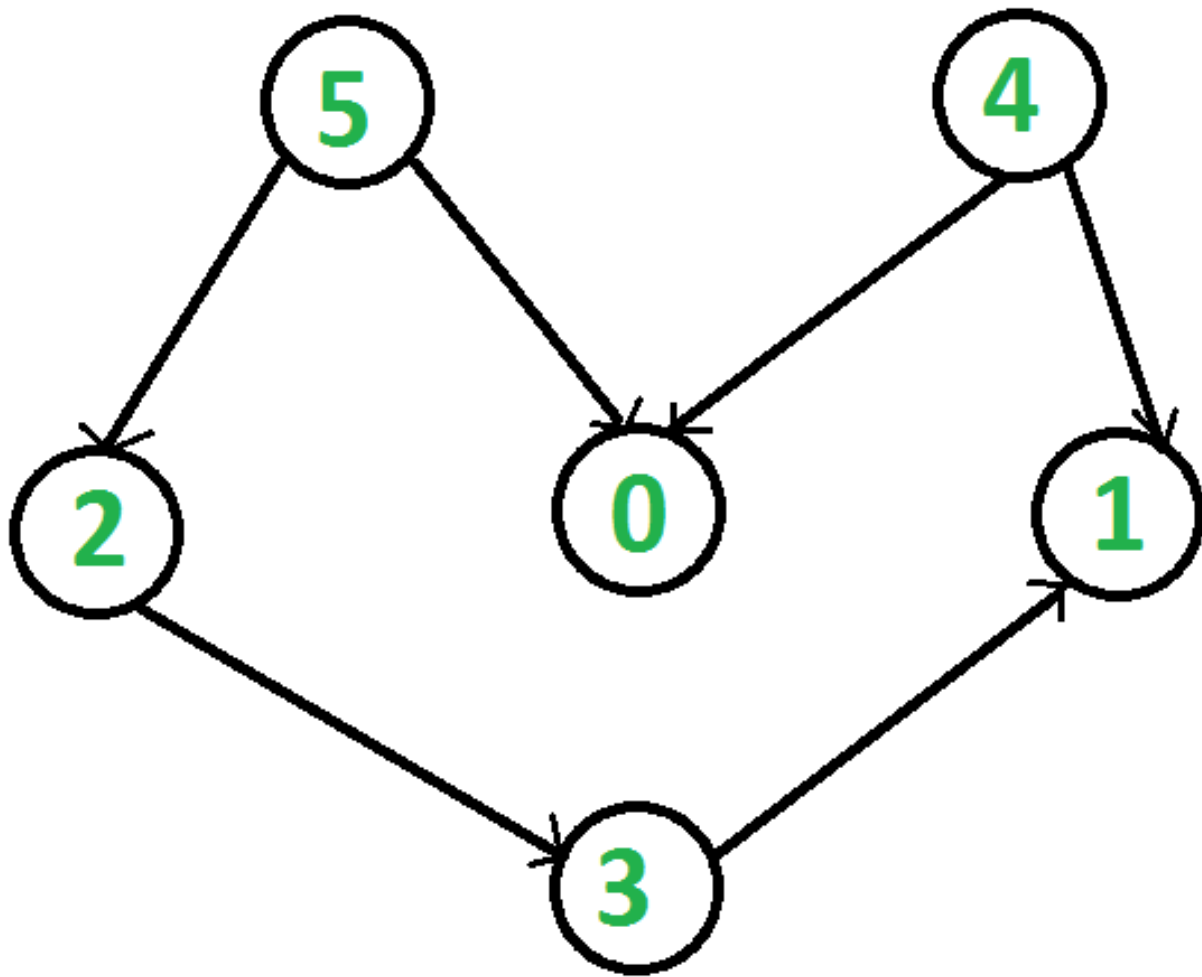


Figure 18.1: Topological Sorting

that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Chapter 19

Data Structure

19.1 Easy

19.1.1 Two Sum III

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

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

For example,

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

Tags: Hash Table, Data Structure

19.1.2 Min Stack

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

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

```
public static class MinStack {  
    Stack<Integer> stack = new Stack<Integer>();  
    Stack<Integer> minStack = new Stack<Integer>();  
    public void push(int x) {  
        stack.push(x);  
        if (minStack.isEmpty() || x <= minStack.peek()) {  
            minStack.push(x);  
        }  
    }  
}
```

```

public void pop() { // java boxing & unboxing, container, object specific methods
    if (stack.peek().intValue() == minStack.peek().intValue())
        minStack.pop();
    stack.pop();
}

public int top() {
    return stack.peek();
}

public int getMin() {
    if (!minStack.isEmpty()) return minStack.peek();
    else return -1;
}
}

```

19.2 Hard

19.2.1 LRU Cache

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

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

```

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

    private HashMap<Integer, Node> hash;
    private int cap;
    private int number;
    Node head;
    Node tail;
    public LRUCache(int capacity) {
        cap = capacity;
        number = 0;
    }
}

```



```

    head = new Node(-1, -1);
    head.prev = null;
    head.next = null;
    tail = head;
    hash = new HashMap<Integer, Node>(capacity); // so I can restrict a size !!
}

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

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

    Node temp = head.next; // head node in the hash
    Node prevNode = tmp.prev;
    Node nextNode = tmp.next;
    // set to be most recently used~~ move the tmp node to be head.next, 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

Chapter 21

Math

21.1 Easy

21.1.1 Add Binary

21.1.2 Roman to Integers

21.1.3 String to Integer (atoi)

21.1.4 Palindrome Number

21.1.5 Plus One

21.1.6 Factorial Trailing Zeroes

21.1.7 Excel Sheet Column Title

21.1.8 Excel Sheet Column Number

21.1.9 Reverse Integer

21.2 Medium

21.2.1 Multiply Strings

21.2.2 Sqrt(x)

21.2.3 Divide Two Integers

21.2.4 Pow(x, n)

21.2.5 Fraction to Recurring Decimal

21.2.6 Permutation Sequence

21.2.7 Integer to Roman

21.2.8 Next Permutation: Math

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

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

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

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

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

21.3 Hard

21.3.1 Valid Number

21.3.2 Max Points on a Line