Table of Contents

算法珠玑——一个最精简的题库

本书的目标读者是准备去硅谷找工作的码农，也适用于在国内找工作的码农，以及刚接触ACM算法竞赛的新手。

市场上讲解算法的书已经汗牛充栋，为什么还要写这本书呢？主要原因是我对目前市场上的大部分算法书都不太满意。 本书有如下特色：

1. 背后有强大的AlgoHub支持。

   本书的所有题目，都可以在 www.algohub.org(即将上线) 上在线判断代码。这样的一大好处是，读者可以边看书，边实现自己的代码，然后提交到网站上验证自己的想法是否正确。AlgoHub的使命是成为最好的算法学习和交流平台。AlgoHub囊括了 POJ, ZOJ, leetcode, HackerRank 等网站的经典题目（一些质量不高的题目则忽略），且 AlgoHub有非常简单的加题系统，用户不需要写一行代码即可自己添加题目，所以AlgoHub的题库还在飞速增长中。

2. 每道题都有完整的代码。

   市场上的大部分书，都会讲思路，但给出的代码都是片段，不是完整可编译的代码。本书每题都有完整的代码，且每个代码经过千锤百炼，保证可读性的前提下尽可能简短，方面读者在面试中能快速写出来。

3. 每道题都有多种解法。

   本书的宗旨是，用尽可能少的题目，覆盖尽可能多的算法。本书中的的每道题都有多种解法，每种解法不是简单的小改进，而是完全不同的思路，力求举一反三，让读者触类旁通。

4. 本书支持多种主流编程语言。

   目前支持 Java, C++, C#, Python, Ruby, JavaScript, Swift, Scala, Clojure, 将来还会支持更多编程语言。

在线阅读

https://www.gitbook.com/book/soulmachine/algorithm-essentials/

内容目录

    - Best Time to Buy and Sell Stock with Cooldown
    - Interleaving String
    - Scramble String
    - Minimum Path Sum
    - Edit Distance
    - Decode Ways
    - Distinct Subsequences
    - Word Break
    - Word Break II
    - Dungeon Game
    - House Robber
    - House Robber II
    - House Robber III
    - Range Sum Query - Immutable
    - Range Sum Query 2D - Immutable
- 图
    - Clone Graph
- 位操作
    - Reverse Bits
    - Repeated DNA Sequences
    - Number of 1 Bits
    - Gray Code
    - Single Number
    - Single Number II
    - Single Number III
    - Power of Two
    - Missing Number
    - Maximum Product of Word Lengths
    - Bitwise AND of Numbers Range
    - Power of Three
    - Rectangle Area
- 数论
    - Happy Number
    - Ugly Number
    - Ugly Number II
    - Super Ugly Number
    - Fraction to Recurring Decimal
    - Factorial Trailing Zeroes
    - Nim Game
- 模拟
    - Reverse Integer
    - Palindrome Number
    - Insert Interval
    - Merge Intervals
    - Minimum Window Substring
    - Multiply Strings

    - Substring with Concatenation of All Words
    - Pascal's Triangle
    - Pascal's Triangle II
    - Spiral Matrix
    - Spiral Matrix II
    - ZigZag Conversion
    - Divide Two Integers
    - Text Justification
    - Max Points on a Line
- Java集合框架总结

Community

Github: https://www.github.com/soulmachine/algorithm-essentials

微博：@灵魂机器

小密圈：

License

这类题目考察线性表的操作，例如，数组，单链表，双向链表等。

本节主要讲关于数组的各种算法题。

Remove Duplicates from Sorted Array

描述

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

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

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

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

分析

无

代码

```
// Remove Duplicates from Sorted Array
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        int index = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] != nums[index-1])
                nums[index++] = nums[i];
        }
        return index;
    }
};
```

相关题目

- Remove Duplicates from Sorted Array II

Remove Duplicates from Sorted Array II

描述

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

For example, given sorted array `A = [1,1,1,2,2,3]`, your function should return length = `5`, and A is now `[1,1,2,2,3]`

分析

加一个变量记录一下元素出现的次数即可。这题因为是已经排序的数组，所以一个变量即可解决。如果是没有排序的数组，则需要引入一个hashmap来记录出现次数。

代码1

```
// Remove Duplicates from Sorted Array II
// Time complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length <= 2) return nums.length;

        int index = 2;
        for (int i = 2; i < nums.length; i++){
            if (nums[i] != nums[index - 2])
                nums[index++] = nums[i];
        }

        return index;
    }
};
```

代码2

下面是一个更简洁的版本。上面的代码略长，不过扩展性好一些，例如将 `occur < 2` 改为 `occur < 3`，就变成了允许重复最多3次。

```java
// Remove Duplicates from Sorted Array II
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int removeDuplicates(int[] nums) {
        int n = nums.length;
        int index = 0;
        for (int i = 0; i < n; ++i) {
            if (i > 0 && i < n - 1 && nums[i] == nums[i - 1] && nums[i] == nums[i + 1])
                continue;

            nums[index++] = nums[i];
        }
        return index;
    }
};
```

相关题目

- Remove Duplicates from Sorted Array

Longest Consecutive Sequence

描述

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, Given `[100, 4, 200, 1, 3, 2]`, The longest consecutive elements sequence is `[1, 2, 3, 4]`. Return its length: 4.

Your algorithm should run in `O(n)` complexity.

分析

如果允许$O(n \log n)$的复杂度，那么可以先排序，可是本题要求 `O(n)` 。

由于序列里的元素是无序的，又要求 `O(n)` ，首先要想到用哈希表。

用一个哈希表存储所有出现过的元素，对每个元素，以该元素为中心，往左右扩张，直到不连续为止，记录下最长的长度。

代码

```
// Longest Consecutive Sequence
// Time Complexity: O(n)，Space Complexity: O(n)
public class Solution {
    public int longestConsecutive(int[] nums) {
        final HashSet<Integer> mySet = new HashSet<Integer>();
        for (int i : nums) mySet.add(i);

        int longest = 0;
        for (int i : nums) {
            int length = 1;
            for (int j = i - 1; mySet.contains(j); --j) {
                mySet.remove(j);
                ++length;
            }
            for (int j = i + 1; mySet.contains(j); ++j) {
                mySet.remove(j);
                ++length;
            }
            longest = Math.max(longest, length);
        }
        return longest;
    }
}
```

Two Sum

描述

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

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

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

Input: `numbers={2, 7, 11, 15}, target=9`

Output: `index1=1, index2=2`

分析

方法1：暴力，复杂度$O(n^2)$，会超时

方法2：hash。用一个哈希表，存储每个数对应的下标，复杂度 `O(n)`.

方法3：先排序，然后左右夹逼，排序$O(n \log n)$，左右夹逼 `O(n)` ，最终$O(n \log n)$。但是注意，这题需要返回的是下标，而不是数字本身，因此这个方法行不通。

代码

```
// Two Sum
// 方法2：hash。用一个哈希表，存储每个数对应的下标
// Time Complexity: O(n)，Space Complexity: O(n)
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        final HashMap<Integer, Integer> myMap = new HashMap<Integer, Integer>();
        int[] result = new int[2];
        for (int i = 0; i < nums.length; i++) {
            myMap.put(nums[i],i);
        }
        for (int i = 0; i < nums.length; i++) {
            final Integer v = myMap.get(target-nums[i]);
            if (v != null && v > i) {
                return new int[]{i+1, v+1};
            }
        }
        return null;
    }
};
```

相关题目

- 3Sum
- 3Sum Closest
- 4Sum

3Sum

描述

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

Note:

- Elements in a triplet `(a,b,c)` must be in non-descending order. (ie, $a \leq b \leq c$)
- The solution set must not contain duplicate triplets.

For example, given array `S = {-1 0 1 2 -1 -4}`.

A solution set is:

```
(-1, 0, 1)
(-1, -1, 2)
```

分析

先排序，然后左右夹逼，复杂度 $O(n^2)$。

这个方法可以推广到 `k-sum`，先排序，然后做 `k-2` 次循环，在最内层循环左右夹逼，时间复杂度是 $O(\max\{n \log n, n^{k-1}\})$。

代码

```
// 3Sum
// 先排序，然后左右夹逼，注意跳过重复的数
// Time Complexity: O(n^2)，Space Complexity: O(1)
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        if (nums.length < 3) return result;
        Arrays.sort(nums);
        final int target = 0;

        for (int i = 0; i < nums.length - 2; ++i) {
            if (i > 0 && nums[i] == nums[i-1]) continue;
            int j = i+1;
            int k = nums.length-1;
            while (j < k) {
                if (nums[i] + nums[j] + nums[k] < target) {
                    ++j;
                    while(nums[j] == nums[j-1] && j < k) ++j;
                } else if(nums[i] + nums[j] + nums[k] > target) {
                    --k;
                    while(nums[k] == nums[k+1] && j < k) --k;
                } else {
                    result.add(Arrays.asList(nums[i], nums[j], nums[k]));
                    ++j;
                    --k;
                    while(nums[j] == nums[j-1] && j < k) ++j;
                    while(nums[k] == nums[k+1] && j < k) --k;
                }
            }
        }
        return result;
    }
};
```

相关题目

- Two sum
- 3Sum Closest
- 4Sum

3Sum Closest

描述

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

For example, given array `S = {-1 2 1 -4}`, and `target = 1`.

The sum that is closest to the target is 2. (`-1 + 2 + 1 = 2`).

分析

先排序，然后左右夹逼，复杂度 $O(n^2)$。

代码

```
// 3Sum Closest
// 先排序，然后左右夹逼
// Time Complexity: O(n^2), Space Complexity: O(1)
public class Solution {
    public int threeSumClosest(int[] nums, int target) {
        int result = 0;
        int minGap = Integer.MAX_VALUE;
        Arrays.sort(nums);

        for (int i = 0; i < nums.length - 1; ++i) {
            int j = i + 1;
            int k = nums.length - 1;

            while(j < k) {
                final int sum = nums[i] + nums[j] + nums[k];
                final int gap = Math.abs(sum - target);
                if (gap < minGap) {
                    result = sum;
                    minGap = gap;
                }

                if (sum < target) ++j;
                else              --k;
            }
        }
        return result;
    }
}
```

相关题目

- Two sum
- 3Sum
- 4Sum

4Sum

描述

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

Note:

- Elements in a quadruplet `(a,b,c,d)` must be in non-descending order. (ie, $a \leq b \leq c \leq d$)
- The solution set must not contain duplicate quadruplets.

For example, given array `S = {1 0 -1 0 -2 2}` , and `target = 0` .

A solution set is:

```
(-1,  0, 0, 1)
(-2, -1, 1, 2)
(-2,  0, 0, 2)
```

分析

先排序，然后左右夹逼，复杂度 $O(n^3)$，会超时。

可以用一个hashmap先缓存两个数的和，最终复杂度$O(n^3)$。这个策略也适用于 3Sum 。

左右夹逼

```java
// 4Sum
// 先排序，然后左右夹逼
// Time Complexity: O(n^3)，Space Complexity: O(1)
public class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        if (nums.length < 4) return result;
        Arrays.sort(nums);

        for (int i = 0; i < nums.length - 3; ++i) {
            if (i > 0 && nums[i] == nums[i-1]) continue;
            for (int j = i + 1; j < nums.length - 2; ++j) {
                if (j > i+1 && nums[j] == nums[j-1]) continue;
                int k = j + 1;
                int l = nums.length - 1;
                while (k < l) {
                    final int sum = nums[i] + nums[j] + nums[k] + nums[l];
                    if (sum < target) {
                        ++k;
                        while(nums[k] == nums[k-1] && k < l) ++k;
                    } else if (sum > target) {
                        --l;
                        while(nums[l] == nums[l+1] && k < l) --l;
                    } else {
                        result.add(Arrays.asList(nums[i], nums[j], nums[k], nums[l]));
                        ++k;
                        --l;
                        while(nums[k] == nums[k-1] && k < l) ++k;
                        while(nums[l] == nums[l+1] && k < l) --l;
                    }
                }
            }
        }
        return result;
    }
}
```

HashMap 做缓存

```
// 4Sum
// 先排序，然后左右夹逼
// Time Complexity: O(n^3)，Space Complexity: O(1)
public class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        if (nums.length < 4) return result;
        Arrays.sort(nums);

        final HashMap<Integer, ArrayList<int[]>> cache = new HashMap<>();
        for (int i = 0; i < nums.length; ++i) {
            for (int j = i + 1; j < nums.length; ++j) {
                ArrayList<int[]> value = cache.get(nums[i] + nums[j]);
                if (value == null) {
                    value = new ArrayList<>();
                    cache.put(nums[i] + nums[j], value);
                }
                value.add(new int[]{i, j});
            }
        }

        final HashSet<String> used = new HashSet<>(); // avoid duplicates
        for (int i = 0; i < nums.length; ++i) {
            if (i > 0 && nums[i] == nums[i-1]) continue;
            for (int j = i + 1; j < nums.length - 2; ++j) {
                if (j > i+1 && nums[j] == nums[j-1]) continue;
                final ArrayList<int[]> list = cache.get(target - nums[i] - nums[j]);
                if (list == null) continue;;
                for (int[] pair : list) {
                    if (j >= pair[0]) continue;  // overlap

                    final Integer[] sol = new Integer[]{nums[i], nums[j], nums[pair[0]]
, nums[pair[1]]};
                    Arrays.sort(sol);
                    final String key = Arrays.toString(sol);

                    if(!used.contains(key)){
                        result.add(Arrays.asList(sol));
                        used.add(key);
                    }
                }
            }
        }
        return result;
    }
}
```

相关题目

- Two sum
- 3Sum

- 3Sum Closest

Remove Element

描述

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

分析

无

代码

```
// Remove Element
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int removeElement(int[] nums, int target) {
        int index = 0;
        for (int i = 0; i < nums.length; ++i) {
            if (nums[i] != target) {
                nums[index++] = nums[i];
            }
        }
        return index;
    }
};
```

相关题目

- Move Zeroes

Move Zeroes

描述

Given an array `nums`, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given `nums = [0, 1, 0, 3, 12]`, after calling your function, nums should be `[1, 3, 12, 0, 0]`.

Note:

1. You must do this in-place without making a copy of the array.
2. Minimize the total number of operations.

分析

这题跟 "Remove Element" 思路一模一样，只是最后要把后半截设置为0。

代码

```
// Move Zeroes
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public void moveZeroes(int[] nums) {
        int index = 0;
        for (int i = 0; i < nums.length; ++i) {
            if (nums[i] != 0) {
                nums[index++] = nums[i];
            }
        }
        for (int i = index; i < nums.length; ++i) {
            nums[i] = 0;
        }
    }
}
```

相关题目

- Remove Element

Next Permutation

描述

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
```

分析

算法过程如下图所示（来自http://fisherlei.blogspot.com/2012/12/leetcode-next-permutation.html）。

Example	6	8	7	4	3	2
Step 1	6	8	7	4	3	2
Step 2	6	8	7	4	3	2
Step 3	7	8	6	4	3	2
Step 4	7	8	6	4	3	2
	7	2	3	4	6	8

1. From right to left, find the first digit (PartitionNumber)which violate the increase trend, in this example, 6 will be selected since 8,7,4,3,2 already in a increase trend.
2. From right to left, find the first digit which large than PartitionNumber, call it changeNumber. Here the 7 will be selected.
3. Swap the PartitionNumber and ChangeNumber.
4. Reverse all the digit on the right of partition index.

Figure: 下一个排列算法流程

代码

```java
// Next Permutation
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public void nextPermutation(int[] nums) {
        nextPermutation(nums, 0, nums.length);
    }
    private static boolean nextPermutation(int[] nums, int begin, int end) {
        // From right to left, find the first digit(partitionNumber)
        // which violates the increase trend
        int p = end - 2;
        while (p > -1 && nums[p] >= nums[p + 1]) --p;

        // If not found, which means current sequence is already the largest
        // permutation, then rearrange to the first permutation and return false
        if(p == -1) {
            reverse(nums, begin, end);
            return false;
        }

        // From right to left, find the first digit which is greater
        // than the partition number, call it changeNumber
        int c = end - 1;
        while (c > 0 && nums[c] <= nums[p]) --c;

        // Swap the partitionNumber and changeNumber
        swap(nums, p, c);
        // Reverse all the digits on the right of partitionNumber
        reverse(nums, p+1, end);
        return true;
    }
    private static void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
    private static void reverse(int[] nums, int begin, int end) {
        end--;
        while (begin < end) {
            swap(nums, begin++, end--);
        }
    }
};
```

相关题目

- Permutation Sequence
- Permutations
- Permutations II
- Combinations

Permutation Sequence

描述

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

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

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

Given `n` and `k`, return the kth permutation sequence.

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

分析

首先可以想到一个简单直白的方法，即调用 `k-1` 次 `next_permutation()`，从而得到第 `k` 个排列。这个方法把前 `k` 个排列全部求出来了，比较浪费，时间复杂度是 `O(kn)`，所以会超时。有没有办法直接求第 `k` 个排列呢？有！

利用康托编码的思路，假设有 `n` 个不重复的元素，第 `k` 个排列是$a_1, a_2, a_3, ..., a_n$，那么a_1是哪一个位置呢？

我们把a_1去掉，那么剩下的排列为 $a_2, a_3, ..., a_n$, 共计 `n-1` 个元素， `n-1` 个元素共有 `(n-1)!` 个排列，于是就可以知道 $a_1 = k/(n-1)!$。

同理，$a_2, a_3, ..., a_n$ 的值推导如下：

$k_2 = k\%(n-1)!$

$a_2 = k_2/(n-2)!$

$\cdots$

$k_{n-1} = k_{n-2}\%2!$

$a_{n-1} = k_{n-1}/1!$

$a_n = 0$

康托编码

```
// Permutation Sequence
// 康托编码
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public String getPermutation(int n, int k) {
        string s(n, '0');
        string result;
        for (int i = 0; i < n; ++i)
            s[i] += i + 1;

        return kth_permutation(s, k);
    }
private:
    int factorial(int n) {
        int result = 1;
        for (int i = 1; i < n+1; ++i)
            result *= i;
        return result;
    }

    // s 已排好序，是第一个排列
    string kth_permutation(string &s, int k) {
        const int n = s.size();
        string result;

        int base = factorial(n - 1);
        --k;  // 康托编码从0开始

        for (int i = n - 1; i > 0; k %= base, base /= i, --i) {
            auto a = next(s.begin(), k / base);
            result.push_back(*a);
            s.erase(a);
        }

        result.push_back(s[0]); // 最后一个
        return result;
    }
};
```

相关题目

- Next Permutation
- Permutations
- Permutations II
- Combinations

Valid Sudoku

描述

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules http://sudoku.com.au/TheRules.aspx .

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

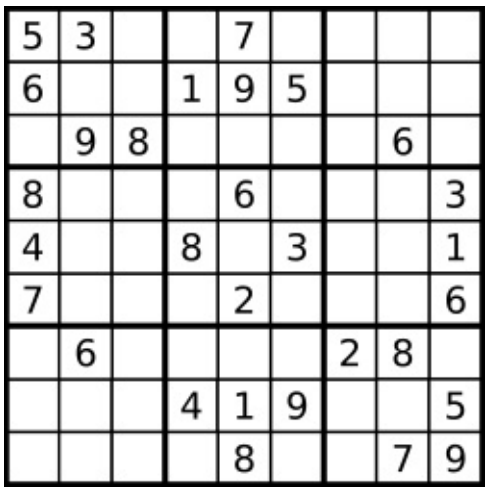


Figure: Valid Sudoku

分析

细节实现题。

代码

```java
// Valid Sudoku
// 时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public boolean isValidSudoku(char[][] board) {
        boolean[] used = new boolean[9];

        for (int i = 0; i < 9; ++i) {
            Arrays.fill(used, false);

            for (int j = 0; j < 9; ++j) // 检查行
                if (!check(board[i][j], used))
                    return false;

            Arrays.fill(used, false);

            for (int j = 0; j < 9; ++j) // 检查列
                if (!check(board[j][i], used))
                    return false;
        }

        for (int r = 0; r < 3; ++r) // 检查 9 个子格子
            for (int c = 0; c < 3; ++c) {
                Arrays.fill(used, false);

                for (int i = r * 3; i < r * 3 + 3; ++i)
                    for (int j = c * 3; j < c * 3 + 3; ++j)
                        if (!check(board[i][j], used))
                            return false;
            }

        return true;
    }

    private static boolean check(char ch, boolean[] used) {
        if (ch == '.') return true;

        if (used[ch - '1']) return false;

        return used[ch - '1'] = true;
    }
};
```

相关题目

- Sudoku Solver

Trapping Rain Water

描述

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.

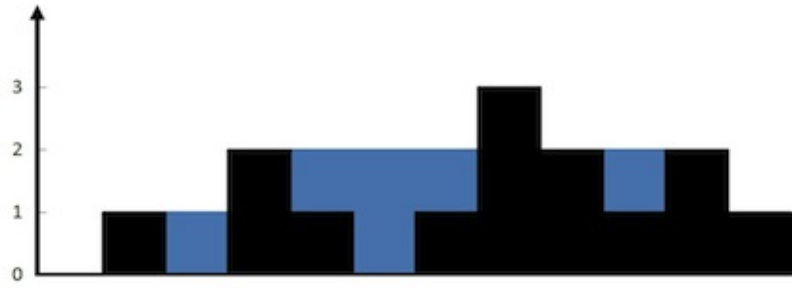


Figure: Trapping Rain Water

分析

对于每个柱子，找到其左右两边最高的柱子，该柱子能容纳的面积就是`min(max_left, max_right) - height`。所以，

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

也可以，

1. 扫描一遍，找到最高的柱子，这个柱子将数组分为两半；
2. 处理左边一半；
3. 处理右边一半。

代码1

```
// Trapping Rain Water
// 思路1，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int trap(int[] A) {
        final int n = A.length;
        int[] left_peak = new int[n];
        int[] right_peak = new int[n];

        for (int i = 1; i < n; i++) {
            left_peak[i] = Math.max(left_peak[i-1], A[i-1]);
        }
        for (int i = n - 2; i >=0; --i) {
            right_peak[i] = Math.max(right_peak[i+1], A[i+1]);
        }

        int sum = 0;
        for (int i = 0; i < n; i++) {
            int height = Math.min(left_peak[i], right_peak[i]);
            if (height > A[i]) {
                sum += height - A[i];
            }
        }

        return sum;
    }
};
```

代码2

```
// Trapping Rain Water
// 思路2，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int trap(int[] A) {
        final int n = A.length;
        int peak_index = 0; // 最高的柱子，将数组分为两半
        for (int i = 0; i < n; i++)
            if (A[i] > A[peak_index]) peak_index = i;

        int water = 0;
        for (int i = 0, left_peak = 0; i < peak_index; i++) {
            if (A[i] > left_peak) left_peak = A[i];
            else water += left_peak - A[i];
        }
        for (int i = n - 1, right_peak = 0; i > peak_index; i--) {
            if (A[i] > right_peak) right_peak = A[i];
            else water += right_peak - A[i];
        }
        return water;
    }
};
```

相关题目

- Container With Most Water
- Largest Rectangle in Histogram

Rotate Image

描述

You are given an `n × n` 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up: Could you do this in-place?

分析

首先想到，纯模拟，从外到内一圈一圈的转，但这个方法太慢。

如下图，首先沿着副对角线翻转一次，然后沿着水平中线翻转一次。

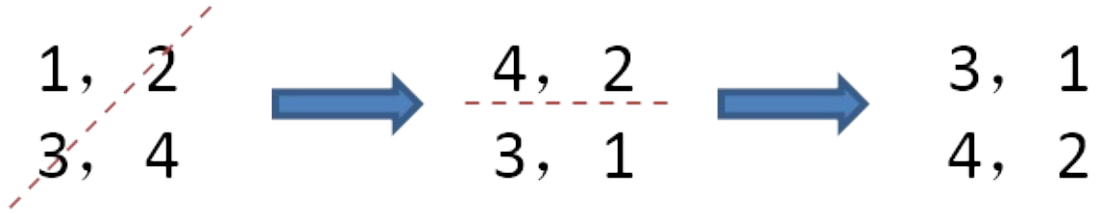


Figure: Rotate image

或者，首先沿着水平中线翻转一次，然后沿着主对角线翻转一次。

代码1

```
// Rotate Image
// 思路 1，时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public void rotate(final int[][] matrix) {
        final int n = matrix.length;

        for (int i = 0; i < n; ++i)  // 沿着副对角线反转
            for (int j = 0; j < n - i; ++j)
                swap(matrix, i, j, n - 1 - j, n - 1 - i);

        for (int i = 0; i < n / 2; ++i) // 沿着水平中线反转
            for (int j = 0; j < n; ++j)
                swap(matrix, i, j, n - 1 - i, j);
    }
    private static void swap(final int[][] matrix,
            int i, int j, int p, int q) {
        int tmp = matrix[i][j];
        matrix[i][j] = matrix[p][q];
        matrix[p][q] = tmp;
    }
};
```

代码2

```
// Rotate Image
// 思路 2，时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public void rotate(final int[][] matrix) {
        final int n = matrix.length;

        for (int i = 0; i < n / 2; ++i) // 沿着水平中线反转
            for (int j = 0; j < n; ++j)
                swap(matrix, i, j, n - 1 - i, j);

        for (int i = 0; i < n; ++i)  // 沿着主对角线反转
            for (int j = i + 1; j < n; ++j)
                swap(matrix, i, j, j, i);
    }
    private static void swap(final int[][] matrix,
            int i, int j, int p, int q) {
        int tmp = matrix[i][j];
        matrix[i][j] = matrix[p][q];
        matrix[p][q] = tmp;
    }
};
```

Plus One

描述

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

分析

高精度加法。

代码

```
// Plus One
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int[] plusOne(int[] digits) {
        return add(digits, 1);
    }
    private static int[] add(int[] digits, int digit) {
        int c = digit;  // carry, 进位

        for (int i = digits.length - 1; i >= 0; --i) {
            digits[i] += c;
            c = digits[i] / 10;
            digits[i] %= 10;
        }

        if (c > 0) { // assert (c == 1)
            int[] tmp = new int[digits.length + 1];
            System.arraycopy(digits, 0, tmp, 1, digits.length);
            tmp[0] = c;
            return tmp;
        } else {
            return digits;
        }
    }
};
```

Climbing Stairs

描述

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?

分析

设 `f(n)` 表示爬 `n` 阶楼梯的不同方法数，为了爬到第 `n` 阶楼梯，有两个选择：

- 从第 `n-1` 阶前进1步；
- 从第 `n-1` 阶前进2步；

因此，有 `f(n)=f(n-1)+f(n-2)` 。

这是一个斐波那契数列。

方法1，递归，太慢；方法2，迭代。

方法3，数学公式。斐波那契数列的通项公式为 $a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$ 。

迭代

```
// Climbing Stairs
// 迭代，时间复杂度O(n)，空间复杂度O(1)
class Solution {
public:
    int climbStairs(int n) {
        int prev = 0;
        int cur = 1;
        for(int i = 1; i <= n ; ++i){
            int tmp = cur;
            cur += prev;
            prev = tmp;
        }
        return cur;
    }
};
```

数学公式

```
// Climbing Stairs
// 数学公式，时间复杂度O(1)，空间复杂度O(1)
public class Solution {
    public int climbStairs(int n) {
        final double s = Math.sqrt(5);
        return (int)Math.floor((Math.pow((1+s)/2, n+1) +
            Math.pow((1-s)/2, n+1))/s + 0.5);
    }
};
```

相关题目

- Decode Ways

Set Matrix Zeroes

描述

Given a `m × n` matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Follow up: Did you use extra space?

A straight forward solution using `O(mn)` space is probably a bad idea.

A simple improvement uses `O(m + n)` space, but still not the best solution.

Could you devise a constant space solution?

分析

`O(m+n)` 空间的方法很简单，设置两个bool数组，记录每行和每列是否存在0。

想要常数空间，可以复用第一行和第一列。

代码1

```java
// Set Matrix Zeroes
// 时间复杂度O(m*n)，空间复杂度O(m+n)
public class Solution {
    public void setZeroes(int[][] matrix) {
        final int m = matrix.length;
        final int n = matrix[0].length;
        boolean[] row = new boolean[m]; // 标记该行是否存在0
        boolean[] col = new boolean[n]; // 标记该列是否存在0

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    row[i] = col[j] = true;
                }
            }
        }

        for (int i = 0; i < m; ++i) {
            if (row[i]) Arrays.fill(matrix[i], 0);
        }
        for (int j = 0; j < n; ++j) {
            if (col[j]) {
                for (int i = 0; i < m; ++i) {
                    matrix[i][j] = 0;
                }
            }
        }
    }
}
```

代码2

```java
// Set Matrix Zeroes
// 时间复杂度O(m*n)，空间复杂度O(1)
public class Solution {
    public void setZeroes(int[][] matrix) {
        final int m = matrix.length;
        final int n = matrix[0].length;
        boolean row_has_zero = false; // 第一行是否存在 0
        boolean col_has_zero = false; // 第一列是否存在 0

        for (int i = 0; i < n; i++)
            if (matrix[0][i] == 0) {
                row_has_zero = true;
                break;
            }

        for (int i = 0; i < m; i++)
            if (matrix[i][0] == 0) {
                col_has_zero = true;
                break;
            }

        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                if (matrix[i][j] == 0) {
                    matrix[0][j] = 0;
                    matrix[i][0] = 0;
                }
        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                if (matrix[i][0] == 0 || matrix[0][j] == 0)
                    matrix[i][j] = 0;
        if (row_has_zero)
            for (int i = 0; i < n; i++)
                matrix[0][i] = 0;
        if (col_has_zero)
            for (int i = 0; i < m; i++)
                matrix[i][0] = 0;
    }
};
```

Gas Station

描述

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.

分析

首先想到的是$O(N^2)$的解法，对每个点进行模拟。

`O(N)`的解法是，设置两个变量，`sum`判断当前的指针的有效性；`total`则判断整个数组是否有解，有就返回通过`sum`得到的下标，没有则返回-1。

代码

```
// Gas Station
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int total = 0;
        int j = -1;
        for (int i = 0, sum = 0; i < gas.length; ++i) {
            sum += gas[i] - cost[i];
            total += gas[i] - cost[i];
            if (sum < 0) {
                j = i;
                sum = 0;
            }
        }
        return total >= 0 ? j + 1 : -1;
    }
};
```

Candy

描述

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?

分析

无

迭代版

```
// Candy
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int candy(int[] ratings) {
        final int n = ratings.length;
        final int[] increment = new int[n];

        // 左右各扫描一遍
        for (int i = 1, inc = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1])
                increment[i] = Math.max(inc++, increment[i]);
            else
                inc = 1;
        }

        for (int i = n - 2, inc = 1; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1])
                increment[i] = Math.max(inc++, increment[i]);
            else
                inc = 1;
        }
        // 初始值为n，因为每个小朋友至少一颗糖
        int sum = n;
        for (int i : increment) sum += i;
        return sum;
    }
};
```

递归版

```java
// Candy
// 备忘录法，时间复杂度O(n)，空间复杂度O(n)
// java.lang.StackOverflowError
public class Solution {
    public int candy(int[] ratings) {
        final int[] f = new int[ratings.length];
        int sum = 0;
        for (int i = 0; i < ratings.length; ++i)
            sum += solve(ratings, f, i);
        return sum;
    }
    int solve(int[] ratings, int[] f, int i) {
        if (f[i] == 0) {
            f[i] = 1;
            if (i > 0 && ratings[i] > ratings[i - 1])
                f[i] = Math.max(f[i], solve(ratings, f, i - 1) + 1);
            if (i < ratings.length - 1 && ratings[i] > ratings[i + 1])
                f[i] = Math.max(f[i], solve(ratings, f, i + 1) + 1);
        }
        return f[i];
    }
}
```

Majority Element

描述

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.

分析

这题最简单的解法，先把数组排序，`O(nlogn)`，然后从头到尾扫描一遍，找出最长的连续子串。

由于超过 `⌊ n/2 ⌋` 次，可以对上面的方法改进一下，排序后，不需要扫描，直接返回中间那个元素，`nums[n/2]`，肯定就是答案。

上述两个方法都是 `O(nlogn)` 的，本题有一个线性解法。由于超过 `⌊ n/2 ⌋`，可以用相抵消的思想，凡是和最多元素不相等的，就抵消，最后剩下的一定就是最多的那个元素。

解法1 排序

```
// Majority Element
// Time Complexity: O(nlogn), Space Complexity: O(1)
public class Solution {
    public int majorityElement(int[] nums) {
        Arrays.sort(nums);
        return nums[nums.length/2];
    }
}
```

解法2 线性解法

```java
// Majority Element
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int majorityElement(int[] nums) {
        int result = 0;
        int count = 0;

        for (int x : nums) {
            if (count == 0) {
                result = x;
                count = 1;
            } else if (result == x) {
                ++count;
            } else {
                --count;
            }
        }
        return result;
    }
}
```

Rotate Array

描述

Rotate an array of `n` elements to the right by `k` steps.

For example, with `n = 7` and `k = 3`, the array `[1,2,3,4,5,6,7]` is rotated to `[5,6,7,1,2,3,4]`.

Note: Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

分析

最简单的方法，开一个 `k` 长的数组，先把右边 `k` 个元素存入这个临时数组，然后把数组中的前 `n-k` 右移 `k` 位，再把临时数组的 `k` 个元素存入到原始数组左边。时间复杂度 `O(n)` ，空间复杂度 `O(k)` 。

第二个简单的方法，先实现一个函数，把数组右移一位，调用这个函数 `k` 次即可。时间复杂度 `O(n*k)` ，空间复杂度 `O(1)` 。

第三个方法，先将数组分为两段，前 `n-k` 个为一段，后 `k` 个元素作为第二段，将第一段reverse, 第二段reverse, 然后将整个数组reverse, 这样经过三轮reverse，就完成了循环右移。时间复杂度 `O(n)` ，空间复杂度 `O(1)` 。

解法1 三轮reverse

```
// Rotate Array
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public void rotate(int[] nums, int k) {
        k %= nums.length;
        reverse(nums, 0, nums.length - k);
        reverse(nums, nums.length - k, nums.length);
        reverse(nums, 0, nums.length);
    }
    private static void reverse(int[] nums, int begin, int end) {
        int left = begin;
        int right = end - 1;
        while (left < right) {
            // swap
            int tmp = nums[left];
            nums[left] = nums[right];
            nums[right] = tmp;
            ++left;
            --right;
        }
    }
}
```

Contains Duplicate

描述

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

分析

方法1， 用一个 HashSet, 不断往里面塞元素，如果发现有重复，说明存在重复。时间复杂度 `O(n)` ，空间复杂度 `O(n)` 。

方法2， 排序，然后从头到尾扫描，如果发现相邻两个元素相等，则存在重复。时间复杂度 `O(nlogn)` ，空间复杂度 `O(1)` 。

解法1 哈希表

```
// Contains Duplicate
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public boolean containsDuplicate(int[] nums) {
        final Set<Integer> existed = new HashSet<>();
        for (int x : nums) {
            if (existed.contains(x)) {
                return true;
            } else {
                existed.add(x);
            }
        }
        return false;
    }
}
```

解法2 排序

```
// Contains Duplicate
// Time Complexity: O(nlogn), Space Complexity: O(1)
public class Solution {
    public boolean containsDuplicate(int[] nums) {
        Arrays.sort(nums);

        for (int i = 1; i < nums.length; ++i) {
            if (nums[i-1] == nums[i]) return true;
        }
        return false;
    }
}
```

Contains Duplicate II

描述

Given an array of integers and an integer `k`, find out whether there are two distinct indices `i` and `j` in the array such that `nums[i] = nums[j]` and the difference between `i` and `j` is at most `k`.

分析

维护一个HashMap, key为整数，value为下标，将数组中的元素不断添加进这个HashMap, 碰到重复时，计算二者的下标距离，如果距离小于或等于k，则返回true, 如果直到数组扫描完，距离都大于k，则返回false。

代码

```
// Contains Duplicate II
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        final Map<Integer, Integer> map = new HashMap<>();
        int min = Integer.MAX_VALUE;

        for(int i = 0; i < nums.length; i++){
            if(map.containsKey(nums[i])){
                final int preIndex = map.get(nums[i]);
                final int gap = i - preIndex;
                min = Math.min(min, gap);
            }
            map.put(nums[i], i);
        }
        return min <= k;
    }
}
```

Contains Duplicate III

描述

Given an array of integers, find out whether there are two distinct indices `i` and `j` in the array such that the difference between `nums[i]` and `nums[j]` is at most `t` and the difference between `i` and `j` is at most `k`.

分析

这一题比 "Contains Duplicate II" 有多了个条件，难度陡然就增大了。

对于数组内的一个整数，如果能方便的求出大于它的最小整数和小于它的最大整数，那么我们就可以判断差值是否大于 `t`。能方便的求出最大下限和最小上限，最先想到的数据结构是二叉搜索树BST，因为左孩子就是最大下限，右孩子就是最小上限。

可以用一个大小为k的滑动窗口，将窗口内的元素组织成一个BST，每次向前滑动一步，添加一个新元素，同时删除一个最老的元素，如此不断向前滑动，不断更新BST。如果BST内部有两个元素差值大于 `t`，则返回true，如果直到扫描完数组，BST里都没有出现差值大于k的两个数，则返回false。

对于BST数据结构，可以用现成的，C++里是 `multiset`，Java里是 `TreeSet`。

代码 滑动窗口+BST

```
// Contains Duplicate III
// Time Complexity: O(nlogk), Space Complexity: O(k)
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (k < 1 || t < 0) return false;

        final TreeSet<Integer> set = new TreeSet<>();
        for (int i = 0; i < nums.length; i++) {
            final int x = nums[i];
            final Integer floor = set.floor(x);
            final Integer ceiling = set.ceiling(x);

            if ((floor != null && x <= floor + t)
                    || (ceiling != null && x >= ceiling -t))
                return true;

            set.add(x);
            if (i >= k) set.remove(nums[i - k]);
        }

        return false;
    }
}
```

Product of Array Except Self

描述

Given an array of `n` integers where `n > 1`, `nums`, return an array output such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it **without division** and in `O(n)`.

For example, given `[1,2,3,4]`, return `[24,12,8,6]`.

Follow up:

Could you solve it with constant space complexity? (Note: The output array does not count as extra space for the purpose of space complexity analysis.)

分析

我们以一个4个元素的数组为例，`nums=[a1,a2,a3,a4]`，要想在`O(n)`的时间内输出结果，比较好的解决方法是提前构造好两个数组：

- `[1, a1, a1*a2, a1*a2*a3]`
- `[a2*a3*a4, a3*a4, a4, 1]`

然后两个数组一一对应相乘，即可得到最终结果`[a2*a3*a4, a1*a3*a4, a1*a2*a4, a1*a2*a3]`。

不过，上述方法的空间复杂度为`O(n)`，可以进一步优化成常数空间，即用一个整数代替第二个数组。

代码1 O(n)空间

```java
// Product of Array Except Self
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        final int[] result = new int[nums.length];
        final int[] left = new int[nums.length];
        final int[] right = new int[nums.length];

        left[0] = 1;
        right[nums.length - 1] = 1;

        for (int i = 1; i < nums.length; ++i) {
            left[i] = nums[i - 1] * left[i - 1];
        }

        for (int i = nums.length - 2; i >= 0; --i) {
            right[i] = nums[i + 1] * right[i + 1];
        }

        for (int i = 0; i < nums.length; ++i) {
            result[i] = left[i] * right[i];
        }

        return result;
    }
}
```

代码2 O(1)空间

```java
// Product of Array Except Self
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        final int[] left = new int[nums.length];
        left[0] = 1;

        for (int i = 1; i < nums.length; ++i) {
            left[i] = nums[i - 1] * left[i - 1];
        }

        int right = 1;
        for (int i = nums.length - 1; i >= 0; --i) {
            left[i] *= right;
            right *= nums[i];
        }
        return left;
    }
}
```

Game of Life

描述

According to the Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

分析

最简单的办法是新建一个矩阵保存下一轮局面。

不过本提要求 inplace, 难度就瞬间增大了。因为我们修改一个位置的值后，它的旧值就丢失了，而它周围还有邻居依赖它的旧值。

因为题目给出的是一个int矩阵，大有空间可以利用。我们可以换一种方式进行编码，假设对于每个点，值的含义为：

- 状态0：死细胞转为死细胞
- 状态1：活细胞转为活细胞
- 状态2：活细胞转为死细胞
- 状态3：死细胞转为活细胞

得到这样一个矩阵后，最后将所有状态对2取模，状态0和2变成死细胞，1和3变成活细胞，就是所求的下一轮局面了。

代码

```java
// Game of Life
// Time complexity: O(mxn), Space complexity: O(1)
public class Solution {
    public void gameOfLife(int[][] board) {
        final int m = board.length;
        final int n = board[0].length;
        // clock wise, start from upper-left corner
        final int[] dx = new int[] {-1, -1, -1, 0, 1, 1, 1, 0};
        final int[] dy = new int[] {-1, 0, 1, 1, 1, 0, -1, -1};

        // encode
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                int live = 0; // number of live cells
                for (int k = 0; k < 8; ++k) {
                    final int x = i + dx[k];
                    final int y = j + dy[k];
                    if (x > -1 && x < m && y > -1 && y < n &&
                            (board[x][y] == 1 || board[x][y] == 2)) {
                        ++live;
                    }
                }
                if (board[i][j] == 0 && live == 3) {
                    board[i][j] = 3;
                } else if (board[i][j] == 1 && (live < 2 || live > 3)) {
                    board[i][j] = 2;
                }
            }
        }

        //decode
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                board[i][j] %= 2;
            }
        }
    }
}
```

Increasing Triplet Subsequence

描述

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

More specifically, if there exists `i` , `j` , `k` such that `arr[i] < arr[j] < arr[k]` given `0 ≤ i < j < k ≤ n-1` return `true` else return `false` .

Your function should run in `O(n)` time complexity and `O(1)` space complexity.

Examples:

- Given `[1, 2, 3, 4, 5]` , return `true` .
- Given `[5, 4, 3, 2, 1]` , return `false` .

分析

扫描一遍数组，用变量 `x1` 保存当前最小的值，变量 `x2` 保存当前第二小的值。如果右面能碰到一个数大于 `x2` ，说明必然存在一个递增的三元组。

代码

```
// Increasing Triplet Subsequence
// Time complexity: O(n), Space complexity: O(1)
public class Solution {
    public boolean increasingTriplet(int[] nums) {
        int x1 = Integer.MAX_VALUE;
        int x2 = Integer.MAX_VALUE;

        for (int x : nums) {
            if (x <= x1) x1 = x;
            else if (x <= x2) x2 = x;
            else return true;
        }
        return false;
    }
}
```

相关题目

- Longest Increasing Subsequence

本节主要讲关于单链表的算法。

单链表节点的定义如下：

```
// 单链表节点
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

Reverse Linked List

描述

Reverse a singly linked list.

分析

用三个指针 `tail`, `p`, `q` ，紧紧相邻，不断前进，每次将 `p.next` 指向 `tail` ，将 `q.next` 指向 `p` 。

解法1 迭代

```
// Reverse Linked List
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) return head;

        ListNode tail = null;
        ListNode p = head;
        ListNode q = p.next;

        while (q != null) {
            ListNode old = q.next;
            p.next = tail;
            q.next = p;

            tail = p;
            p = q;
            q = old;
        }
        return p;
    }
}
```

解法2 递归

```java
// Reverse Linked List
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) return head;

        ListNode tail = head.next;
        head.next = null;
        ListNode newHead = reverseList(tail);
        tail.next = head;

        return newHead;
    }
}
```

Odd Even Linked List

描述

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in `O(1)` space complexity and `O(n)` time complexity.

Example:

Given `1->2->3->4->5->NULL`,

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

Note:

1. The relative order inside both the even and odd groups should remain as it was in the input.
2. The first node is considered odd, the second node even and so on ...

分析

创建两个新的空链表，遍历原始链表，把奇数位置的节点添加到第一个小链表，把偶数位置的节点添加到第二个小链表。

代码

```java
// Odd Even Linked List
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public ListNode oddEvenList(ListNode head) {
        final ListNode oddDummy = new ListNode(0);
        final ListNode evenDummy = new ListNode(0);
        ListNode odd = oddDummy;
        ListNode even = evenDummy;

        int index = 1;
        while (head != null) {
            if (index  % 2 == 1) {
                odd.next = head;
                odd = odd.next;
            } else {
                even.next = head;
                even = even.next;
            }

            ListNode tmp = head.next;
            head.next = null;
            head = tmp;
            ++index;
        }

        odd.next = evenDummy.next;
        return oddDummy.next;
    }
}
```

Add Two Numbers

描述

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

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

Output: `7 -> 0 -> 8`

分析

跟 Add Binary 很类似

代码

```
// Add Two Numbers
// 跟Add Binary 很类似
// 时间复杂度O(m+n)，空间复杂度O(1)
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1); // 头节点
        int carry = 0;
        ListNode prev = dummy;
        for (ListNode pa = l1, pb = l2;
             pa != null || pb != null;
             pa = pa == null ? null : pa.next,
             pb = pb == null ? null : pb.next,
             prev = prev.next) {
            final int ai = pa == null ? 0 : pa.val;
            final int bi = pb == null ? 0 : pb.val;
            final int value = (ai + bi + carry) % 10;
            carry = (ai + bi + carry) / 10;
            prev.next = new ListNode(value); // 尾插法
        }
        if (carry > 0)
            prev.next = new ListNode(carry);
        return dummy.next;
    }
};
```

相关题目

- Add Binary

Reverse Linked List II

描述

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

For example: Given `1->2->3->4->5->nullptr`, `m` = 2 and `n` = 4,

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

Note: Given `m`, `n` satisfy the following condition: $1 \leq m \leq n \leq$ length of list.

分析

这题非常繁琐，有很多边界检查，15分钟内做到bug free很有难度！

代码

```
// Reverse Linked List II
// 迭代版，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;

        ListNode prev = dummy;
        for (int i = 0; i < m-1; ++i)
            prev = prev.next;
        ListNode head2 = prev;

        prev = head2.next;
        ListNode cur = prev.next;
        for (int i = m; i < n; ++i) {
            prev.next = cur.next;
            cur.next = head2.next;
            head2.next = cur;  // 头插法
            cur = prev.next;
        }

        return dummy.next;
    }
};
```

Partition List

描述

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

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

For example, Given `1->4->3->2->5->2` and `x = 3`, return `1->2->2->4->3->5`.

分析

无

代码

```
// Partition List
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode partition(ListNode head, int x) {
        ListNode left_dummy = new ListNode(-1); // 头结点
        ListNode right_dummy = new ListNode(-1); // 头结点

        ListNode left_cur = left_dummy;
        ListNode right_cur = right_dummy;

        for (ListNode cur = head; cur != null; cur = cur.next) {
            if (cur.val < x) {
                left_cur.next = cur;
                left_cur = cur;
            } else {
                right_cur.next = cur;
                right_cur = cur;
            }
        }

        left_cur.next = right_dummy.next;
        right_cur.next = null;

        return left_dummy.next;
    }
};
```

Remove Duplicates from Sorted List

描述

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

For example,

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

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

分析

无

递归版

```
// Remove Duplicates from Sorted List
// 递归版，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return head;
        ListNode dummy = new ListNode(head.val + 1); // 值只要跟head不同即可
        dummy.next = head;

        recur(dummy, head);
        return dummy.next;
    }
    private static void recur(ListNode prev, ListNode cur) {
        if (cur == null) return;

        if (prev.val == cur.val) { // 删除head
            prev.next = cur.next;
            recur(prev, prev.next);
        } else {
            recur(prev.next, cur.next);
        }
    }
};
```

迭代版

```
// Remove Duplicates from Sorted List
// 迭代版，时间复杂度O(n)，空间复杂度O(1)
class Solution {
public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null;

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

相关题目

- Remove Duplicates from Sorted List II

Remove Duplicates from Sorted List II

描述

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

For example,

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

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

分析

无

递归版

```
// Remove Duplicates from Sorted List II
// 递归版，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head ==null || head.next == null) return head;

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

迭代版

```java
// Remove Duplicates from Sorted List II
// 迭代版，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return head;

        ListNode dummy = new ListNode(Integer.MAX_VALUE); // 头结点
        dummy.next = head;
        ListNode prev = dummy, cur = head;
        while (cur != null) {
            boolean duplicated = false;
            while (cur.next != null && cur.val == cur.next.val) {
                duplicated = true;
                cur = cur.next;
            }
            if (duplicated) { // 删除重复的最后一个元素
                cur = cur.next;
                continue;
            }
            prev.next = cur;
            prev = prev.next;
            cur = cur.next;
        }
        prev.next = cur;
        return dummy.next;
    }
}
```

相关题目

- Remove Duplicates from Sorted List

Rotate List

描述

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

For example: Given `1->2->3->4->5->nullptr` and `k = 2`, return `4->5->1->2->3->nullptr`.

分析

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

代码

```
// Remove Rotate List
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || k == 0) return head;

        int len = 1;
        ListNode p = head;
        while (p.next != null) { // 求长度
            len++;
            p = p.next;
        }
        k = len - k % len;

        p.next = head; // 首尾相连
        for(int step = 0; step < k; step++) {
            p = p.next;  //接着往后跑
        }
        head = p.next; // 新的首节点
        p.next = null; // 断开环
        return head;
    }
};
```

Remove Nth Node From End of List

描述

Given a linked list, remove the `n` -th node from the end of list and return its head.

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

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

Note:

- Given `n` will always be valid.
- Try to do this in one pass.

分析

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

代码

```
// Remove Nth Node From End of List
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode p = dummy, q = dummy;

        for (int i = 0; i < n; i++)  // q先走n步
            q = q.next;

        while(q.next != null) { // 一起走
            p = p.next;
            q = q.next;
        }
        p.next = p.next.next;
        return dummy.next;
    }
}
```

Swap Nodes in Pairs

描述

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

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

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

分析

无

代码

```
// Swap Nodes in Pairs
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode dummy = new ListNode(-1);
        dummy.next = head;

        for(ListNode prev = dummy, cur = prev.next, next = cur.next;
                next != null;
                prev = cur, cur = cur.next, next = cur != null ? cur.next: null) {
            prev.next = next;
            cur.next = next.next;
            next.next = cur;
        }
        return dummy.next;
    }
}
```

下面这种写法更简洁，但题目规定了不准这样做。

```
// Swap Nodes in Pairs
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode p = head;

        while (p != null && p.next != null) {
            int tmp = p.val;
            p.val = p.next.val;
            p.next.val = tmp;

            p = p.next.next;
        }

        return head;
    }
}
```

相关题目

- Reverse Nodes in k-Group

Reverse Nodes in k-Group

描述

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of `k` then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example, Given this linked list: `1->2->3->4->5`

For `k = 2`, you should return: `2->1->4->3->5`

For `k = 3`, you should return: `3->2->1->4->5`

分析

无

递归版

```java
// Reverse Nodes in k-Group
// 递归版，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null || head.next == null || k < 2)
            return head;

        ListNode next_group = head;
        for (int i = 0; i < k; ++i) {
            if (next_group != null)
                next_group = next_group.next;
            else
                return head;
        }
        // next_group is the head of next group
        // new_next_group is the new head of next group after reversion
        ListNode new_next_group = reverseKGroup(next_group, k);
        ListNode prev = null, cur = head;
        while (cur != next_group) {
            ListNode next = cur.next;
            cur.next = prev != null ? prev : new_next_group;
            prev = cur;
            cur = next;
        }
        return prev; // prev will be the new head of this group
    }
}
```

迭代版

```java
// Reverse Nodes in k-Group
// 迭代版，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null || head.next == null || k < 2) return head;
        ListNode dummy = new ListNode(-1);
        dummy.next = head;

        for(ListNode prev = dummy, end = head; end != null; end = prev.next) {
            for (int i = 1; i < k && end != null; i++)
                end = end.next;
            if (end  == null) break;  // 不足 k 个

            prev = reverse(prev, prev.next, end);
        }

        return dummy.next;
    }

    // prev 是 first 前一个元素, [begin, end] 闭区间，保证三者都不为 null
    // 返回反转后的倒数第1个元素
    ListNode reverse(ListNode prev, ListNode begin, ListNode end) {
        ListNode end_next = end.next;
        for (ListNode p = begin, cur = p.next, next = cur.next;
                cur != end_next;
                p = cur, cur = next, next = next != null ? next.next : null) {
            cur.next = p;
        }
        begin.next = end_next;
        prev.next = end;
        return begin;
    }
};
```

相关题目

- Swap Nodes in Pairs

Copy List with Random Pointer

描述

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

Return a deep copy of the list.

分析

无

代码

```
// Copy List with Random Pointer
// 两遍扫描，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public RandomListNode copyRandomList(RandomListNode head) {
        for (RandomListNode cur = head; cur != null; ) {
            RandomListNode node = new RandomListNode(cur.label);
            node.next = cur.next;
            cur.next = node;
            cur = node.next;
        }

        for (RandomListNode cur = head; cur != null; ) {
            if (cur.random != null)
                cur.next.random = cur.random.next;
            cur = cur.next.next;
        }

        // 分拆两个单链表
        RandomListNode dummy = new RandomListNode(-1);
        for (RandomListNode cur = head, new_cur = dummy;
                cur != null; ) {
            new_cur.next = cur.next;
            new_cur = new_cur.next;
            cur.next = cur.next.next;
            cur = cur.next;
        }
        return dummy.next;
    }
};
```

Linked List Cycle

描述

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

Follow up: Can you solve it without using extra space?

分析

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

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

代码

```
// Linked List Cycle
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public boolean hasCycle(ListNode head) {
        // 设置两个指针，一个快一个慢
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) return true;
        }
        return false;
    }
};
```

相关题目

- Linked List Cycle II

Linked List Cycle II

描述

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

Follow up: Can you solve it without using extra space?

分析

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

`2s = s + nr`

`s = nr`

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

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

`x = (n-1)r + (L – x – a)`

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

代码

```
// Linked List Cycle II
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                ListNode slow2 = head;

                while (slow2 != slow) {
                    slow2 = slow2.next;
                    slow = slow.next;
                }
                return slow2;
            }
        }
        return null;
    }
}
```

相关题目

- Linked List Cycle

Reorder List

描述

Given a singly linked list $L: L_0 \rightarrow L_1 \rightarrow \cdots \rightarrow L_{n-1} \rightarrow L_n$, reorder it to:

$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \cdots$

You must do this in-place without altering the nodes' values.

For example, Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

分析

题目规定要in-place，也就是说只能使用`O(1)`的空间。

可以找到中间节点，断开，把后半截单链表reverse一下，再合并两个单链表。

代码

```java
// Reorder List
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public void reorderList(ListNode head) {
        if (head == null || head.next == null) return;

        ListNode slow = head, fast = head, prev = null;
        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        prev.next = null; // cut at middle

        slow = reverse(slow);

        // merge two lists
        ListNode curr = head;
        while (curr.next != null) {
            ListNode tmp = curr.next;
            curr.next = slow;
            slow = slow.next;
            curr.next.next = tmp;
            curr = tmp;
        }
        curr.next = slow;
    }

    ListNode reverse(ListNode head) {
        if (head == null || head.next == null) return head;

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

LRU Cache

描述

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.

分析

为了使查找、插入和删除都有较高的性能，这题的关键是要使用一个双向链表和一个HashMap，因为：

- HashMap保存每个节点的地址，可以基本保证在`O(1)`时间内查找节点
- 双向链表能后在`O(1)`时间内添加和删除节点，单链表则不行

具体实现细节：

- 越靠近链表头部，表示节点上次访问距离现在时间最短，尾部的节点表示最近访问最少
- 访问节点时，如果节点存在，把该节点交换到链表头部，同时更新hash表中该节点的地址
- 插入节点时，如果cache的size达到了上限capacity，则删除尾部节点，同时要在hash表中删除对应的项；新节点插入链表头部

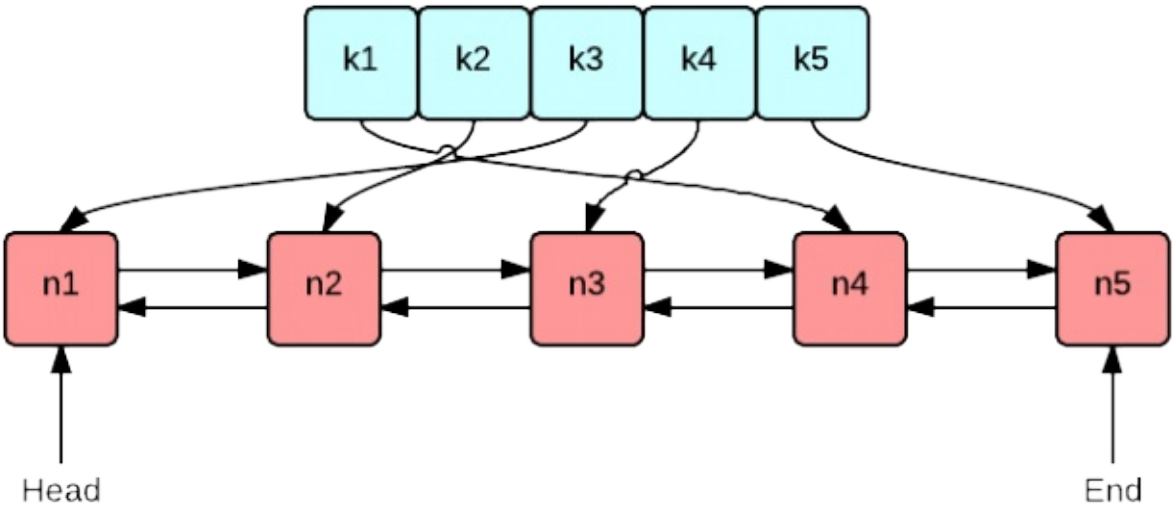


Figure: LRU Cche

代码

Java中也有双向链表 `LinkedList`，但是 `LinkedList` 封装的太深，没有能在 `O(1)` 时间内删除中间某个元素的API(C++的 `list` 有个 `splice()`，O(1)，所以本题C++可以放心使用 `splice()`)，于是我们只能自己实现一个双向链表。

本题有的人直接用 `LinkedHashMap` ，代码更短，但这是一种偷懒做法，面试官一定会让你自己重新实现。

```
// LRU Cache
// 时间复杂度O(logn)，空间复杂度O(n)
public class LRUCache {
    private int capacity;
    private final HashMap<Integer, Node> map;
    private Node head;
    private Node end;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
    }

    public int get(int key) {
        if(map.containsKey(key)){
            Node n = map.get(key);
            remove(n);
            setHead(n);
            return n.value;
        }

        return -1;
    }

    public void set(int key, int value) {
        if (map.containsKey(key)){
            Node old = map.get(key);
            old.value = value;
            remove(old);
            setHead(old);
        } else {
            Node created = new Node(key, value);
            if (map.size() >= capacity){
                map.remove(end.key);
                remove(end);
                setHead(created);
            } else {
                setHead(created);
            }

            map.put(key, created);
        }
    }

    private void remove(Node n){
```

```
        if (n.prev !=null) {
            n.prev.next = n.next;
        } else {
            head = n.next;
        }

        if (n.next != null) {
            n.next.prev = n.prev;
        } else {
            end = n.prev;
        }

    }

    private void setHead(Node n){
        n.next = head;
        n.prev = null;

        if (head!=null ) head.prev = n;

        head = n;

        if(end == null) end = head;
    }

    // doubly linked list
    static class Node {
        int key;
        int value;
        Node prev;
        Node next;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

Palindrome Linked List

描述

Given a singly linked list, determine if it is a palindrome.

Follow up:

Could you do it in `O(n)` time and `O(1)` space?

分析

首先要寻找中点，原理是使用快慢指针，每次快指针走两步，慢指针走一步。同时还要用栈，每次慢指针走一步，都把值存入栈中。等快指针走完时，链表的前半段都存入栈中了。最后慢指针继续往前走，每次与栈顶元素进行比较。空间复杂度 `O(n)` 。

如何做到用 `O(1)` 空间呢？可以先找到中点，把后半段reverse一下，然后比较两个小链表。

代码

```java
// Palindrome Linked List
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null) return true;

        final ListNode middle = findMiddle(head);
        middle.next = reverse(middle.next);

        ListNode p1 = head;
        ListNode p2 = middle.next;
        while (p1 != null && p2 != null && p1.val == p2.val) {
            p1 = p1.next;
            p2 = p2.next;
        }
        return p2 == null;

    }
    private static ListNode findMiddle(ListNode head) {
        if (head == null) return null;

        ListNode slow = head;
        ListNode fast = head.next;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    private static ListNode reverse(ListNode head) {
        ListNode prev = null;

        while (head != null) {
            ListNode tmp = head.next;
            head.next = prev;
            prev = head;
            head = tmp;
        }

        return prev;
    }
}
```

字符串

本章主要讲字符串相关的算法。

Valid Palindrome

描述

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

For example,

`"A man, a plan, a canal: Panama"` is a palindrome. `"race a car"` is not a palindrome.

Note: Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

分析

无

代码

```
// Valid Palindrome
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public boolean isPalindrome(String s) {
        s = s.toLowerCase();
        int left = 0;
        int right = s.length() - 1;
        while (left < right) {
            if (!Character.isLetterOrDigit(s.charAt(left))) ++left;
            else if (!Character.isLetterOrDigit(s.charAt(right))) --right;
            else if (s.charAt(left) != s.charAt(right)) return false;
            else { ++left; --right; }
        }
        return true;
    }
}
```

相关题目

- Palindrome Number

Implement strStr()

描述

Implement strStr().

Returns a pointer to the first occurrence of needle in haystack, or null if needle is not part of haystack.

分析

暴力算法的复杂度是 `O(m*n)`，代码如下。更高效的的算法有KMP算法、Boyer-Mooer算法和Rabin-Karp算法。面试中暴力算法足够了，一定要写得没有BUG。

暴力匹配

```
// Implement strStr()
// 暴力解法，时间复杂度O(N*M)，空间复杂度O(1)
class Solution {
    public int strStr(final String haystack, final String needle) {
        if (needle.isEmpty()) return 0;

        final int N = haystack.length() - needle.length() + 1;
        for (int i = 0; i < N; i++) {
            int j = i;
            int k = 0;
            while (j < haystack.length() && k < needle.length() &&
                    haystack.charAt(j) == needle.charAt(k)) {
                j++;
                k++;
            }
            if (k == needle.length()) return i;
        }
        return -1;
    }
}
```

KMP

```
// Implement strStr()
// KMP，时间复杂度O(N+M)，空间复杂度O(M)
public class Solution {
    public int strStr(final String haystack, final String needle) {
        return kmp(haystack, needle);
    }

    /*
     * 计算部分匹配表，即next数组.
     *
```

```
     * @param[in] pattern 模式串
     * @param[out] next next数组
     * @return 无
     */
    private static void compute_prefix(final String pattern, final int[] next) {
        int i;
        int j = -1;

        next[0] = j;
        for (i = 1; i < pattern.length(); i++) {
            while (j > -1 && pattern.charAt(j + 1) != pattern.charAt(i)) j = next[j];

            if (pattern.charAt(i) == pattern.charAt(j + 1)) j++;
            next[i] = j;
        }
    }

    /*
     * @brief KMP算法.
     *
     * @param[in] text 文本
     * @param[in] pattern 模式串
     * @return 成功则返回第一次匹配的位置，失败则返回-1
     */
    private static int kmp(final String text, final String pattern) {
        int i;
        int j = -1;
        final int n = text.length();
        final int m = pattern.length();
        if (n == 0 && m == 0) return 0; /* "","" */
        if (m == 0) return 0;  /* "a","" */
        int[] next = new int[m];

        compute_prefix(pattern, next);

        for (i = 0; i < n; i++) {
            while (j > -1 && pattern.charAt(j + 1) != text.charAt(i)) j = next[j];

            if (text.charAt(i) == pattern.charAt(j + 1)) j++;
            if (j == m - 1) {
                return i-j;
            }
        }

        return -1;
    }
}
```

相关题目

- String to Integer (atoi)

String to Integer (atoi)

描述

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

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

Requirements for atoi:

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, `INT_MAX (2147483647)` or `INT_MIN (-2147483648)` is returned.

分析

细节题。注意几个测试用例：

1. 不规则输入，但是有效，"-3924x8fc"， " + 413",
2. 无效格式，" ++c", " ++1"
3. 溢出数据，"2147483648"

代码

```java
// String to Integer (atoi)
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public int myAtoi(final String str) {
        int num = 0;
        int sign = 1;
        final int n = str.length();
        if (n == 0) return 0;

        int i = 0;
        while (i < n && str.charAt(i) == ' ') i++;

        if (str.charAt(i) == '+') {
            i++;
        } else if (str.charAt(i) == '-') {
            sign = -1;
            i++;
        }

        for (; i < n; i++) {
            if (str.charAt(i) < '0' || str.charAt(i) > '9')
                break;
            if (num > Integer.MAX_VALUE / 10 ||
                    (num == Integer.MAX_VALUE / 10 &&
                            (str.charAt(i) - '0') > Integer.MAX_VALUE % 10)) {
                return sign == -1 ? Integer.MIN_VALUE : Integer.MAX_VALUE;
            }
            num = num * 10 + str.charAt(i) - '0';
        }
        return num * sign;
    }
}
```

相关题目

- Implement strStr()

Add Binary

描述

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

For example,

```
a = "11"
b = "1"
```

Return `100`.

分析

无

代码

```java
// Add Binary
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public String addBinary(String a, String b) {
        StringBuilder result = new StringBuilder();
        int i = a.length() - 1;
        int j = b.length() - 1;
        int carry = 0;

        while(i >= 0 || j >= 0 || carry > 0) {
            int valueA = i < 0 ? 0 : a.charAt(i--) - '0';
            int valueB = j < 0 ? 0 : b.charAt(j--) - '0';
            int sum = valueA + valueB + carry;
            result.insert(0, Character.forDigit(sum % 2, 10));
            carry = sum / 2;
        }
        return result.toString();
    }
}
```

相关题目

- Add Two Numbers

Longest Palindromic Substring

描述

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

分析

最长回文子串，非常经典的题。

思路一：暴力枚举，以每个元素为中间元素，同时从左右出发，复杂度 `O(n^2)`。

思路二：记忆化搜索，复杂度 `O(n^2)`。设 `f[i][j]` 表示[i,j]之间的最长回文子串，递推方程如下：

```
f[i][j] = if (i == j) S[i]
          if (S[i] == S[j] && f[i+1][j-1] == S[i+1][j-1]) S[i][j]
          else max(f[i+1][j-1], f[i][j-1], f[i+1][j])
```

思路三：动规，复杂度 `O(n^2)`。设状态为 `f(i,j)`，表示区间[i,j]是否为回文串，则状态转移方程为

$$f(i,j) = \begin{cases} true & ,i = j \\ S[i] = S[j] & ,j = i+1 \\ S[i] = S[j] \text{ and } f(i+1,j-1) & ,j > i+1 \end{cases}$$

思路四：Manacher's Algorithm, 复杂度 `O(n)`。详细解释见 http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html。

备忘录法

```
// Longest Palindromic Substring
// 备忘录法，会超时
// 时间复杂度O(n^2)，空间复杂度O(n^2)
public class Solution {
    private final HashMap<Pair, String> cache = new HashMap<>();

    public String longestPalindrome(final String s) {
        cache.clear();
        return cachedLongestPalindrome(s, 0, s.length() - 1);
    }

    String longestPalindrome(final String s, int i, int j) {
        final int length = j - i + 1;
        if (length < 2) return s.substring(i, j + 1);

        final String s1 = cachedLongestPalindrome(s, i + 1, j - 1);

        if (s1.length() == length - 2 && s.charAt(i + 1) == s.charAt(j - 1))
            return s.substring(i, j + 1);
```

```
        final String s2 = cachedLongestPalindrome(s, i + 1, j);
        final String s3 = cachedLongestPalindrome(s, i, j - 1);

        // return max(s1, s2, s3)
        if (s1.length() > s2.length()) return s1.length() > s3.length() ? s1 : s3;
        else return s2.length() > s3.length() ? s2 : s3;
    }

    String cachedLongestPalindrome(final String s, int i, int j) {
        final Pair key = new Pair(i, j);

        if (cache.containsKey(key)) {
            return cache.get(key);
        } else {
            final String result = longestPalindrome(s, i, j);
            cache.put(key, result);
            return result;
        }
    }

    // immutable
    static class Pair {
        private int x;
        private int y;

        public Pair(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public int hashCode() {
            return x * 31 + y;
        }

        @Override
        public boolean equals(Object other) {
            if (this == other) return true;
            if (this.hashCode() != other.hashCode()) return false;
            if (!(other instanceof Pair)) return false;

            final Pair o = (Pair) other;
            return this.x == o.x && this.y == o.y;
        }
    }
}
```

动规

```java
// Longest Palindromic Substring
// 动规，时间复杂度O(n^2)，空间复杂度O(n^2)
class Solution {
    public String longestPalindrome(final String s) {
        final int n = s.length();
        final boolean[][] f = new boolean[n][n];
        int maxLen = 1, start = 0;  // 最长回文子串的长度，起点

        for (int i = 0; i < n; i++) {
            f[i][i] = true;
            for (int j = 0; j < i; j++) {  // [j, i]
                f[j][i] = (s.charAt(j) == s.charAt(i) &&
                        (i - j < 2 || f[j + 1][i - 1]));
                if (f[j][i] && maxLen < (i - j + 1)) {
                    maxLen = i - j + 1;
                    start = j;
                }
            }
        }
        return s.substring(start, start + maxLen);
    }
}
```

Manacher’s Algorithm

```java
// Longest Palindromic Substring
// Manacher’s Algorithm
// 时间复杂度O(n)，空间复杂度O(n)
class Solution {
    // Transform S into T.
    // For example, S = "abba", T = "^#a#b#b#a#$".
    // ^ and $ signs are sentinels appended to each end to avoid bounds checking
    public String preProcess(final String s) {
        int n = s.length();
        if (n == 0) return "^$";

        StringBuilder ret = new StringBuilder("^");
        for (int i = 0; i < n; i++) ret.append("#" + s.charAt(i));

        ret.append("#$");
        return ret.toString();
    }

    String longestPalindrome(String s) {
        String T = preProcess(s);
        final int n = T.length();
        // 以T[i]为中心，向左/右扩张的长度，不包含T[i]自己，
        // 因此 P[i]是源字符串中回文串的长度
        int[] P = new int[n];
        int C = 0, R = 0;
```

```
        for (int i = 1; i < n - 1; i++) {
            int iMirror = 2 * C - i; // equals to i' = C - (i-C)

            P[i] = (R > i) ? Math.min(R - i, P[iMirror]) : 0;

            // Attempt to expand palindrome centered at i
            while (T.charAt(i + 1 + P[i]) == T.charAt(i - 1 - P[i]))
                P[i]++;

            // If palindrome centered at i expand past R,
            // adjust center based on expanded palindrome.
            if (i + P[i] > R) {
                C = i;
                R = i + P[i];
            }
        }

        // Find the maximum element in P.
        int maxLen = 0;
        int centerIndex = 0;
        for (int i = 1; i < n - 1; i++) {
            if (P[i] > maxLen) {
                maxLen = P[i];
                centerIndex = i;
            }
        }

        final int start =(centerIndex - 1 - maxLen) / 2;
        return s.substring(start, start + maxLen);
    }
}
```

Regular Expression Matching

描述

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
```

分析

这是一道很有挑战的题。

递归版

```java
// Regular Expression Matching
// Time complexity: O(n)
// Space complexity: O(1)
class Solution {
    public boolean isMatch(final String s, final String p) {
        return isMatch(s, 0, p, 0);
    }
    private static boolean matchFirst(String s, int i, String p, int j) {
        if (j == p.length()) return i == s.length();
        if (i == s.length()) return j == p.length();
        return p.charAt(j) == '.' || s.charAt(i) == p.charAt(j);
    }
    private static boolean isMatch(String s, int i, String p, int j) {
        if (j == p.length()) return i == s.length();

        // next char is not '*', then must match current character
        final char b = p.charAt(j);
        if (j == p.length() - 1 || p.charAt(j + 1) != '*') {
            if (matchFirst(s, i, p, j)) return isMatch(s, i + 1, p, j + 1);
            else return false;
        } else { // next char is '*'
            if (isMatch(s, i, p, j+2)) return true;  // try the length of 0
            while (matchFirst(s, i, p, j))  // try all possible lengths
                if (isMatch(s, ++i, p, j+2)) return true;
            return false;
        }
    }
}
```

相关题目

- Wildcard Matching

Wildcard Matching

描述

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
```

分析

跟上一题很类似。

主要是 `'*'` 的匹配问题。 `p` 每遇到一个 `'*'` ，就保留住当前 `'*'` 的坐标和 `s` 的坐标，然后 `s` 从前往后扫描，如果不成功，则 `s++` ，重新扫描。

递归版

```java
// Wildcard Matching
// 递归版，会超时，用于帮助理解题意
// 时间复杂度O(n!*m!)，空间复杂度O(n)
class Solution {
    public boolean isMatch(String s, String p) {
        return isMatch(s, 0, p, 0);
    }
    private boolean isMatch(String s, int i, String p, int j) {
        if (i == s.length() && j == p.length()) return true;
        if (i == s.length() || j == p.length()) return false;

        if (p.charAt(j) == '*') {
            while (j < p.length() && p.charAt(j) == '*') ++j;  //skip continuous '*'
            if (j == p.length()) return true;
            while (i < s.length() && !isMatch(s, i, p, j)) ++i;

            return i < s.length();
        }
        else if (p.charAt(j) == s.charAt(i) || p.charAt(j) == '?')
            return isMatch(s, ++i, p, ++j);
        else return false;
    }
}
```

迭代版

```
// Wildcard Matching
// 迭代版，时间复杂度O(n*m)，空间复杂度O(1)
public class Solution {
    public boolean isMatch(String s, String p) {
        int i = 0, j = 0;
        int ii = -1, jj = -1;
        while (i < s.length()) {
            if (j < p.length() && p.charAt(j) == '*') {
                // skip continuous '*'
                while (j < p.length() && p.charAt(j) == '*') ++j;
                if (j == p.length()) return true;
                ii = i;
                jj = j;
            }
            if (j < p.length() && (p.charAt(j) == '?' || p.charAt(j) == s.charAt(i))) {
                ++i; ++j;
            } else {
                if (ii == -1) return false;
                ++ii;
                i = ii;
                j = jj;
            }
        }
        // skip continuous '*'
        while (j < p.length() && p.charAt(j) == '*') ++j;
        return i == s.length() && j == p.length();
    }
}
```

相关题目

- Regular Expression Matching

Longest Common Prefix

描述

Write a function to find the longest common prefix string amongst an array of strings.

分析

从位置0开始，对每一个位置比较所有字符串，直到遇到一个不匹配。

纵向扫描

```
// Longest Common Prefix
// 纵向扫描，从位置0开始，对每一个位置比较所有字符串，直到遇到一个不匹配
// 时间复杂度O(n1+n2+...)
public class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs.length == 0) return "";

        for (int j = 0; j < strs[0].length(); ++j) { // 纵向扫描
            for (int i = 1; i < strs.length; ++i) {
                if (j == strs[i].length() ||
                        strs[i].charAt(j) != strs[0].charAt(j))
                    return strs[0].substring(0, j);
            }
        }
        return strs[0];
    }
}
```

横向扫描

```java
// Longest Common Prefix
// 横向扫描，每个字符串与第0个字符串，从左到右比较，直到遇到一个不匹配，
// 然后继续下一个字符串
// 时间复杂度O(n1+n2+...)
class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs.length == 0) return "";

        int right_most = strs[0].length();
        for (int i = 1; i < strs.length; i++)
            for (int j = 0; j < right_most; j++)
                // 不会越界，请参考string::[]的文档
                if (j == strs[i].length() ||
                        strs[i].charAt(j) != strs[0].charAt(j))
                    right_most = j;

        return strs[0].substring(0, right_most);
    }
}
```

Valid Number

描述

Validate if a given string is numeric.

Some examples:

```
"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
```

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

分析

细节实现题。

本题的功能与标准库中的 `strtod()` 功能类似。

有限自动机

```java
// Valid Number
// finite automata，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public boolean isNumber(String s) {
        int[][] transitionTable  = new int[][] {
            { -1, 0, 3, 1, 2, -1 }, // next states for state 0
            { -1, 8, -1, 1, 4, 5 },     // next states for state 1
            { -1, -1, -1, 4, -1, -1 },     // next states for state 2
            { -1, -1, -1, 1, 2, -1 },     // next states for state 3
            { -1, 8, -1, 4, -1, 5 },     // next states for state 4
            { -1, -1, 6, 7, -1, -1 },     // next states for state 5
            { -1, -1, -1, 7, -1, -1 },     // next states for state 6
            { -1, 8, -1, 7, -1, -1 },     // next states for state 7
            { -1, 8, -1, -1, -1, -1 }     // next states for state 8
        };

        int state = 0;
        for (int i = 0; i < s.length(); ++i) {
            final char ch = s.charAt(i);
            InputType inputType = InputType.INVALID;

            if (Character.isSpaceChar(ch))
                inputType = InputType.SPACE;
            else if (ch == '+' || ch == '-')
```

```
                inputType = InputType.SIGN;
            else if (Character.isDigit(ch))
                inputType = InputType.DIGIT;
            else if (ch == '.')
                inputType = InputType.DOT;
            else if (ch == 'e' || ch == 'E')
                inputType = InputType.EXPONENT;

            // Get next state from current state and input symbol
            state = transitionTable[state][inputType.ordinal()];

            // Invalid input
            if (state == -1) return false;
        }
        // If the current state belongs to one of the accepting (final) states,
        // then the number is valid
        return state == 1 || state == 4 || state == 7 || state == 8;

    }
    enum InputType {
        INVALID,    // 0
        SPACE,      // 1
        SIGN,       // 2
        DIGIT,      // 3
        DOT,        // 4
        EXPONENT,   // 5
        NUM_INPUTS  // 6
    }
}
```

Integer to Roman

描述

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

分析

无

代码

```
// Integer to Roman
// 时间复杂度O(num)，空间复杂度O(1)
class Solution {
    public String intToRoman(int num) {
        final int radix[] = {1000, 900, 500, 400, 100, 90,
                50, 40, 10, 9, 5, 4, 1};
        final String symbol[] = {"M", "CM", "D", "CD", "C", "XC",
                "L", "XL", "X", "IX", "V", "IV", "I"};

        StringBuilder roman = new StringBuilder();
        for (int i = 0; num > 0; ++i) {
            int count = num / radix[i];
            num %= radix[i];
            for (; count > 0; --count) roman.append(symbol[i]);
        }
        return roman.toString();
    }
}
```

相关题目

- Roman to Integer

Roman to Integer

描述

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

分析

从前往后扫描，用一个临时变量记录分段数字。

如果当前比前一个大，说明这一段的值应该是当前这个值减去上一个值。比如 `IV = 5 - 1`；否则，将当前值加入到结果中，然后开始下一段记录。比如 `VI = 5 + 1, II=1+1`

代码

```
// Roman to Integer
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public int romanToInt(String s) {
        int result = 0;
        for (int i = 0; i < s.length(); i++) {
            if (i > 0 && map(s.charAt(i)) > map(s.charAt(i - 1))) {
                result += (map(s.charAt(i)) - 2 * map(s.charAt(i - 1)));
            } else {
                result += map(s.charAt(i));
            }
        }
        return result;
    }
    private static int map(char c) {
        switch (c) {
            case 'I': return 1;
            case 'V': return 5;
            case 'X': return 10;
            case 'L': return 50;
            case 'C': return 100;
            case 'D': return 500;
            case 'M': return 1000;
            default: return 0;
        }
    }
}
```

相关题目

- Integer to Roman

Count and Say

描述

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

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

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

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

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

Given an integer `n`, generate the nth sequence.

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

分析

模拟。

代码

```java
// Count and Say
// @author 连城 (http://weibo.com/lianchengzju)
// 时间复杂度O(n^2)，空间复杂度O(n)
class Solution {
    public String countAndSay(int n) {
        String s = "1";

        while (--n > 0)
            s = getNext(s);

        return s;
    }

    String getNext(final String s) {
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < s.length();) {
            int j = notEqual(s, i);
            sb.append(j - i);
            sb.append(s.charAt(i));
            i = j;
        }

        return sb.toString();
    }
    // find the first char that not equal to fromIndex
    private static int notEqual(final String s, int fromIndex) {
        final char target = s.charAt(fromIndex);
        int i = fromIndex;
        for (; i < s.length(); ++i) {
            if (s.charAt(i) != target) break;
        }
        return i;
    }
}
```

Anagrams

描述

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

分析

Anagram（回文构词法）是指打乱字母顺序从而得到新的单词，比如 `"dormitory"` 打乱字母顺序会变成 `"dirty room"` ，`"tea"` 会变成 `"eat"` 。

回文构词法有一个特点：单词里的字母的种类和数目没有改变，只是改变了字母的排列顺序。因此，将几个单词按照字母顺序排序后，若它们相等，则它们属于同一组 anagrams 。

代码

```
// Anagrams
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        final HashMap<String, ArrayList<String>> group = new HashMap<>();
        for (final String s : strs) {
            char[] tmp = s.toCharArray();
            Arrays.sort(tmp);

            final String key = new String(tmp);
            if (!group.containsKey(key)) {
                group.put(key, new ArrayList<>());
            }
            group.get(key).add(s);
        }

        List<List<String>> result = new ArrayList<>();
        for (Map.Entry<String, ArrayList<String>> entry : group.entrySet()) {
            final ArrayList<String> list = entry.getValue();
            Collections.sort(list);
            result.add(list);
        }
        return result;
    }
}
```

Valid Anagram

描述

Given two strings `s` and `t`, write a function to determine if `t` is an anagram of `s`.

For example,

s = "anagram", t = "nagaram", return true. s = "rat", t = "car", return false.

Note:

You may assume the string contains only lowercase alphabets.

分析

首先能够想到的是，为 `s` 和 `t` 分别建立一个HashMap，统计每个字母出现的次数，比较两个HashMap是否相等。时间复杂度 `O(n)`，空间复杂度 `O(n)`，`n` 为字符串长度。

如果面试官要求空间复杂度均为 `O(1)`，怎么办？

注意这题的字符串只包含小写的字母，意味着只有从 `a-z` 的26个字母，于是我们可以为 `s` 和 `t` 开一个长度为26的整数数组来代替哈希表，此时额外空间为固定长度的两个数组，因此为 `O(1)`。这是这题比较狡诈的地方。

代码

```
// Valid Anagram
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;
        final int[] map = new int[ALPHABET_SIZE];

        for (int i = 0; i < s.length(); ++i) {
            ++map[s.charAt(i) - 'a'];
            --map[t.charAt(i) - 'a'];
        }
        for (int x : map) {
            if (x != 0) return false;
        }
        return true;
    }
    private static final int ALPHABET_SIZE = 26;
}
```

Simplify Path

描述

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

For example,

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

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 `"/home//foo/"`.

  In this case, you should ignore redundant slashes and return `"/home/foo"`.

分析

很有实际价值的题目。

代码

```java
import java.util.*;

// Simplify Path
// 时间复杂度O(n)，空间复杂度O(n)
class Solution {
    public String simplifyPath(String path) {
        Stack<String> dirs = new Stack<>();

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

            int j = path.indexOf('/', i);
            if (j < 0) j = path.length();
            final String dir = path.substring(i, j);

            // 当有连续 '///'时，dir 为空
            if (!dir.isEmpty() && !dir.equals(".")) {
                if (dir.equals("..")) {
                    if (!dirs.isEmpty())
                        dirs.pop();
                } else {
                    dirs.push(dir);
                }
            }

            i = j;
        }

        StringBuilder result = new StringBuilder();
        if (dirs.isEmpty()) {
            result.append('/');
        } else {
            for (final String dir : dirs) {
                result.append('/').append(dir);
            }
        }
        return result.toString();
    }
}
```

Length of Last Word

描述

Given a string s consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example, Given `s = "Hello World"`, return 5.

分析

模拟。先从右到左找到第一个字母，然后从右到左找到第一个非字母，二者的距离就是最后一个word的长度。

代码

```java
// Length of Last Word
// 偷懒，用 STL
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int lengthOfLastWord(String s) {
        final Predicate isAlphabet = new Predicate() {
            @Override
            public boolean apply(char ch) {
                return Character.isAlphabetic(ch);
            }
        };
        final Predicate isNotAlphabet = new Predicate() {
            @Override
            public boolean apply(char ch) {
                return !Character.isAlphabetic(ch);
            }
        };
        int j = findIf(s, s.length() - 1, isAlphabet);
        int i = findIf(s, j, isNotAlphabet);
        return j - i;
    }

    interface Predicate {
        boolean apply(char ch);
    }
    // from right to left
    private static int findIf(final String s, int fromIndex, Predicate p) {
        for (int i = fromIndex; i >= 0; --i) {
            if (p.apply(s.charAt(i))) return i;
        }
        return -1;
    }
}
```

Isomorphic Strings

描述

Given two strings `s` and `t`, determine if they are isomorphic.

Two strings are isomorphic if the characters in `s` can be replaced to get `t`.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,

Given `"egg"`, `"add"`, return true.

Given `"foo"`, `"bar"`, return false.

Given `"paper"`, `"title"`, return true.

Note:

You may assume both `s` and `t` have the same length.

分析

用两个HashMap维护字符的映射关系，时间复杂度`O(n)`，空间复杂度`O(n)`。

代码

```java
// Isomorphic Strings
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public boolean isIsomorphic(String s, String t) {
        if (s.length() != t.length()) return false;

        final Map<Character,Character> map1 = new HashMap<>();
        final Map<Character,Character> map2 = new HashMap<>();

        for (int  i = 0; i < s.length(); ++i) {
            final char c1 = s.charAt(i);
            final char c2 = t.charAt(i);

            if (map1.containsKey(c1)) {
                if (map1.get(c1) != c2) return false;
            } else {
                map1.put(c1, c2);
            }

            if (map2.containsKey(c2)) {
                if (map2.get(c2) != c1) return false;
            } else {
                map2.put(c2, c1);
            }
        }
        return true;
    }
}
```

相关题目

- Word Pattern

Word Pattern

描述

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in `pattern` and a non-empty word in `str`.

Examples:

1. pattern = `"abba"`, str = `"dog cat cat dog"` should return true.
2. pattern = `"abba"`, str = `"dog cat cat fish"` should return false.
3. pattern = `"aaaa"`, str = `"dog cat cat dog"` should return false.
4. pattern = `"abba"`, str = `"dog dog dog dog"` should return false.

Notes:

You may assume `pattern` contains only lowercase letters, and `str` contains lowercase letters separated by a single space.

分析

本题跟 "Isomorphic Strings" 很类似，用两个HashMap, 记录从字符到字符串和字符串到字符的映射。

代码

```java
// Word Pattern
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public boolean wordPattern(String pattern, String str) {
        String[] words = str.split(" ");
        if (words.length != pattern.length()) return false;

        final Map<Character, String> map1 = new HashMap<>();
        final Map<String, Character> map2 = new HashMap<>();
        for (int i = 0; i < words.length; ++i) {
            final Character key1 = pattern.charAt(i);
            if (map1.containsKey(key1)) {
                final String value = map1.get(key1);
                if (!value.equals(words[i])) return false;
            } else {
                map1.put(key1, words[i]);
            }

            final String key2 = words[i];
            if (map2.containsKey(key2)) {
                final char value = map2.get(key2);
                if (value != pattern.charAt(i)) return false;
            } else {
                map2.put(key2, pattern.charAt(i));
            }
        }
        return true;
    }
}
```

相关题目

- Isomorphic Strings

本章主要讲栈和队列相关的算法。

本节主要讲栈相关的算法。

Min Stack

描述

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.

分析

用两个栈，一个是真实的栈，另一个作为辅助栈，辅助栈每次 push 时，会把新元素跟当前栈顶元素进行比较，存入二者中较小的那个。

举个例子，对于序列 `18, 19, 21, 15, 17`，两个栈依次 `push` 进去的元素是这样的：

- 真实栈，`18, 19, 21, 15, 17`
- 辅助栈，`18, 18, 18, 15, 15`

  具体过程是这样的，对于 `18`，辅助栈是空的，直接push进去，当遇到 `19` 时，此时栈顶元素是 `18`，二者中 `18` 较小，就把18插入，此时辅助栈中就有了两个 `18`，当遇到 `21` 时，以此类推，还是插入 `18`，遇到 `15` 时，栈顶元素是 `18`，`15` 较小，就把15压入，此时辅助栈中有3个 `18`，1个 `15`，当遇到 `17` 时，栈顶元素是 `15`，二者中 `15` 是较小值，于是插入 `15`，结束。

代码

```java
// Min Stack
// Time Complexity: O(n), Space Complexity: O(1)
class MinStack {
    public void push(int x) {
        s.push(x);
        int minValue = minStack.isEmpty() ? x :
                Math.min(minStack.peek(), x);
        minStack.push(minValue);
    }

    public void pop() {
        s.pop();
        minStack.pop();
    }

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

    public int getMin() {
        return minStack.peek();
    }

    private Stack<Integer> s = new Stack<>();
    private Stack<Integer> minStack = new Stack<>();
}
```

Valid Parentheses

描述

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.

分析

无

代码

```
// Valid Parentheses
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public boolean isValid(String s) {
        final String left = "([{";
        final String right = ")]}";
        Stack<Character> stk = new Stack<>();

        for (int i = 0; i < s.length(); ++i) {
            final char c = s.charAt(i);
            if (left.indexOf(c) != -1) {
                stk.push (c);
            } else {
                if (!stk.isEmpty() &&
                        stk.peek() == left.charAt(right.indexOf(c)))
                    stk.pop ();
                else
                    return false;
            }
        }
        return stk.empty();
    }
}
```

相关题目

- Generate Parentheses
- Longest Valid Parentheses

Longest Valid Parentheses

描述

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.

分析

无

使用栈

```
// Longest Valid Parenthese
// 使用栈，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int longestValidParentheses(String s) {
        // the position of the last ')'
        int maxLen = 0, last = -1;
        // keep track of the positions of non-matching '('s
        Stack<Integer> lefts = new Stack<>();

        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) =='(') {
                lefts.push(i);
            } else {
                if (lefts.empty()) {
                    // no matching left
                    last = i;
                } else {
                    // find a matching pair
                    lefts.pop();
                    if (lefts.empty()) {
                        maxLen = Math.max(maxLen, i-last);
                    } else {
                        maxLen = Math.max(maxLen, i-lefts.peek());
                    }
                }
            }
        }
        return maxLen;
    }
}
```

Dynamic Programming, One Pass

```
// Longest Valid Parenthese
// 两遍扫描，时间复杂度O(n)，空间复杂度O(1)
// @author 曹鹏(http://weibo.com/cpcs)
class Solution {
    public int longestValidParentheses(final String s) {
        int result = 0, depth = 0, start = -1;
        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) == '(') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    result = Math.max(result, i - start);
                }
            }
        }

        depth = 0;
        start = s.length();
        for (int i = s.length() - 1; i >= 0; --i) {
            if (s.charAt(i) == ')') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    result = Math.max(result, start - i);
                }
            }
        }
        return result;
    }
}
```

两遍扫描

```
// Longest Valid Parenthese
// 两遍扫描，时间复杂度O(n)，空间复杂度O(1)
// @author 曹鹏(http://weibo.com/cpcs)
class Solution {
    public int longestValidParentheses(final String s) {
        int result = 0, depth = 0, start = -1;
        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) == '(') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    result = Math.max(result, i - start);
                }
            }
        }

        depth = 0;
        start = s.length();
        for (int i = s.length() - 1; i >= 0; --i) {
            if (s.charAt(i) == ')') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    result = Math.max(result, start - i);
                }
            }
        }
        return result;
    }
}
```

相关题目

- Valid Parentheses
- Generate Parentheses

Largest Rectangle in Histogram

描述

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.

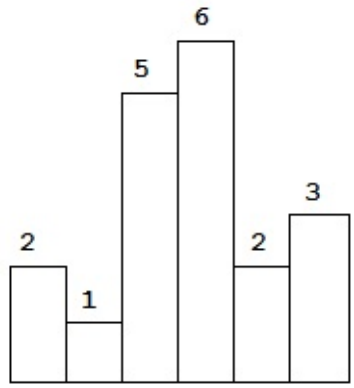


Figure: Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.

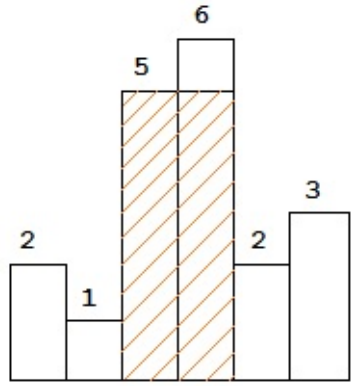


Figure: The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example, given height = `[2,1,5,6,2,3]`, return 10.

分析

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。

如上图所示，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。如果当前元素大于栈顶元素，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。

代码

```
// Largest Rectangle in Histogram
// 时间复杂度O(n)，空间复杂度O(n)
class Solution {
    public int largestRectangleArea(int[] heights) {
        Stack<Integer> s = new Stack<>();
        int result = 0;
        for (int i = 0; i <= heights.length; ) {
            final int value = i < heights.length ? heights[i] : 0;
            if (s.isEmpty() || value > heights[s.peek()])
                s.push(i++);
            else {
                int tmp = s.pop();
                result = Math.max(result,
                        heights[tmp] * (s.isEmpty() ? i : i - s.peek() - 1));
            }
        }
        return result;
    }
}
```

相关题目

- Trapping Rain Water
- Container With Most Water

Evaluate Reverse Polish Notation

描述

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
```

分析

逆波兰表达式是典型的递归结构，所以可以用递归来求解，也可以用栈来求解。

递归版

递归的写法，C++版可以AC，但Java版会爆栈`StackOverflowError`，所以Java只能用栈来解决。

```java
// Evaluate Reverse Polish Notation
// 递归，时间复杂度O(n)，空间复杂度O(logn)
// StackOverflowError
class Solution {
    public int evalRPN(final String[] tokens) {
        return evalRPN(tokens, tokens.length - 1);
    }
    private static int evalRPN(final String[] tokens, int i) {
        if (i < 0) return 0;
        int x, y;
        final String token = tokens[i--];
        if (isOperator(token))  {
            y = evalRPN(tokens, i--);
            x = evalRPN(tokens, i--);

            switch (token.charAt(0)) {
                case '+': x += y; break;
                case '-': x -= y; break;
                case '*': x *= y; break;
                default: x /= y;
            }
        } else  {
            x = Integer.parseInt(token);
        }
        return x;
    }
    private static boolean isOperator(final String op) {
        return op.length() == 1 && OPS.indexOf(op) != -1;
    }
    private static String OPS = new String("+-*/");
}
```

迭代版

```java
// Max Points on a Line
// 迭代，时间复杂度O(n)，空间复杂度O(logn)
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<String> s = new Stack<>();
        for (String token : tokens) {
            if (!isOperator(token)) {
                s.push(token);
            } else {
                int y = Integer.parseInt(s.pop());
                int x = Integer.parseInt(s.pop());
                switch (token.charAt(0)) {
                    case '+': x += y; break;
                    case '-': x -= y; break;
                    case '*': x *= y; break;
                    default: x /= y;
                }
                s.push(String.valueOf(x));
            }
        }
        return Integer.parseInt(s.peek());
    }
    private static boolean isOperator(final String op) {
        return op.length() == 1 && OPS.indexOf(op) != -1;
    }
    private static String OPS = new String("+-*/");
}
```

Implement Stack using Queues

描述

Implement the following operations of a stack using queues.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- empty() -- Return whether the stack is empty.

Notes:

- You must use only standard operations of a queue -- which means only push to back, peek/pop from front, size, and is empty operations are valid.
- Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
- You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

分析

可以用两个队列，`q` 和 `tmp`，`q` 存放元素，`tmp` 用来作中转。

- `push(x)`，先将 `x` push 到 `tmp`，然后把 `q` 中的元素全部弹出来，存入 `tmp`，最后切换 `q` 和 `tmp`
- `pop()`，直接将 `q` 的队首元素弹出来即可

该算法 `push` 的算法复杂度是 `O(n)`，`pop` 的算法复杂度 `O(1)`。

另个一个方法是，让 `pop` 是 `O(n)`，`push` 是 `O(1)`，思路很类似，就不赘述了。

代码

```java
// Implement Stack using Queues
class MyStack {
    // Push element x onto stack.
    // Time Complexity O(n)
    public void push(int x) {
        tmp.offer(x);
        while (!q.isEmpty()) {
            final int e = q.poll();
            tmp.offer(e);
        }
        // swap q and tmp
        Queue temp = tmp;
        tmp = q;
        q = temp;
    }

    // Removes the element on top of the stack.
    // Time Complexity O(1)
    public void pop() {
        q.poll();
    }

    // Get the top element.
    public int top() {
        return q.peek();
    }

    // Return whether the stack is empty.
    public boolean empty() {
        return q.isEmpty();
    }

    private Queue q = new LinkedList<>();
    private Queue tmp = new LinkedList<>();
}
```

相关题目

- Implement Queue using Stacks

本节主要讲队列相关的算法。

Implement Queue using Stacks

描述

Implement the following operations of a queue using stacks.

- push(x) -- Push element x to the back of queue.
- pop() -- Removes the element from in front of queue.
- peek() -- Get the front element.
- empty() -- Return whether the queue is empty.

Notes:

- You must use only standard operations of a stack -- which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
- You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

分析

可以用两个栈，`s` 和 `tmp`，`s` 存放元素，`tmp` 用来作中转。

- `push(x)`，先将 `s` 中的元素全部弹出来，存入 `tmp`，把 `x` push 到 `tmp`，然后把 `tmp` 中的元素全部弹出来，存入 `s`
- `pop()`，直接将 `s` 的栈顶元素弹出来即可

该算法 `push` 的算法复杂度是 `O(n)`，`pop` 的算法复杂度 `O(1)`。

另个一个方法是，让 `pop` 是 `O(n)`，`push` 是 `O(1)`，思路很类似，就不赘述了。

代码

```java
// Implement Queue using Stacks
class MyQueue {
    // Push element x to the back of queue.
    // Time Complexity: O(n)
    public void push(int x) {
        while (!s.isEmpty()) {
            final int e = s.pop();
            tmp.push(e);
        }
        tmp.push(x);

        while(!tmp.isEmpty()) {
            final int e = tmp.pop();
            s.push(e);
        }
    }

    // Removes the element from in front of queue.
    // Time Complexity: O(1)
    public void pop() {
        s.pop();
    }

    // Get the front element.
    public int peek() {
        return s.peek();
    }

    // Return whether the queue is empty.
    public boolean empty() {
        return s.isEmpty();
    }

    private final Stack s = new Stack<>();
    private final Stack tmp = new Stack<>();
}
```

相关题目

- Implement Stack using Queues

本章主要讲树相关的算法。

LeetCode 上二叉树的节点定义如下：

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}
```

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)。

Binary Tree Preorder Traversal

描述

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?

分析

用栈或者Morris遍历。

栈

```
// Binary Tree Preorder Traversal
// 使用栈，时间复杂度O(n)，空间复杂度O(n)
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<>();
        Stack<TreeNode> s = new Stack<>();
        if (root != null) s.push(root);

        while (!s.isEmpty()) {
            final TreeNode p = s.pop();
            result.add(p.val);

            if (p.right != null) s.push(p.right);
            if (p.left != null) s.push(p.left);
        }
        return result;
    }
}
```

Morris先序遍历

```
// Binary Tree Preorder Traversal
// Morris先序遍历，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public List preorderTraversal(TreeNode root) {
        ArrayList result = new ArrayList<>();
        TreeNode cur = root;
        TreeNode prev = null;

        while (cur != null) {
            if (cur.left == null) {
                result.add(cur.val);
                prev = cur; /* cur刚刚被访问过 */
                cur = cur.right;
            } else {
                /* 查找前驱 */
                TreeNode node = cur.left;
                while (node.right != null && node.right != cur)
                    node = node.right;

                if (node.right == null) { /* 还没线索化，则建立线索 */
                    result.add(cur.val); /* 仅这一行的位置与中序不同 */
                    node.right = cur;
                    prev = cur; /* cur刚刚被访问过 */
                    cur = cur.left;
                } else {    /* 已经线索化，则删除线索  */
                    node.right = null;
                    /* prev = cur; 不能有这句，cur已经被访问 */
                    cur = cur.right;
                }
            }
        }
        return result;
    }
}
```

相关题目

- Binary Tree Inorder Traversal
- Binary Tree Postorder Traversal
- Recover Binary Search Tree

Binary Tree Inorder Traversal

描述

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?

分析

用栈或者Morris遍历。

栈

```
// Binary Tree Inorder Traversal
// 使用栈，时间复杂度O(n)，空间复杂度O(n)
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<>();
        Stack<TreeNode> s = new Stack<>();
        TreeNode p = root;

        while (!s.empty() || p != null) {
            if (p != null) {
                s.push(p);
                p = p.left;
            } else {
                p = s.pop();
                result.add(p.val);
                p = p.right;
            }
        }
        return result;
    }
}
```

Morris中序遍历

```
// Binary Tree Inorder Traversal
// Morris中序遍历，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public List inorderTraversal(TreeNode root) {
        ArrayList result = new ArrayList<>();
        TreeNode cur = root;
        TreeNode prev = null;

        while (cur != null) {
            if (cur.left == null) {
                result.add(cur.val);
                prev = cur;
                cur = cur.right;
            } else {
                /* 查找前驱 */
                TreeNode node = cur.left;
                while (node.right != null && node.right != cur)
                    node = node.right;

                if (node.right == null) { /* 还没线索化，则建立线索 */
                    node.right = cur;
                    /* prev = cur; 不能有这句，cur还没有被访问 */
                    cur = cur.left;
                } else {    /* 已经线索化，则访问节点，并删除线索  */
                    result.add(cur.val);
                    node.right = null;
                    prev = cur;
                    cur = cur.right;
                }
            }
        }
        return result;
    }
}
```

相关题目

- Binary Tree Preorder Traversal
- Binary Tree Postorder Traversal
- Recover Binary Search Tree

Binary Tree Postorder Traversal

描述

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?

分析

用栈或者Morris遍历。

栈

```
// Binary Tree Postorder Traversal
// 使用栈，时间复杂度O(n)，空间复杂度O(n)
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<>();
        Stack<TreeNode> s = new Stack<>();
        /* p，正在访问的结点，q，刚刚访问过的结点*/
        TreeNode p = root;
        TreeNode q = null;

        do {
            while (p != null) { /* 往左下走*/
                s.push(p);
                p = p.left;
            }
            q = null;
            while (!s.empty()) {
                p = s.pop();
                /* 右孩子不存在或已被访问，访问之*/
                if (p.right == q) {
                    result.add(p.val);
                    q = p; /* 保存刚访问过的结点*/
                } else {
                    /* 当前结点不能访问，需第二次进栈*/
                    s.push(p);
                    /* 先处理右子树*/
                    p = p.right;
                    break;
                }
            }
        } while (!s.empty());

        return result;
    }
}
```

Morris后序遍历

```
// Binary Tree Postorder Traversal
// Morris后序遍历，时间复杂度O(n)，空间复杂度O(1)
class Solution {
    public List postorderTraversal(TreeNode root) {
        ArrayList result = new ArrayList<>();
        TreeNode dummy = new TreeNode(-1);
        dummy.left = root;
        TreeNode cur = dummy;
        TreeNode prev = null;

        while (cur != null) {
            if (cur.left == null) {
                prev = cur; /* 必须要有 */
```

```
                cur = cur.right;
            } else {
                TreeNode node = cur.left;
                while (node.right != null && node.right != cur)
                    node = node.right;

                if (node.right == null) { /* 还没线索化，则建立线索 */
                    node.right = cur;
                    prev = cur; /* 必须要有 */
                    cur = cur.left;
                } else { /* 已经线索化，则访问节点，并删除线索  */
                    visit_reverse(cur.left, prev, result);
                    prev.right = null;
                    prev = cur; /* 必须要有 */
                    cur = cur.right;
                }
            }
        }
        return result;
    }
    // 逆转路径
    private static void reverse(TreeNode from, TreeNode to) {
        TreeNode x = from;
        TreeNode y = from.right;
        TreeNode z = null;
        if (from == to) return;

        while (x != to) {
            z = y.right;
            y.right = x;
            x = y;
            y = z;
        }
    }

    // 访问逆转后的路径上的所有结点
    private static void visit_reverse(TreeNode from, TreeNode to,
                                      List result) {
        TreeNode p = to;
        reverse(from, to);

        while (true) {
            result.add(p.val);
            if (p == from)
                break;
            p = p.right;
        }

        reverse(to, from);
    }
}
```

相关题目

- Binary Tree Preorder Traversal
- Binary Tree Inorder Traversal
- Recover Binary Search Tree

Binary Tree Level Order Traversal

描述

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]
]
```

分析

无

递归版

```
// Binary Tree Level Order Traversal
// 递归版，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        traverse(root, 1, result);
        return result;
    }

    void traverse(TreeNode root, int level,
                  List<List<Integer>> result) {
        if (root == null) return;

        if (level > result.size())
            result.add(new ArrayList<>());

        result.get(level-1).add(root.val);
        traverse(root.left, level+1, result);
        traverse(root.right, level+1, result);
    }
}
```

迭代版

```
// Binary Tree Level Order Traversal
// 迭代版，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        Queue<TreeNode> current = new LinkedList<>();
        Queue<TreeNode> next = new LinkedList<>();

        if(root == null) {
            return result;
        } else {
            current.offer(root);
        }

        while (!current.isEmpty()) {
            ArrayList<Integer> level = new ArrayList<>(); // elments in one level
            while (!current.isEmpty()) {
                TreeNode node = current.poll();
                level.add(node.val);
                if (node.left != null) next.add(node.left);
                if (node.right != null) next.add(node.right);
            }
            result.add(level);
            // swap
            Queue<TreeNode> tmp = current;
            current = next;
            next = tmp;
        }
        return result;
    }
}
```

相关题目

- Binary Tree Level Order Traversal II
- Binary Tree Zigzag Level Order Traversal

Binary Tree Level Order Traversal II

描述

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

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

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

return its bottom-up level order traversal as:

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

分析

在上一题 Binary Tree Level Order Traversal 的基础上，`reverse()` 一下即可。

递归版

```
// Binary Tree Level Order Traversal II
// 递归版，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        traverse(root, 1, result);
        Collections.reverse(result);
        return result;
    }

    void traverse(TreeNode root, int level,
                  List<List<Integer>> result) {
        if (root == null) return;

        if (level > result.size())
            result.add(new ArrayList<>());

        result.get(level-1).add(root.val);
        traverse(root.left, level+1, result);
        traverse(root.right, level+1, result);
    }
}
```

迭代版

```
// Binary Tree Level Order Traversal II
// 迭代版，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        Queue<TreeNode> current = new LinkedList<>();
        Queue<TreeNode> next = new LinkedList<>();

        if(root == null) {
            return result;
        } else {
            current.offer(root);
        }

        while (!current.isEmpty()) {
            ArrayList<Integer> level = new ArrayList<>(); // elments in one level
            while (!current.isEmpty()) {
                TreeNode node = current.poll();
                level.add(node.val);
                if (node.left != null) next.add(node.left);
                if (node.right != null) next.add(node.right);
            }
            result.add(level);
            // swap
            Queue<TreeNode> tmp = current;
            current = next;
            next = tmp;
        }
        Collections.reverse(result);
        return result;
    }
}
```

相关题目

- Binary Tree Level Order Traversal
- Binary Tree Zigzag Level Order Traversal

Binary Tree Right Side View

描述

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example, given the following binary tree,

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

You should return `[1, 3, 4]`.

分析

层次遍历。

代码

```java
// Binary Tree Right Side View
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        Queue<TreeNode> current = new LinkedList<>();
        Queue<TreeNode> next = new LinkedList<>();

        if(root == null) {
            return result;
        } else {
            current.offer(root);
        }

        while (!current.isEmpty()) {
            ArrayList<Integer> level = new ArrayList<>(); // elments in one level
            while (!current.isEmpty()) {
                TreeNode node = current.poll();
                level.add(node.val);
                if (node.left != null) next.add(node.left);
                if (node.right != null) next.add(node.right);
            }
            result.add(level.get(level.size()-1));
            // swap
            Queue<TreeNode> tmp = current;
            current = next;
            next = tmp;
        }
        return result;
    }
}
```

Invert Binary Tree

描述

Invert a binary tree.

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

to

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

分析

这题是大名鼎鼎的 Homebrew 的作者 Max Howell 在 Twitter 上发牢骚的那道题。原始 Tweet 地址：https://twitter.com/mxcl/status/608682016205344768

这题最简单的办法，是层次遍历，每次交换左右子树。

但是，这题也可以用递归解决，代码非常短。

解法1 层次遍历

```java
// Invert Binary Tree
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public TreeNode invertTree(TreeNode root) {
        Queue<TreeNode> q = new LinkedList<>();
        if (root != null) q.offer(root);

        while (!q.isEmpty()) {
            TreeNode node = q.poll();
            // swap left and right
            TreeNode tmp = node.left;
            node.left = node.right;
            node.right = tmp;

            if (node.left != null) q.offer(node.left);
            if (node.right != null) q.offer(node.right);
        }
        return root;
    }
}
```

解法2 递归

```java
// Invert Binary Tree
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) return root;

        TreeNode tmp = root.left;
        root.left = invertTree(root.right);
        root.right = invertTree(tmp);

        return root;
    }
}
```

Binary Search Tree Iterator

描述

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.

分析

考察非递归的中序遍历。这道题本质上是写一个二叉树的中序遍历的迭代器。内部设置一个栈，初始化的时候，存储从根节点到最左叶子节点的路径。在遍历的过程中，每次从栈中弹出一个元素，作为当前的返回结果，同时探测一下当前节点是否存在右孩子，如果有，则进入右孩子，并把从该右孩子到最左叶子节点的所有节点入栈。

代码

```
// Binary Search Tree Iterator
public class BSTIterator {
    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        while (root != null) {
            stack.push(root);
            root = root.left;
        }
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        final TreeNode node = stack.pop();
        if (node.right != null) {
            TreeNode p = node.right;
            while (p != null) {
                stack.push(p);
                p = p.left;
            }

        }
        return node.val;
    }
    private Stack<TreeNode> stack;
}
```

Binary Tree Zigzag Level Order Traversal

描述

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

For example:

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

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

return its zigzag level order traversal as:

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

分析

广度优先遍历，用一个bool记录是从左到右还是从右到左，每一层结束就翻转一下。

递归版

```
// Binary Tree Zigzag Level Order Traversal
// 递归版，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List> zigzagLevelOrder(TreeNode root) {
        List> result = new ArrayList<>();
        traverse(root, 1, result, true);
        return result;
    }

    private static void traverse(TreeNode root, int level, List> result,
                                 boolean left_to_right) {
        if (root == null) return;

        if (level > result.size())
            result.add(new ArrayList<>());

        if (left_to_right)
            result.get(level-1).add(root.val);
        else
            result.get(level-1).add(0, root.val);

        traverse(root.left, level+1, result, !left_to_right);
        traverse(root.right, level+1, result, !left_to_right);
    }
}
```

迭代版

```
// Binary Tree Zigzag Level Order Traversal
// 广度优先遍历，用一个bool记录是从左到右还是从右到左，每一层结束就翻转一下。
// 迭代版，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List> zigzagLevelOrder(TreeNode root) {
        List> result = new ArrayList<>();
        Queue current = new LinkedList<>();
        Queue next = new LinkedList<>();
        boolean left_to_right = true;

        if(root == null) {
            return result;
        } else {
            current.offer(root);
        }

        while (!current.isEmpty()) {
            ArrayList level = new ArrayList<>(); // elments in one level
            while (!current.isEmpty()) {
                TreeNode node = current.poll();
                level.add(node.val);
                if (node.left != null) next.offer(node.left);
                if (node.right != null) next.offer(node.right);
            }
            if (!left_to_right) Collections.reverse(level);
            result.add(level);
            left_to_right = !left_to_right;
            // swap
            Queue tmp = current;
            current = next;
            next = tmp;
        }
        return result;
    }
}
```

相关题目

- Binary Tree Level Order Traversal
- Binary Tree Level Order Traversal II

Recover Binary Search Tree

描述

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?

分析

`O(logn)`空间的解法是，中序递归遍历，用两个指针存放在遍历过程中碰到的两处逆向的位置。

本题要求`O(1)`空间，只能用Morris中序遍历。

中序遍历，递归方式

```
// Recover Binary Search Tree
// 中序遍历,递归
// 时间复杂度O(n)，空间复杂度O(logn)
// 本代码仅仅是为了帮助理解题目
public class Solution {
    private TreeNode p1 = null;
    private TreeNode p2 = null;
    private TreeNode prev = null;

    public void recoverTree(TreeNode root) {
        inOrder( root);
        // swap
        int tmp = p1.val;
        p1.val = p2.val;
        p2.val = tmp;
    }

    private void inOrder(TreeNode root) {
        if ( root ==  null ) return;
        if ( root.left != null ) inOrder(root.left);

        if ( prev != null && root.val < prev.val ) {
            if ( p1 == null) {
                p1 = prev;
                p2 = root;
            } else {
                p2 = root;
            }
        }
        prev = root;
        if ( root.right != null ) inOrder(root.right);
    }
}
```

Morris中序遍历

```
// Recover Binary Search Tree
// Morris中序遍历，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public void recoverTree(TreeNode root) {
        TreeNode[] broken = new TreeNode[2];
        TreeNode prev = null;
        TreeNode cur = root;

        while (cur != null) {
            if (cur.left == null) {
                detect(broken, prev, cur);
                prev = cur;
                cur = cur.right;
            } else {
                TreeNode node = cur.left;

                while (node.right != null && node.right != cur)
                    node = node.right;

                if (node.right == null) {
                    node.right = cur;
                    //prev = cur; 不能有这句！因为cur还没有被访问
                    cur = cur.left;
                } else {
                    detect(broken, prev, cur);
                    node.right = null;
                    prev = cur;
                    cur = cur.right;
                }
            }
        }
        // swap
        int tmp = broken[0].val;
        broken[0].val = broken[1].val;
        broken[1].val = tmp;
    }

    void detect(TreeNode[] broken, TreeNode prev,
                TreeNode current) {
        if (prev != null && prev.val > current.val) {
            if (broken[0] == null) {
                broken[0] = prev;
            } //不能用else，例如 {0,1}，会导致最后 swap时second为nullptr，
            //会 Runtime Error
            broken[1] = current;
        }
    }
}
```

相关题目

- Binary Tree Inorder Traversal

Same Tree

描述

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.

分析

无

递归版

```
// Same Tree
// 递归版，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;   // 终止条件
        if (p == null || q == null) return false;  // 剪枝
        return p.val == q.val      // 三方合并
                && isSameTree(p.left, q.left)
                && isSameTree(p.right, q.right);
    }
}
```

迭代版

```
// Same Tree
// 迭代版，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        Stack<TreeNode> s = new Stack<>();
        s.push(p);
        s.push(q);

        while(!s.empty()) {
            p = s.pop();
            q = s.pop();

            if (p == null && q == null) continue;
            if (p == null || q == null) return false;
            if (p.val != q.val) return false;

            s.push(p.left);
            s.push(q.left);

            s.push(p.right);
            s.push(q.right);
        }
        return true;
    }
}
```

相关题目

- Symmetric Tree

Symmetric Tree

描述

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

For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

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

But the following `[1,2,2,null,3,null,3]` is not:

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

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

分析

无

递归版

```
// Symmetric Tree
// 递归版，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) return true;
        return isSymmetric(root.left, root.right);
    }
    private static boolean isSymmetric(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;   // 终止条件
        if (p == null || q == null) return false;  // 终止条件
        return p.val == q.val      // 三方合并
                && isSymmetric(p.left, q.right)
                && isSymmetric(p.right, q.left);
    }
}
```

迭代版

```
// Symmetric Tree
// 迭代版，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public boolean isSymmetric (TreeNode root) {
        if (root == null) return true;

        Stack<TreeNode> s = new Stack<>();
        s.push(root.left);
        s.push(root.right);

        while (!s.isEmpty()) {
            TreeNode p = s.pop ();
            TreeNode q = s.pop ();

            if (p == null && q == null) continue;
            if (p == null || q == null) return false;
            if (p.val != q.val) return false;

            s.push(p.left);
            s.push(q.right);

            s.push(p.right);
            s.push(q.left);
        }

        return true;
    }
}
```

相关题目

- Same Tree

Balanced Binary Tree

描述

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.

分析

无

代码

```
// Balanced Binary Tree
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public boolean isBalanced (TreeNode root) {
        return balancedHeight (root) >= 0;
    }

    /**
     * Returns the height of `root` if `root` is a balanced tree,
     * otherwise, returns `-1`.
     */
    private static int balancedHeight (TreeNode root) {
        if (root == null) return 0;  // 终止条件

        int left = balancedHeight (root.left);
        int right = balancedHeight (root.right);

        if (left < 0 || right < 0 || Math.abs(left - right) > 1) return -1;  // 剪枝

        return Math.max(left, right) + 1; // 三方合并
    }
}
```

Flatten Binary Tree to Linked List

描述

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

For example, Given

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

The flattened tree should look like:

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

分析

无

递归版1

```
// Flatten Binary Tree to Linked List
// 递归版1，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public void flatten(TreeNode root) {
        if (root == null) return;  // 终止条件

        flatten(root.left);
        flatten(root.right);

        if (root.left == null) return;

        // 三方合并，将左子树所形成的链表插入到root和root->right之间
        TreeNode p = root.left;
        while(p.right != null) p = p.right; //寻找左链表最后一个节点
        p.right = root.right;
        root.right = root.left;
        root.left = null;
    }
}
```

递归版2

```
// Flatten Binary Tree to Linked List
// 递归版2
// @author 王顺达(http://weibo.com/u/1234984145)
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public void flatten(TreeNode root) {
        flatten(root, null);
    }
    // 把root所代表树变成链表后，tail跟在该链表后面
    private static TreeNode flatten(TreeNode root, TreeNode tail) {
        if (root == null) return tail;

        root.right = flatten(root.left, flatten(root.right, tail));
        root.left = null;
        return root;
    }
}
```

Populating Next Right Pointers in Each Node II

描述

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

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

Note: You may only use constant extra space.

For example, Given the following binary tree,

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

After calling your function, the tree should look like:

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

分析

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.

递归版

```
// Populating Next Right Pointers in Each Node II
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public void connect(TreeLinkNode root) {
        if (root == null) return;

        TreeLinkNode dummy = new TreeLinkNode(-1);
        for (TreeLinkNode curr = root, prev = dummy;
                curr != null; curr = curr.next) {
            if (curr.left != null){
                prev.next = curr.left;
                prev = prev.next;
            }
            if (curr.right != null){
                prev.next = curr.right;
                prev = prev.next;
            }
        }
        connect(dummy.next);
    }
}
```

迭代版

```
// Populating Next Right Pointers in Each Node II
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public void connect(TreeLinkNode root) {
        while (root!= null ) {
            TreeLinkNode next = null; // the first node of next level
            TreeLinkNode prev = null; // previous node on the same level
            for (; root != null; root = root.next) {
                if (next == null) next = root.left != null ? root.left : root.right;

                if (root.left != null) {
                    if (prev != null) prev.next = root.left;
                    prev = root.left;
                }
                if (root.right != null) {
                    if (prev != null) prev.next = root.right;
                    prev = root.right;
                }
            }
            root = next; // turn to next level
        }
    }
}
```

相关题目

- Populating Next Right Pointers in Each Node

本节主要讲二叉树的构建。

Construct Binary Tree from Preorder and Inorder Traversal

描述

Given preorder and inorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

分析

无

代码

```
// Construct Binary Tree from Preorder and Inorder Traversal
// 递归，时间复杂度O(n)，空间复杂度O(\logn)
public class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return buildTree(preorder, 0, preorder.length,
                inorder, 0, inorder.length);
    }

    TreeNode buildTree(int[] preorder, int begin1, int end1,
                       int[] inorder, int begin2, int end2) {
        if (begin1 == end1) return null;
        if (begin2 == end2) return null;

        TreeNode root = new TreeNode(preorder[begin1]);

        int inRootPos = find(inorder, begin2, end2, preorder[begin1]);
        int leftSize = inRootPos - begin2;

        root.left = buildTree(preorder, begin1 + 1, begin1 + leftSize + 1,
                inorder, begin2, begin2 + leftSize);
        root.right = buildTree(preorder, begin1 + leftSize + 1, end1,
                inorder, inRootPos + 1, end2);

        return root;
    }
    private static int find(int[] array, int begin, int end, int val) {
        for (int i = begin; i < end; ++i) {
            if (array[i] == val) return i;
        }
        return -1;
    }
}
```

相关题目

- Construct Binary Tree from Inorder and Postorder Traversal

Construct Binary Tree from Inorder and Postorder Traversal

描述

Given inorder and postorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

分析

无

代码

```
// Construct Binary Tree from Inorder and Postorder Traversal
// 递归，时间复杂度O(n)，空间复杂度O(\logn)
public class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        return buildTree(inorder, 0, inorder.length,
                postorder, 0, postorder.length);
    }

    TreeNode buildTree(int[] inorder, int begin1, int end1,
                       int[] postorder, int begin2, int end2) {
        if (begin1 ==end1) return null;
        if (begin2 ==end2) return null;

        int val = postorder[end2 - 1];
        TreeNode root = new TreeNode(val);

        int in_root_pos = find(inorder, begin1, end1, val);
        int left_size = in_root_pos - begin1;
        int post_left_last = begin2 + left_size;

        root.left = buildTree(inorder, begin1, in_root_pos,
                postorder, begin2, post_left_last);
        root.right = buildTree(inorder, in_root_pos + 1, end1,
                postorder, post_left_last, end2 - 1);

        return root;
    }
    private static int find(int[] array, int begin, int end, int val) {
        for (int i = begin; i < end; ++i) {
            if (array[i] == val) return i;
        }
        return -1;
    }
}
```

相关题目

- Construct Binary Tree from Preorder and Inorder Traversal

本节主要讲二叉查找树。

Unique Binary Search Trees

描述

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

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

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

分析

如果把上例的顺序改一下，就可以看出规律了。

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

比如，以1为根的树的个数，等于左子树的个数乘以右子树的个数，左子树是0个元素的树，右子树是2个元素的树。以2为根的树的个数，等于左子树的个数乘以右子树的个数，左子树是1个元素的树，右子树也是1个元素的树。依此类推。

当数组为 `1,2,3,...,n` 时，基于以下原则的构建的BST树具有唯一性： \textbf{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。}

定义 `f(i)` 为以 `[1,i]` 能产生的Unique Binary Search Tree的数目，则

如果数组为空，毫无疑问，只有一种BST，即空树， `f(0)=1` 。

如果数组仅有一个元素{1}，只有一种BST，单个节点， `f(1)=1` 。

如果数组有两个元素{1,2}， 那么有如下两种可能

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

$f(2) = f(0) * f(1)$, when 1 as root

$+ f(1) * f(0)$, when 2 as root

再看一看3个元素的数组，可以发现BST的取值方式如下：

$f(3) = f(0) * f(2)$, when 1 as root

$+f(1)*f(1)$, when 2 as root

$+f(2)*f(0)$, when 3 as root

所以，由此观察，可以得出 `f` 的递推公式为

$f(i)=\sum_{k=1}^{i} f(k-1)\times f(i-k)$

至此，问题划归为一维动态规划。

代码

```
// Unique Binary Search Trees
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public int numTrees(int n) {
        int[] f = new int[n + 1];

        f[0] = 1;
        f[1] = 1;
        for (int i = 2; i
```

相关题目

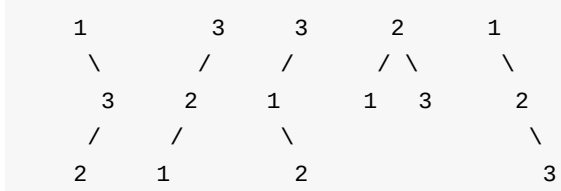
- Unique Binary Search Trees II

Unique Binary Search Trees II

描述

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

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



分析

见前面一题。

代码

```
// Unique Binary Search Trees II
// 时间复杂度TODO，空间复杂度TODO
public class Solution {
    public List<TreeNode> generateTrees(int n) {
        if (n == 0) return new ArrayList<>();
        return generate(1, n);
    }
    private static List<TreeNode > generate(int start, int end) {
        List<TreeNode> subTree = new ArrayList<>();
        if (start > end) {
            subTree.add(null);
            return subTree;
        }
        for (int k = start; k <= end; k++) {
            List<TreeNode> leftSubs = generate(start, k - 1);
            List<TreeNode> rightSubs = generate(k + 1, end);
            for (TreeNode i : leftSubs) {
                for (TreeNode j : rightSubs) {
                    TreeNode node = new TreeNode(k);
                    node.left = i;
                    node.right = j;
                    subTree.add(node);
                }
            }
        }
        return subTree;
    }
}
```

相关题目

- Unique Binary Search Trees

Validate Binary Search Tree

描述

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.

分析

无

代码

```
// Validate Binary Search Tree
// 时间复杂度O(n)，空间复杂度O(\logn)
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, INT_MIN, INT_MAX);
    }

    bool isValidBST(TreeNode* root, int lower, int upper) {
        if (root == nullptr) return true;

        return root.val > lower && root.val < upper
                && isValidBST(root.left, lower, root.val)
                && isValidBST(root.right, root.val, upper);
    }
}
```

相关题目

- Validate Binary Search Tree

Convert Sorted Array to Binary Search Tree

描述

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

分析

二分法。

代码

```
// Convert Sorted Array to Binary Search Tree
// 二分法，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public TreeNode sortedArrayToBST (int[] nums) {
        return sortedArrayToBST(nums, 0, nums.length);
    }

    private static TreeNode sortedArrayToBST (int[] nums, int begin, int end) {
        int length = end - begin;
        if (length < 1) return null;  // 终止条件

        // 三方合并
        int mid = begin + length / 2;
        TreeNode root = new TreeNode (nums[mid]);
        root.left = sortedArrayToBST(nums, begin, mid);
        root.right = sortedArrayToBST(nums, mid + 1, end);

        return root;
    }
}
```

相关题目

- Convert Sorted List to Binary Search Tree

Convert Sorted List to Binary Search Tree

描述

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

分析

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

分治法，自顶向下

分治法，类似于 Convert Sorted Array to Binary Search Tree，自顶向下，复杂度 `O(nlogn)`。

```
// Convert Sorted List to Binary Search Tree
// 二分法，类似于 Convert Sorted Array to Binary Search Tree，
// 自顶向下，时间复杂度O(nlogn)，空间复杂度O(logn)
class Solution {
    public TreeNode sortedListToBST (ListNode head) {
        if(head == null) return null;
        if(head.next == null) return new TreeNode(head.val);

        ListNode mid = cutAtMiddle(head);

        TreeNode root = new TreeNode(mid.val);
        root.left = sortedListToBST(head);
        root.right = sortedListToBST(mid.next);

        return root;
    }

    ListNode cutAtMiddle(ListNode head) {
        if(head == null) return null;

        ListNode fast = head;
        ListNode slow = head;
        ListNode prev_slow = head;

        while(fast != null && fast.next != null){
            prev_slow = slow;
            slow = slow.next;
            fast = fast.next.next;
        }

        prev_slow.next = null;
        return slow;
    }
}
```

自底向上

```java
// Convert Sorted List to Binary Search Tree
// bottom-up，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public TreeNode sortedListToBST(ListNode head) {
        int len = 0;
        ListNode p = head;
        while (p != null) {
            len++;
            p = p.next;
        }
        return sortedListToBST(new Container(head), 0, len - 1);
    }
    private static TreeNode sortedListToBST(Container list, int start, int end) {
        if (start > end) return null;

        int mid = start + (end - start) / 2;
        TreeNode leftChild = sortedListToBST(list, start, mid - 1);
        TreeNode parent = new TreeNode(list.p.val);
        parent.left = leftChild;
        list.p = list.p.next;
        parent.right = sortedListToBST(list, mid + 1, end);
        return parent;
    }
    // 模拟二级指针
    static class Container {
        ListNode p;
        public Container(ListNode p) {
            this.p = p;
        }
    }
}
```

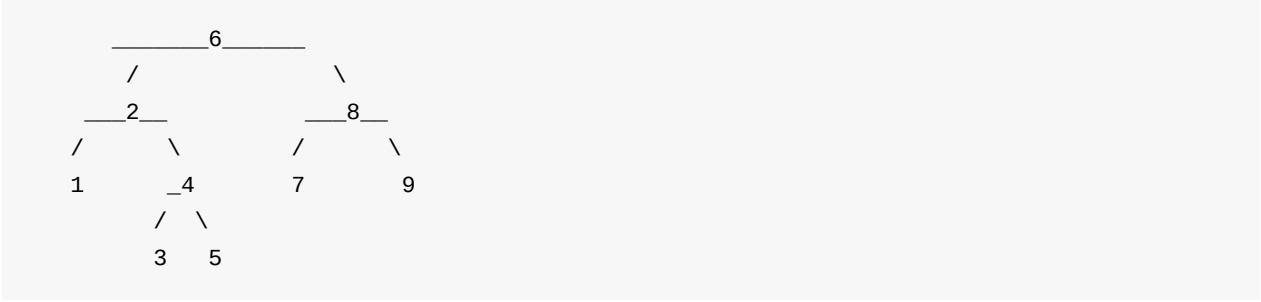
相关题目

- Convert Sorted Array to Binary Search Tree

LCA of BST

描述

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.



For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

分析

根据二叉搜索树的性质，两个子节点 `p`，`q` 和根节点 `root` 的关系，有以下四种情况：

1. 两个子节点都在树的左子树上
2. 两个子节点都在树的右子树上
3. 一个子节点在左子树，一个子节点在右子树
4. 一个子节点的值和根节点的值相等

以题目中的树为例，节点1和节点4为情况1，节点7和节点9为情况2，节点1和节点7为情况3，节点2和4为情况4。若为情况3或4，当前节点即为最近公共祖先，若为情况1或2，则还需递归到左或右子树上，继续这个过程。

该算法的时间复杂度为 `O(h)`，`h` 为树的高度。

解法1 递归

```
// LCA of BST
// Time Complexity: O(h), Space Complexity: O(h)
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return null;

        if (Math.max(p.val, q.val) < root.val) {
            return lowestCommonAncestor(root.left, p, q);
        } else if (Math.min(p.val, q.val) > root.val) {
            return lowestCommonAncestor(root.right, p, q);
        } else {
            return root;
        }
    }
}
```

解法2 迭代

```
// LCA of BST
// Time Complexity: O(h), Space Complexity: O(1)
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        while (root != null) {
            if (Math.max(p.val, q.val) < root.val) {
                root = root.left;
            } else if (Math.min(p.val, q.val) > root.val) {
                root = root.right;
            } else {
                return root;
            }
        }
        return null;
    }
}
```

相关题目

- LCA of Binary Tree

Kth Smallest Element in a BST

描述

Given a binary search tree, write a function `kthSmallest` to find the kth smallest element in it.

Note:

You may assume `k` is always valid, `1 ≤ k ≤` BST's total elements.

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Hint:

- Try to utilize the property of a BST.
- What if you could modify the BST node's structure?
- The optimal runtime complexity is O(height of BST).

分析

最简单的办法，中序遍历，即可以得到递增序列，从而可以找到第k大的元素。时间复杂度`O(k)`。

如果能够修改节点的数据结构`TreeNode`，可以增加一个字段`leftCnt`，表示左子树的节点个数。设当前节点为`root`，

- 若`k == root.leftCnt+1`，则返回root
- 若`k > node.leftCnt`，则`k -= root.leftCnt+1`，`root=root.right`
- 否则，`node = node.left`

算法复杂度为`O(height of BST)`。

解法1

```java
// Kth Smallest Element in a BST
// Time Complexity: O(k), Space Complexity: O(h)
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        Stack s = new Stack<>();
        TreeNode p = root;

        while (!s.empty() || p != null) {
            if (p != null) {
                s.push(p);
                p = p.left;
            } else {
                p = s.pop();
                --k;
                if (k == 0) {
                    return p.val;
                }
                p = p.right;
            }
        }
        return -1;
    }
}
```

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->r->l`，`r->root->l`, `r->l->root` 。

Minimum Depth of Binary Tree

描述

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.

分析

无

递归版

```
// Minimum Depth of Binary Tree
// 递归版，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public int minDepth(TreeNode root) {
        return minDepth(root, false);
    }
    private static int minDepth(TreeNode root, boolean hasbrother) {
        if (root == null) return hasbrother ? Integer.MAX_VALUE : 0;

        return 1 + Math.min(minDepth(root.left, root.right != null),
                minDepth(root.right, root.left != null));
    }
}
```

迭代版

```java
// Minimum Depth of Binary Tree
// 迭代版，时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public int minDepth(TreeNode root) {
        if (root == null) return 0;

        int result = Integer.MAX_VALUE;
        Stack<Pair> s = new Stack<>();
        s.push(new Pair(root, 1));

        while (!s.empty()) {
            final Pair p = s.pop();
            TreeNode node = p.node;
            int depth = p.depth;

            if (node.left == null && node.right == null)
                result = Math.min(result, depth);

            if (node.left != null && result > depth) // 深度控制，剪枝
                s.push(new Pair(node.left, depth + 1));

            if (node.right != null && result > depth) // 深度控制，剪枝
                s.push(new Pair(node.right, depth + 1));
        }

        return result;
    }

    static class Pair {
        TreeNode node;
        int depth;
        public Pair(TreeNode node, int depth) {
            this.node = node;
            this.depth = depth;
        }
    }
}
```

相关题目

- Maximum Depth of Binary Tree

Maximum Depth of Binary Tree

描述

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.

分析

无

代码

```
// Maximum Depth of Binary Tree
// 时间复杂度O(n)，空间复杂度O(logn)
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;

        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}
```

相关题目

- Minimum Depth of Binary Tree

Path Sum

描述

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`,

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

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

分析

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。

代码

```
// Path Sum
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if (root == null) return false;

        if (root.left == null && root.right == null) // leaf
            return sum == root.val;

        return hasPathSum(root.left, sum - root.val)
                || hasPathSum(root.right, sum - root.val);
    }
}
```

相关题目

- Path Sum II

Path Sum II

描述

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

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

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

return

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

分析

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。

代码

```
// Path Sum II
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> result = new ArrayList<>();
        ArrayList<Integer>  cur = new ArrayList<>(); // 中间结果
        pathSum(root, sum, cur, result);
        return result;
    }
    private static void pathSum(TreeNode root, int gap, ArrayList<Integer> cur,
                                List<List<Integer>> result) {
        if (root == null) return;

        cur.add(root.val);

        if (root.left == null && root.right == null) { // leaf
            if (gap == root.val)
                result.add(new ArrayList<>(cur));
        }
        pathSum(root.left, gap - root.val, cur, result);
        pathSum(root.right, gap - root.val, cur, result);

        cur.remove(cur.size() - 1);
    }
}
```

相关题目

- Path Sum

Binary Tree Maximum Path Sum

描述

Given a binary tree, find the maximum path sum.

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

```
  1
 / \
2   3
```

Return `6`.

分析

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，见这节Maximum Subarray。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。

代码

```
// Binary Tree Maximum Path Sum
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public int maxPathSum(TreeNode root) {
        max_sum = Integer.MIN_VALUE;
        dfs(root);
        return max_sum;
    }
    private int max_sum;
    private int dfs(TreeNode root) {
        if (root == null) return 0;
        int l = dfs(root.left);
        int r = dfs(root.right);
        int sum = root.val;
        if (l > 0) sum += l;
        if (r > 0) sum += r;
        max_sum = Math.max(max_sum, sum);
        return Math.max(r, l) > 0 ? Math.max(r, l) + root.val : root.val;
    }
}
```

注意，最后return的时候，只返回一个方向上的值，为什么？这是因为在递归中，只能向父节点返回，不可能存在L->root->R的路径，只可能是L->root或R->root。

相关题目

- Maximum Subarray
- Maximum Product Subarray

Populating Next Right Pointers in Each Node

描述

Given a binary tree

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

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,

```
         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
```

分析

无

代码

```java
// Populating Next Right Pointers in Each Node
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public void connect(TreeLinkNode root) {
        connect(root, null);
    }
    private static void connect(TreeLinkNode root, TreeLinkNode sibling) {
        if (root == null) return;
        else root.next = sibling;

        connect(root.left, root.right);
        if (sibling != null) connect(root.right, sibling.left);
        else connect(root.right, null);
    }
}
```

相关题目

- Populating Next Right Pointers in Each Node II

Sum Root to Leaf Numbers

描述

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`.

分析

无

代码

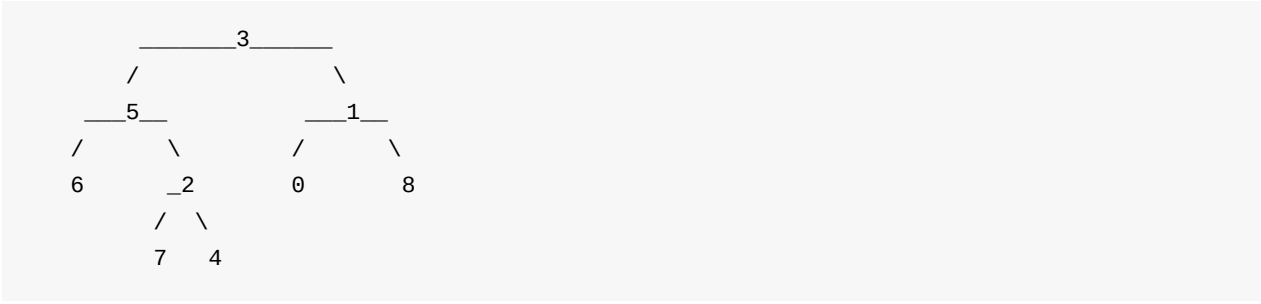
```
// Sum root to leaf numbers
// 时间复杂度O(n)，空间复杂度O(logn)
public class Solution {
    public int sumNumbers(TreeNode root) {
        return dfs(root, 0);
    }
    private static int dfs(TreeNode root, int sum) {
        if (root == null) return 0;
        if (root.left == null && root.right == null)
            return sum * 10 + root.val;

        return dfs(root.left, sum * 10 + root.val) +
                dfs(root.right, sum * 10 + root.val);
    }
}
```

LCA of Binary Tree

描述

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”



For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

分析

用自底向上(bottom-up)的思路，先看看是否能在 `root` 的左子树中找到 `p` 或 `q` ，再看看能否在右子树中找到，

- 如果两边都能找到，说明当前节点就是最近公共祖先
- 如果左边没找到，则说明 `p` 和 `q` 都在右子树
- 如果右边没找到，则说明 `p` 和 `q` 都在左子树

代码

```
// Lowest Common Ancestor of a Binary Tree
// Time complexity: O(n), Space complexity: O(h)
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // if root is null or found p or q
        if (root == null || root == p || root == q) return root;
        // find p or q in the left subtree
        final TreeNode left = lowestCommonAncestor(root.left, p, q);
        // find p or q in the right subtree
        final TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left != null && right != null) return root;
        else return left == null ? right : left;
    }
}
```

相关题目

- LCA of BST

本节主要讲线段树。

Range Sum Query - Mutable

描述

Given an integer array `nums`, find the sum of the elements between indices `i` and `j` (`i ≤ j`), inclusive.

The `update(i, val)` function modifies `nums` by updating the element at index `i` to `val`.

Example:

Given `nums = [1, 3, 5]`

```
sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

Note:

1. The array is only modifiable by the update function.
2. You may assume the number of calls to update and sumRange function is distributed evenly.

分析

由于需要求任意段的和，且会随机修改元素，用线段树(Segment Tree)再合适不过了。

另外一个数据结构，树状数组(Binary Indexed Tree)，也适合解这道题。

解法1 线段树

```
// Range Sum Query - Mutable
// Segment Tree
public class NumArray {
    private SegmentTreeNode root;

    // Time Complexity: O(n)
    public NumArray(int[] nums) {
        this.root = buildTree(nums, 0, nums.length);
    }

    // Time Complexity: O(log n)
    void update(int i, int val) {
        updateHelper(this.root, i, val);
    }

    // Time Complexity: O(log n)
    public int sumRange(int i, int j) {
        return sumRangeHelper(this.root, i, j+1);
    }
```

```
    private static SegmentTreeNode buildTree(int[] nums, int begin, int end) {
        if (nums == null || nums.length == 0 || begin >= end)
            return null;
        if (begin == end - 1) // one element
            return new SegmentTreeNode(begin, end, nums[begin]);

        final SegmentTreeNode root = new SegmentTreeNode(begin, end);
        final int middle = begin + (end - begin) / 2;
        root.left = buildTree(nums, begin, middle);
        root.right = buildTree(nums, middle, end);
        root.sum = root.left.sum + root.right.sum;

        return root;
    }

    private static void updateHelper(SegmentTreeNode root, int i, int val) {
        if (root.begin == root.end - 1 && root.begin == i) {
            root.sum = val;
            return;
        }

        final int middle = root.begin + (root.end - root.begin) / 2;
        if (i < middle) {
            updateHelper(root.left, i, val);
        } else {
            updateHelper(root.right, i, val);
        }

        root.sum = root.left.sum + root.right.sum;
    }
    private static int sumRangeHelper(SegmentTreeNode root, int begin, int end) {
        if (root == null || begin >=root.end || end <= root.begin)
            return 0;
        if (begin <= root.begin && end >= root.end)
            return root.sum;

        final int middle = root.begin + (root.end - root.begin) / 2;
        return sumRangeHelper(root.left, begin, Math.min(end, middle)) +
                sumRangeHelper(root.right, Math.max(middle, begin), end);
    }

    static class SegmentTreeNode {
        private int begin;
        private int end;
        private int sum;
        private SegmentTreeNode left;
        private SegmentTreeNode right;

        public SegmentTreeNode(int begin, int end, int sum) {
            this.begin = begin;
            this.end = end;
            this.sum = sum;
```

```
        }
        public SegmentTreeNode(int begin, int end) {
            this(begin, end, 0);
        }
    }
}
```

解法2 树状数组

```java
// Range Sum Query - Mutable
// Binary Indexed Tree
public class NumArray {
    private int[] nums;
    private int[] bit;

    // Time Complexity: O(n)
    public NumArray(int[] nums) {
        // index 0 is unused
        this.nums = new int[nums.length + 1];
        this.bit = new int[nums.length + 1];

        for (int i = 0; i < nums.length; ++i) {
            update(i, nums[i]);
        }
    }

    // Time Complexity: O(log n)
    public void update(int index, int val) {
        final int diff = val - nums[index + 1];
        for (int i = index + 1; i < nums.length; i += lowbit(i)) {
            bit[i] += diff;
        }
        nums[index + 1] = val;
    }

    // Time Complexity: O(log n)
    public int sumRange(int i, int j) {
        return read(j + 1) - read(i);
    }

    private int read(int index) {
        int result = 0;
        for (int i = index; i > 0; i -= lowbit(i)) {
            result += bit[i];
        }
        return result;
    }

    private static int lowbit(int x) {
        return x & (-x);  // must use parentheses
    }
}
```

相关题目

- Range Sum Query - Immutable
- Range Sum Query 2D - Immutable

排序

本章主要讲解各种排序算法。

本节主要讲插入排序。

Insertion Sort List

描述

Sort a linked list using insertion sort.

分析

无

代码

```
// Insertion Sort List
// 时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode dummy = new ListNode(Integer.MIN_VALUE);
        //dummy.next = head;

        for (ListNode cur = head; cur != null;) {
            ListNode pos = findInsertPos(dummy, cur.val);
            ListNode tmp = cur.next;
            cur.next = pos.next;
            pos.next = cur;
            cur = tmp;
        }
        return dummy.next;
    }

    ListNode findInsertPos(ListNode head, int x) {
        ListNode pre = null;
        for (ListNode cur = head; cur != null && cur.val <= x;
             pre = cur, cur = cur.next)
            ;
        return pre;
    }
}
```

相关题目

- Sort List

本节主要讲归并排序。

Merge Two Sorted Arrays

描述

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

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

分析

无

代码

```
// Merge Two Sorted Arrays
// 时间复杂度O(m+n)，空间复杂度O(1)
public class Solution {
    public void merge(int[] A, int m, int[] B, int n) {
        int ia = m - 1, ib = n - 1, icur = m + n - 1;
        while(ia >= 0 && ib >= 0) {
            A[icur--] = A[ia] >= B[ib] ? A[ia--] : B[ib--];
        }
        while(ib >= 0) {
            A[icur--] = B[ib--];
        }
    }
}
```

相关题目

- Merge Two Sorted Lists
- Merge k Sorted Lists

Merge Two Sorted Lists

描述

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

分析

无

代码

```
// Merge Two Sorted Lists
// 时间复杂度O(min(m,n))，空间复杂度O(1)
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) return l2;
        if (l2 == null) return l1;
        ListNode dummy = new ListNode(-1);
        ListNode p = dummy;
        for (; l1 != null && l2 != null; p = p.next) {
            if (l1.val > l2.val) { p.next = l2; l2 = l2.next; }
            else { p.next = l1; l1 = l1.next; }
        }
        p.next = l1 != null ? l1 : l2;
        return dummy.next;
    }
}
```

相关题目

- Merge Two Sorted Arrays
- Merge k Sorted Lists

Merge k Sorted Lists

描述

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

分析

可以复用 Merge Two Sorted Lists 的函数

代码

```
// Merge k Sorted Lists
// 时间复杂度O(n1+n2+...)，空间复杂度O(1)
// TODO: 会超时
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists.length == 0) return null;

        ListNode p = lists[0];
        for (int i = 1; i < lists.length; i++) {
            p = mergeTwoLists(p, lists[i]);
        }
        return p;
    }

    // Merge Two Sorted Lists
    ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode head = new ListNode(-1);
        for (ListNode p = head; l1 != null || l2 != null; p = p.next) {
            int val1 = l1 == null ? Integer.MAX_VALUE : l1.val;
            int val2 = l2 == null ? Integer.MAX_VALUE : l2.val;
            if (val1
```

相关题目

- Merge Two Sorted Arrays
- Merge Two Sorted Lists

Sort List

描述

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

分析

常数空间且`O(nlogn)`，单链表适合用归并排序，双向链表适合用快速排序。本题可以复用 Merge Two Sorted Lists 的代码。

代码

```java
// Sort List
// 归并排序，时间复杂度O(nlogn)，空间复杂度O(1)
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)return head;

        final ListNode middle = findMiddle(head);
        final ListNode head2 = middle.next;
        middle.next = null; // 断开

        final ListNode l1 = sortList(head);  // 前半段排序
        final ListNode l2 = sortList(head2);  // 后半段排序
        return mergeTwoLists(l1, l2);
    }

    // Merge Two Sorted Lists
    private static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        for (ListNode p = dummy; l1 != null || l2 != null; p = p.next) {
            int val1 = l1 == null ? Integer.MAX_VALUE : l1.val;
            int val2 = l2 == null ? Integer.MAX_VALUE : l2.val;
            if (val1 <= val2) {
                p.next = l1;
                l1 = l1.next;
            } else {
                p.next = l2;
                l2 = l2.next;
            }
        }
        return dummy.next;
    }
    // 快慢指针找到中间节点
    private static ListNode findMiddle(ListNode head) {
        if (head == null) return null;

        ListNode slow = head;
        ListNode fast = head.next;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

相关题目

- Insertion Sort List

本节主要讲快速排序。

Sort Colors

描述

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

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

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

Follow up:

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

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

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

分析

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

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

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

代码1

```
// Sort Colors
// Counting Sort
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public void sortColors(int[] nums) {
        int[] counts = new int[3]; // 记录每个颜色出现的次数

        for (int i = 0; i < nums.length; i++)
            counts[nums[i]]++;

        for (int i = 0, index = 0; i < 3; i++)
            for (int j = 0; j < counts[i]; j++)
                nums[index++] = i;

    }
}
```

代码2

```
// Sort Colors
// 双指针，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public void sortColors(int[] A) {
        // 一个是red的index，一个是blue的index，两边往中间走
        int red = 0, blue = A.length - 1;

        for (int i = 0; i < blue + 1;) {
            if (A[i] == 0)
                swap (A, i++, red++);
            else if (A[i] == 2)
                swap(A, i, blue--);
            else
                i++;
        }
    }
    private static void swap(int[] A, int i, int j) {
        int tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }
}
```

代码3

```
// Sort Colors
// 重新实现 partition()
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public void sortColors(int[] nums) {
        partition(nums, partition(nums, 0, nums.length, new EqualTo(0)),
                nums.length, new EqualTo(1));
    }
    private static int partition(int[] nums, int begin, int end, EqualTo predicate) {
        int pos = begin;

        for (; begin != end; ++begin)
            if (predicate.apply(nums[begin]))
                swap(nums, begin,pos++);

        return pos;
    }
    static class EqualTo {
        private final int target;
        public EqualTo(int target) {
            this.target = target;
        }
        public boolean apply(int x) {
            return x == target;
        }
    }
    private static void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
```

相关题目

- First Missing Positive

Kth Largest Element in an Array

描述

Find the `k` -th largest element in an unsorted array.

For example, given `[3,2,1,5,6,4]` and `k = 2` , return 5.

Note:

You may assume `k` is always valid, 1 ≤ k ≤ array's length.

分析

这题是一道很好的面试题目，

- 题目短小，很快就能说清题意
- 有很多种解法。从简单到复杂的解法都有，梯度均匀。
- 不需要预先知道特殊领域知识。

这题有很多思路：

1. 按从大到小全排序，然后取第 `k` 个元素，时间复杂度 `O(nlogn)` ，空间复杂度 `O(1)`
2. 利用堆进行部分排序。维护一个大根堆，将数组元素全部压入堆，然后弹出 `k` 次，第 `k` 个就是答案。时间复杂度 `O(klogn)` ，空间复杂度 `O(n)`
3. 选择排序，第 `k` 次选择后即可得到第 `k` 大的数，时间复杂度 `O(nk)` ，空间复杂度 `O(1)`
4. 堆排序，维护一个 `k` 大小的小根堆，将数组中的每个元素与堆顶元素进行比较，如果比堆顶元素大，则删除堆顶元素并添加该元素，如果比堆顶元素小，则什么也不做，继续下一个元素。时间复杂度 `O(nlogk)` ，空间复杂度 `O(k)` 。
5. 利用快速排序中的partition思想，从数组中随机选择一个元素x，把数组划分为前后两部分 `sa` 和 `sb` ， `sa` 中的元素小于x， `sb` 中的元素大于或等于x。这时有两种情况：

   i. `sa` 的元素个数小于 `k` ，则递归求解 `sb` 中的第 `k-|sa|` 大的元素
   ii. `sa` 的元素个数大于或等于 `k` ，则递归求解 `sa` 中的第 `k` 大的元素

   时间复杂度 `O(n)` ，空间复杂度 `O(1)`

思路4和5比较高效，可以接受，其他思路太慢了，不采纳。

思路4 partition

```java
// Kth Largest Element in an Array
// Time complexity: O(nlogk), Space complexity: O(k)
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        final Queue<Integer> q = new PriorityQueue<>();

        for (int x : nums) {
            if (q.size() < k) {
                q.offer(x);
            } else {
                final int top = q.peek();
                if (x > top) {
                    q.poll();
                    q.offer(x);
                }
            }
        }
        return q.peek();
    }
}
```

思路**5** 小根堆

```java
// Kth Largest Element in an Array
// Time complexity: O(n), Space complexity: O(1)
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        quickSort(nums, 0, nums.length - 1);
        return nums[nums.length - k];
    }
    private static int findKthLargest(int[] nums, int begin, int end, int k) {
        if (begin + 1 == end && k == 1) return nums[0];

        final int pos = partition(nums, begin, end - 1);
        final int len = pos - begin;

        if (len == k) {
            return nums[pos-1];
        } else if (len < k) {
            return findKthLargest(nums, pos, end, k - len);
        } else {
            return findKthLargest(nums, begin, pos, k);
        }
    }
    private static void quickSort(int[] nums, int left, int right) {
        if (left < right) {
            final int pos = partition(nums, left, right);
            quickSort(nums, left, pos - 1);
            quickSort(nums, pos + 1, right);
        }
    }
    private static int partition(int[] nums, int i, int j) {
        final int pivot = nums[i];
        while (i < j) {
            while (i < j && nums[j] >= pivot) --j;
            nums[i] = nums[j];
            while(i < j && nums[i] <= pivot) ++i;
            nums[j] = nums[i];
        }
        nums[i] = pivot;
        return i;
    }
}
```

桶排序(Bucket Sort)的基本思路是：

1. 将待排序元素划分到不同的痛。先扫描一遍序列求出最大值 `maxV` 和最小值 `minV` ，设桶的个数为 `k` ，则把区间 `[minV, maxV]` 均匀划分成 `k` 个区间，每个区间就是一个桶。将序列中的元素分配到各自的桶。
2. 对每个桶内的元素进行排序。可以选择任意一种排序算法。
3. 将各个桶中的元素合并成一个大的有序序列。

假设数据是均匀分布的，则每个桶的元素平均个数为 `n/k` 。假设选择用快速排序对每个桶内的元素进行排序，那么每次排序的时间复杂度为 `O(n/klog(n/k))` 。总的时间复杂度为 `O(n)+O(m)O(n/klog(n/k)) = O(n+nlog(n/k)) = O(n+nlogn-nlogk` 。当 `k` 接近于 `n` 时，桶排序的时间复杂度就可以金斯认为是 `O(n)` 的。即桶越多，时间效率就越高，而桶越多，空间就越大。

First Missing Positive

描述

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

For example, Given `[1,2,0]` return `3`, and `[3,4,-1,1]` return `2`.

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

分析

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

代码

```
// First Missing Positive
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int firstMissingPositive(int[] nums) {
        bucket_sort(nums);

        for (int i = 0; i < nums.length; ++i)
            if (nums[i] != (i + 1))
                return i + 1;
        return nums.length + 1;
    }
    private static void bucket_sort(int[] A) {
        final int n = A.length;
        for (int i = 0; i < n; i++) {
            while (A[i] != i + 1) {
                if (A[i] < 1 || A[i] > n || A[i] == A[A[i] - 1])
                    break;
                // swap
                int tmp = A[i];
                A[i] = A[tmp - 1];
                A[tmp - 1] = tmp;
            }
        }
    }
}
```

相关题目

- Sort Colors

计数排序(Counting Sort)是一种O(n)的排序算法，其思路是开一个长度为 `maxValue-minValue+1` 的数组，然后

1. 分配。扫描一遍原始数组，以当前值- `minValue` 作为下标，将该下标的计数器增1。
2. 收集。扫描一遍计数器数组，按顺序把值收集起来。

举个例子， `nums=[2, 1, 3, 1, 5]` ,首先扫描一遍获取最小值和最大值， `maxValue=5` , `minValue=1` ，于是开一个长度为5的计数器数组 `counter` ，

1. 分配。统计每个元素出现的频率，得到 `counter=[2, 1, 1, 0, 1]` ，例如 `counter[0]` 表示值 `0+minValue=1` 出现了2次。
2. 收集。 `counter[0]=2` 表示 `1` 出现了两次，那就向原始数组写入两个1， `counter[1]=1` 表示 `2` 出现了1次，那就向原始数组写入一个2，依次类推，最终原始数组变为 `[1,1,2,3,5]` ，排序好了。

计数排序本质上是一种特殊的桶排序，当桶的个数最大的时候，就是计数排序。

H-Index

描述

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each."

For example, given `citations = [3, 0, 6, 1, 5]`, which means the researcher has 5 papers in total and each of them had received `3, 0, 6, 1, 5` citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, his h-index is 3.

Note: If there are several possible values for `h`, the maximum one is taken as the h-index.

分析

H-Index的含义是，如果一个人发表的所有论文中，有 `h` 篇论文分别被引用了至少 `h` 次，那么他的H-Index就是 `h` 。

思路一：先从大到小排序，然后从前往后扫描，如果当前文章数（即当前下标+1）等于值本身，则返回当前文章数作为 h-index；如果当前文章数大于值本身，则返回当前文章数-1作为H-Index, 因为当前文章的引用数小于当前文章数，不能算在内。时间复杂度$O(n \log n$，空间复杂度O(1)。

思路二：跟思路一类似，不过排序算法换成了计数排序。有一个小技巧，因为H-Index最大不可能超过论文综述，所以我们只需要开一个长度为 `n+1` 的数组，如果某篇论文的引用数超过了 `n` ，就将其当做 `n` 。

代码1 全排序

```java
// H-Index
// Time complexity: O(nlogn), Space complexity: O(1)
public class Solution {
    public int hIndex(int[] citations) {
        Arrays.sort(citations);
        reverse(citations);
        for (int i = 0; i < citations.length; ++i) {
            if (i + 1 == citations[i]) return i+1;
            if (i + 1 > citations[i]) return i;
        }
        return citations.length;
    }
    private static void reverse(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            final int tmp = nums[left];
            nums[left] = nums[right];
            nums[right] = tmp;
            ++left;
            --right;
        }
    }
}
```

代码2 计数排序

```java
// H-Index
// Time complexity: O(n), Space complexity: O(n)
public class Solution {
    public int hIndex(int[] citations) {
        final int n = citations.length + 1;
        final int[] histogram = new int[n+1];

        for (int x : citations) {
            ++histogram[x > n ? n : x];
        }

        int sum = 0; // current number of papers
        for (int i = n; i > 0; --i) {
            sum += histogram[i];
            if (sum >= i) {
                return i;
            }
        }
        return 0;
    }
}
```

相关题目

- H-Index II

基数排序是一种非比较排序算法，时间复杂度是 `O(n)` 。它的主要思路是，

1. 将所有待排序整数（注意，必须是非负整数）统一为位数相同的整数，位数较少的前面补零。一般用10进制，也可以用16进制甚至2进制。所以前提是能够找到最大值，得到最长的位数，设 `k` 进制下最长为位数为 `d` 。
2. 从最低位开始，依次进行一次稳定排序。这样从最低位一直到最高位排序完成以后，整个序列就变成了一个有序序列。

举个例子，有一个整数序列，0, 123, 45, 386, 106，下面是排序过程：

- 第一次排序，个位，00**0** 12**3** 04**5** 38**6** 10**6**，无任何变化
- 第二次排序，十位，0**0**0 1**0**6 1**2**3 0**4**5 3**8**6
- 第三次排序，百位，**0**00 **0**45 **1**06 **1**23 **3**86

最终结果，0, 45, 106, 123, 386, 排序完成。

为什么同一数位的排序子程序要用稳定排序？因为稳定排序能将上一次排序的成果保留下来。例如十位数的排序过程能保留个位数的排序成果，百位数的排序过程能保留十位数的排序成果。

能不能用2进制？能，可以把待排序序列中的每个整数都看成是01组成的二进制数值。那这样的话，岂不是任意一个非负整数序列都可以用基数排序算法？理论上是的，假设待排序序列中最大整数为2^64-1，则最大位数 `d=64` ，时间复杂度为 `O(64n)` 。可见任意一个非负整数序列都可以在线性时间内完成排序。

既然任意一个非负整数序列都可以在线性时间内完成排序，那么基于比较排序的算法有什么意义呢？基于比较的排序算法，时间复杂度是 `O(nlogn)` ，看起来比 `O(64n)` 慢，仔细一想，其实不是， `O(nlogn)` 只有当序列非常长，达到2^{64}个元素的时候，才会与 `O(64n)` 相等，因此，64这个常数系数太大了，大部分时候， `n` 远远小于2^{64}，基于比较的排序算法还是比 `O(64n)` 快的。

当使用2进制时， `k=2` 最小，位数 `d` 最大，时间复杂度 `O(nd)` 会变大，空间复杂度 `O(n+k)` 会变小。当用最大值作为基数时， `k=maxV` 最大， `d=1` 最小，此时时间复杂度 `O(nd)` 变小，但是空间复杂度 `O(n+k)` 会急剧增大，此时基数排序退化成了计数排序。

Maximum Gap

描述

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

分析

这道题最直接的解法是，先排序，得到有序数组，然后相邻元素相减，找出差最大的，时间复杂度 `O(n log n)` 。

然而本题要求 `O(n)` 时间，有没有 `O(n)` 的排序算法呢？桶排序、基数排序、计数排序。

解法**1** 桶排序

```java
// Maximum Gap
// Bucket Sort
// Time Complexity: O(n+k), Space Complexity: O(n+k)
public class Solution {
    public int maximumGap(int[] nums) {
        if (nums.length < 2) return 0;
        bucketSort(nums);

        int maxDiff = Integer.MIN_VALUE;
        for (int i = 1; i < nums.length; ++i) {
            maxDiff = Math.max(maxDiff, nums[i] - nums[i - 1]);
        }
        return maxDiff;
    }

    private static void bucketSort(int[] nums) {
        if (nums.length < 2) return;
        int minValue = Integer.MAX_VALUE;
        int maxValue = Integer.MIN_VALUE;

        for (int i : nums) {
            minValue = Math.min(minValue, i);
            maxValue = Math.max(maxValue, i);
        }

        final int bucketSize = (maxValue - minValue) / nums.length + 1;
        final int bucketCount = (maxValue - minValue) / bucketSize + 1;
        final ArrayList<Integer>[] buckets = new ArrayList[bucketCount];
        for (int i = 0; i < buckets.length; ++i) {
            buckets[i] = new ArrayList<>();
        }

        for (int x : nums) {
            final int index = (x - minValue) / bucketSize;
            buckets[index].add(x);
        }

        for (final ArrayList<Integer> list : buckets) {
            Collections.sort(list);
        }

        int i = 0;
        for (final ArrayList<Integer> list : buckets) {
            for (int x : list) {
                nums[i++] = x;
            }
        }
    }
}
```

解法2 基数排序

```java
// Maximum Gap
// Radix Sort
// Time Complexity: O(nd), Space Complexity: O(n+d)
public class Solution {
    public int maximumGap(int[] nums) {
        if (nums.length < 2) return 0;
        radixSort(nums);

        int maxDiff = Integer.MIN_VALUE;
        for (int i = 1; i < nums.length; ++i) {
            maxDiff = Math.max(maxDiff, nums[i] - nums[i - 1]);
        }
        return maxDiff;
    }
    private static void radixSort(int[] nums) {
        int minValue = Integer.MAX_VALUE;
        int maxValue = Integer.MIN_VALUE;

        for (int i : nums) {
            minValue = Math.min(minValue, i);
            maxValue = Math.max(maxValue, i);
        }

        final int D = Integer.toString(maxValue - minValue).length();
        final ArrayList<Integer>[] buckets = new ArrayList[10];
        for (int i = 0; i < buckets.length; ++i) {
            buckets[i] = new ArrayList<>();
        }

        for (int i = 0; i < D; ++i) {
            for (int x : nums) {
                final int index = getDigit(x - minValue, i);
                final ArrayList<Integer> bucket = buckets[index];
                bucket.add(x);
            }

            int index = 0;
            for (ArrayList<Integer> bucket : buckets) {
                for (int x : bucket) {
                    nums[index++] = x;
                }
                bucket.clear();
            }
        }
    }

    // get the i-th digit of n
    private static int getDigit(int n, int i) {
        for (int j = 0; j < i; ++j) {
            n /= 10;
        }
        return n % 10;
```

```
    }
}
```

解法3 计数排序

计数排序本质上是一种特殊的桶排序，当桶的个数最大的时候，就是计数排序。

本题用计数排序会MLE。

```
// Maximum Gap
// Counting Sort
// Time Complexity: O(n), Space Complexity: O(max-min)
public class Solution {
    public int maximumGap(int[] nums) {
        if (nums.length < 2) return 0;
        countingSort(nums);

        int maxDiff = Integer.MIN_VALUE;
        for (int i = 1; i < nums.length; ++i) {
            maxDiff = Math.max(maxDiff, nums[i] - nums[i - 1]);
        }
        return maxDiff;
    }
    private static void countingSort(int[] nums) {
        int minValue = Integer.MAX_VALUE;
        int maxValue = Integer.MIN_VALUE;

        for (int i : nums) {
            minValue = Math.min(minValue, i);
            maxValue = Math.max(maxValue, i);
        }

        final int[] buckets = new int[maxValue - minValue + 1];

        for (int i : nums) {
            buckets[i - minValue]++;
        }

        for (int i = 0, index = 0; i < buckets.length; ++i) {
            Arrays.fill(nums, index, index + buckets[i], i + minValue);
            index += buckets[i];
        }
    }
}
```

本节是一些排序相关的小技巧题目。

Largest Number

描述

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given `[3, 30, 34, 5, 9]`, the largest formed number is `9534330`.

Note: The result may be very large, so you need to return a string instead of an integer.

分析

这题可以先把每个整数变成字符串，得到一个字符串数组，然后把这个数组按特定规则排个序，顺序输出即可。

代码

```
// Largest Number
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public String largestNumber(int[] nums) {
        final String[] strings = new String[nums.length];
        for (int i = 0; i < nums.length; ++i) {
            strings[i] = String.valueOf(nums[i]);
        }
        Arrays.sort(strings, (String s1, String s2) -> {
            String leftRight = s1 + s2;
            String rightLeft = s2 + s1;
            return -leftRight.compareTo(rightLeft);
        });


        StringBuilder sb = new StringBuilder();
        for (final String s : strings) {
            sb.append(s);
        }

        while(sb.charAt(0)=='0' && sb.length()>1){
            sb.deleteCharAt(0);
        }

        return sb.toString();
    }
}
```

基数排序、桶排序和计数排序的区别

先比较时间复杂度和空间复杂度。

算法	时间复杂度	空间复杂度	适用场景
基数排序	O(nd)	O(n+k)	1.非负整数 2. `maxV` 和 `minV` 差距尽可能小
桶排序	O(n+k)	O(n+k)	元素尽可能均匀分布
计数排序	O(n+maxV-minV)	O(maxV-minV)	`maxV` 和 `minV` 差距尽可能小

其中，`d` 表示位数，`k` 在基数排序中表示 `k` 进制，在桶排序中表示桶的个数，`maxV` 和 `minV` 表示元素最大值和最小值。

首先，基数排序和计数排序都可以看作是桶排序。

- 计数排序本质上是一种特殊的桶排序，当桶的个数取最大(`maxV-minV+1`)的时候，就变成了计数排序。
- 基数排序也是一种桶排序。桶排序是按值区间划分桶，基数排序是按数位来划分；基数排序可以看做是多轮桶排序，每个数位上都进行一轮桶排序。
- 当用最大值作为基数时，基数排序就退化成了计数排序。

  当使用2进制时， `k=2` 最小，位数 `d` 最大，时间复杂度 `O(nd)` 会变大，空间复杂度 `O(n+k)` 会变小。当用最大值作为基数时， `k=maxV` 最大， `d=1` 最小，此时时间复杂度 `O(nd)` 变小，但是空间复杂度 `O(n+k)` 会急剧增大，此时基数排序退化成了计数排序。

查找

本章主要讲查找算法。

Search for a Range

描述

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

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

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

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

分析

已经排好了序，用二分查找。

重新实现 **lower_bound** 和 **upper_bound**

```
// Search for a Range
// 重新实现 lower_bound 和 upper_bound
// 时间复杂度O(logn)，空间复杂度O(1)
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        int lower = lower_bound(nums, 0, nums.length, target);
        int upper = upper_bound(nums, 0, nums.length, target);

        if (lower == nums.length || nums[lower] != target)
            return new int[]{-1, -1};
        else
            return new int[]{lower, upper-1};
    }

    int lower_bound (int[] A, int first, int last, int target) {
        while (first != last) {
            int mid = first + (last - first) / 2;
            if (target > A[mid]) first = ++mid;
            else                 last = mid;
        }

        return first;
    }

    int upper_bound (int[] A, int first, int last, int target) {
        while (first != last) {
            int mid = first + (last - first) / 2;
            if (target >= A[mid]) first = ++mid;  // 与 lower_bound 仅此不同
            else                  last = mid;
        }

        return first;
    }
}
```

相关题目

- Search Insert Position

Search Insert Position

描述

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

You may assume no duplicates in the array.

Here are few examples.

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

分析

即 `std::lower_bound()` 。

代码

```
// Search Insert Position
// 重新实现 lower_bound
// 时间复杂度O(logn)，空间复杂度O(1)
public class Solution {
    public int searchInsert(int[] nums, int target) {
        return lower_bound(nums, 0, nums.length, target);
    }

    int lower_bound (int[] A, int first, int last, int target) {
        while (first != last) {
            int mid = first + (last - first) / 2;
            if (target > A[mid]) first = ++mid;
            else                 last = mid;
        }

        return first;
    }
}
```

相关题目

- Search for a Range

Search in Rotated Sorted Array

描述

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

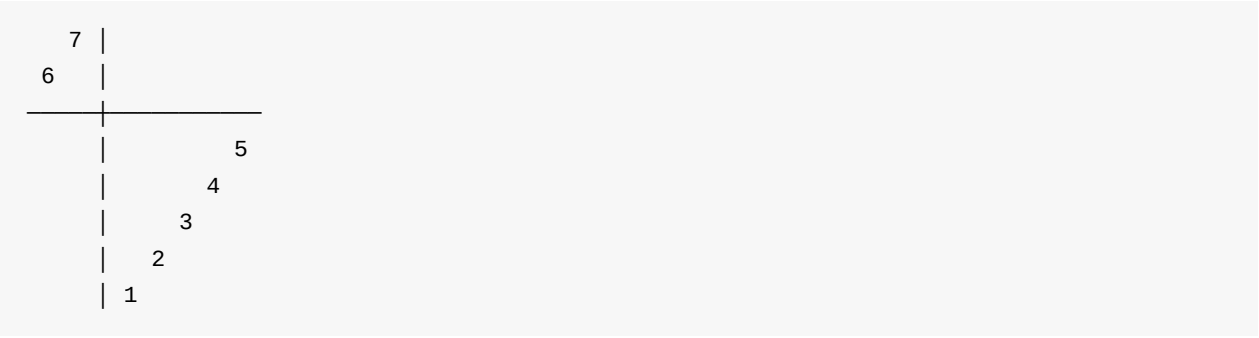
(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

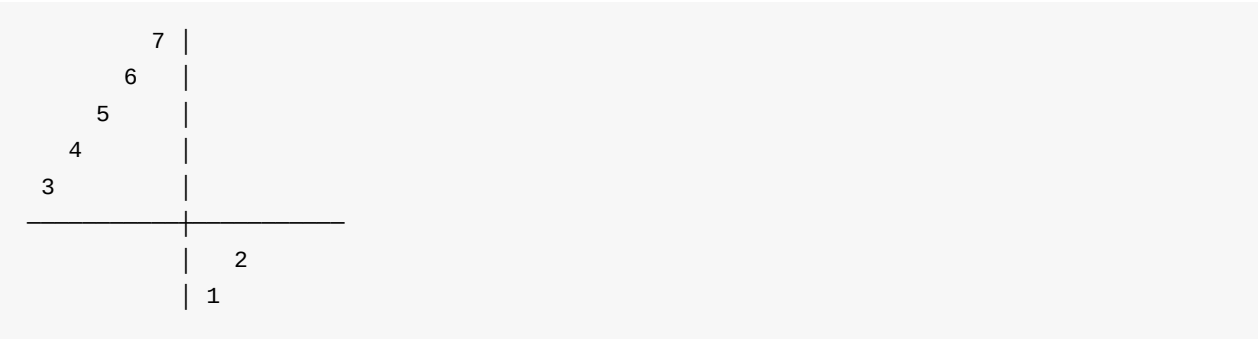
You are given a target value to search. If found in the array return its index, otherwise return `-1`.

You may assume no duplicate exists in the array.

分析

一个有序数组被循环右移，只可能有一下两种情况：





本题依旧可以用二分查找，难度主要在于左右边界的确定。仔细观察上面两幅图，我们可以得出如下结论：

如果 `A[left] <= A[mid]`,那么 `[left,mid]` 一定为单调递增序列。

代码

```java
// Search in Rotated Sorted Array
// Time Complexity: O(log n)，Space Complexity: O(1)
public class Solution {
    public int search(int[] nums, int target) {
        int first = 0, last = nums.length;
        while (first != last) {
            final int mid = first  + (last - first) / 2;
            if (nums[mid] == target)
                return mid;
            if (nums[first] <= nums[mid]) {
                if (nums[first] <= target && target < nums[mid])
                    last = mid;
                else
                    first = mid + 1;
            } else {
                if (nums[mid] < target && target <= nums[last-1])
                    first = mid + 1;
                else
                    last = mid;
            }
        }
        return -1;
    }
};
```

相关题目

- Search in Rotated Sorted Array II
- Find Minimum in Rotated Sorted Array
- Find Minimum in Rotated Sorted Array II

Search in Rotated Sorted Array II

描述

Follow up for "Search in Rotated Sorted Array": What if **duplicates** are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

分析

允许重复元素，则上一题中如果 `A[left] <= A[mid]` ,那么 `[left,mid]` 为递增序列的假设就不能成立了，比如 `[1,3,1,1,1]` 。

既然 `A[left] <= A[mid]` 不能确定递增，那就把它拆分成两个条件：

- 若 `A[left] < A[mid]` ，则区间 `[left,mid]` 一定递增
- 若 `A[left] == A[mid]` 确定不了，那就 `left++` ，往下看一步即可。

代码

```
// Search in Rotated Sorted Array II
// Time Complexity: O(n)，Space Complexity: O(1)
public class Solution {
    public boolean search(int[] nums, int target) {
        int first = 0, last = nums.length;
        while (first != last) {
            final int mid = first  + (last - first) / 2;
            if (nums[mid] == target)
                return true;
            if (nums[first] < nums[mid]) {
                if (nums[first] <= target && target < nums[mid])
                    last = mid;
                else
                    first = mid + 1;
            } else if (nums[first] > nums[mid]) {
                if (nums[mid] < target && target <= nums[last-1])
                    first = mid + 1;
                else
                    last = mid;
            } else
                //skip duplicate one
                first++;
        }
        return false;
    }
};
```

相关题目

- Search in Rotated Sorted Array
- Find Minimum in Rotated Sorted Array
- Find Minimum in Rotated Sorted Array II

Search a 2D Matrix

描述

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

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example, Consider the following matrix:

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

Given `target = 3`, return true.

分析

二分查找。

代码

```java
// Search a 2D Matrix
// 时间复杂度O(logn)，空间复杂度O(1)
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix.length == 0) return false;
        final int  m = matrix.length;
        final int n = matrix[0].length;

        int first = 0;
        int last = m * n;

        while (first < last) {
            int mid = first + (last - first) / 2;
            int value = matrix[mid / n][mid % n];

            if (value == target)
                return true;
            else if (value < target)
                first = mid + 1;
            else
                last = mid;
        }

        return false;
    }
}
```

相关题目

- Search a 2D Matrix II

Search a 2D Matrix II

描述

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

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

Given target = `5`, return `true`.

Given target = `20`, return `false`.

分析

从右上角开始, 比较 `target` 和 `matrix[i][j]` 的值。如果小于 `target`, 则该行不可能有此数, 所以 `i++`; 如果大于 `target`, 则该列不可能有此数, 所以 `j--` 。遇到边界则表明该矩阵不含 `target`.

代码

```java
// Search a 2D Matrix II
// 时间复杂度O(m + n)，空间复杂度O(1)
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix.length==0 || matrix[0].length==0) return false;

        int i = 0;
        int j = matrix[0].length-1;
        while(i < matrix.length && j >= 0) {
            final int x = matrix[i][j];
            if(target == x) {
                return true;
            } else if (x < target) {
                ++i;
            } else {
                --j;
            }
        }
        return false;
    }
}
```

相关题目

- Search a 2D Matrix

Find Minimum in Rotated Sorted Array

描述

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

Find the minimum element.

You may assume no duplicate exists in the array.

分析

从左向右扫描，扫描到的第一个逆序的位置，肯定是原始数组中第一个元素，时间复杂度 `O(n)` 。

不过本题依旧可以用二分查找，最关键的是要判断那个“断层”是在左边还是右边。

- 若 `A[mid] < A[right]` ，则区间 `[mid,right]` 一定递增，断层一定在左边
- 若 `A[mid] > A[right]` ，则区间 `[left,mid]` 一定递增，断层一定在右边
- `nums[mid] == nums[right]` ，这种情况不可能发生，因为数组是严格单调递增的，不存在重复元素

代码

```
// Find Minimum in Rotated Sorted Array
// 时间复杂度O(logn)，空间复杂度O(1)
public class Solution {
    public int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < nums[right]) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return nums[left];
    }
}
```

相关题目

- Search in Rotated Sorted Array
- Search in Rotated Sorted Array II
- Find Minimum in Rotated Sorted Array II

Find Minimum in Rotated Sorted Array II

描述

Follow up for "Find Minimum in Rotated Sorted Array":

What if duplicates are allowed?

分析

同 Find Minimum in Rotated Sorted Array 类似，要判断“断层”在左边还是右边。

- 若 `A[mid] < A[right]` ，则区间 `[mid,right]` 一定递增，断层一定在左边
- 若 `A[mid] > A[right]` ，则区间 `[left,mid]` 一定递增，断层一定在右边
- 若 `A[mid] == A[right]` 确定不了，这个时候，断层既可能在左边，也可能在右边，所以我们不能扔掉一半，不过这时，我们可以 `--right` 扔掉一个

本题还有另一种思路，

- 若 `A[left] < A[mid]` ，则区间 `[left,mid]` 一定递增，断层一定在右边
- 若 `A[left] > A[mid]` ，则区间 `[mid,right]` 一定递增，断层一定在左边
- 若 `A[left] == A[mid]` 确定不了，这个时候，断层既可能在左边，也可能在右边，所以我们不能扔掉一半，不过这时，我们可以 `++left` 扔掉一

注意，第三种情况，我们认为可以 `++left` 扔掉一个，这个做法是不对的，因为数组被分成两段后，两段分别是递增的， `left` 这个元素有可能是全局最小值，不能贸然扔掉。而在前一种思路中， `end` 可以扔掉，因为 `end` 在右边，它的左边必然有小于或等于它的元素，所以可以放心 `--end` 。

代码

```java
// Find Minimum in Rotated Sorted Array II
// 时间复杂度O(logn)，最坏 O(n)，空间复杂度O(1)
public class Solution {
    public int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < nums[right]) {
                right = mid;
            } else if (nums[mid] > nums[right]) {
                left = mid + 1;
            } else {
                --right;
            }
        }
        return nums[left];
    }
}
```

相关题目

- Search in Rotated Sorted Array
- Search in Rotated Sorted Array II
- Find Minimum in Rotated Sorted Array

Median of Two Sorted Arrays

描述

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))`.

分析

这是一道非常经典的题。这题更通用的形式是，给定两个已经排序好的数组，找到两者所有元素中第 `k` 大的元素。

`O(m+n)` 的解法比较直观，直接merge两个数组，然后求第 `k` 大的元素。

不过我们仅仅需要第 `k` 大的元素，是不需要“排序”这么昂贵的操作的。可以用一个计数器，记录当前已经找到第 `m` 大的元素了。同时我们使用两个指针 `pA` 和 `pB`，分别指向A和B数组的第一个元素，使用类似于merge sort的原理，如果数组A当前元素小，那么 `pA++`，同时 `m++`；如果数组B当前元素小，那么 `pB++`，同时 `m++`。最终当 `m` 等于 `k` 的时候，就得到了我们的答案，`O(k)` 时间，`O(1)` 空间。但是，当 `k` 很接近 `m+n` 的时候，这个方法还是 `O(m+n)` 的。

有没有更好的方案呢？我们可以考虑从 `k` 入手。如果我们每次都能够删除一个一定在第 `k` 大元素之前的元素，那么我们需要进行 `k` 次。但是如果每次我们都删除一半呢？由于A和B都是有序的，我们应该充分利用这里面的信息，类似于二分查找，也是充分利用了“有序”。

假设A和B的元素个数都大于 `k/2`，我们将A的第 `k/2` 个元素（即 `A[k/2-1]`）和B的第 `k/2` 个元素（即 `B[k/2-1]`）进行比较，有以下三种情况（为了简化这里先假设 `k` 为偶数，所得到的结论对于 `k` 是奇数也是成立的）：

- `A[k/2-1] == B[k/2-1]`
- `A[k/2-1] > B[k/2-1]`
- `A[k/2-1] < B[k/2-1]`

如果 `A[k/2-1] < B[k/2-1]`，意味着 `A[0]` 到 `A[k/2-1]` 的肯定在$A \cup B$的top k元素的范围内，换句话说，`A[k/2-1]` 不可能大于$A \cup B$的第 `k` 大元素。留给读者证明。

因此，我们可以放心的删除A数组的这 `k/2` 个元素。同理，当 `A[k/2-1] > B[k/2-1]` 时，可以删除B数组的 `k/2` 个元素。

当 `A[k/2-1] == B[k/2-1]` 时，说明找到了第 `k` 大的元素，直接返回 `A[k/2-1]` 或 `B[k/2-1]` 即可。

因此，我们可以写一个递归函数。那么函数什么时候应该终止呢？

- 当A或B是空时，直接返回 `B[k-1]` 或 `A[k-1]`；
- 当 `k=1` 是，返回 `min(A[0], B[0])`；
- 当 `A[k/2-1] == B[k/2-1]` 时，返回 `A[k/2-1]` 或 `B[k/2-1]`

代码

```java
// Median of Two Sorted Arrays
// Time Complexity: O(log(m+n))，Space Complexity: O(log(m+n))
public class Solution {
    public double findMedianSortedArrays(final int[] A, final int[] B) {
        int total = A.length + B.length;
        if (total %2 == 1)
            return findKth(A, 0, B, 0, total / 2 + 1);
        else
            return (findKth(A, 0, B, 0, total / 2)
                    + findKth(A, 0, B, 0, total / 2 + 1)) / 2.0;
    }

    private static int findKth(final int[] A, int ai, final int[] B, int bi, int k) {
        //always assume that A is shorter than B
        if (A.length - ai > B.length - bi) {
            return findKth(B, bi, A, ai, k);
        }
        if (A.length - ai == 0) return B[bi + k - 1];
        if (k == 1) return Math.min(A[ai], B[bi]);

        //divide k into two parts
        int k1 = Math.min(k / 2, A.length - ai), k2 = k - k1;
        if (A[ai + k1 - 1] < B[bi + k2 - 1])
            return findKth(A, ai + k1, B, bi, k - k1);
        else if (A[ai + k1 - 1] > B[bi + k2 - 1])
            return findKth(A, ai, B, bi + k2, k - k2);
        else
            return A[ai + k1 - 1];
    }
};
```

H-Index II

描述

Follow up for H-Index: What if the citations array is sorted in ascending order? Could you optimize your algorithm?

分析

设数组长度为 `n` ，那么 `n-i` 就是引用次数大于等于 `nums[i]` 的文章数。如果 `nums[i]<n-i` ，说明 `i` 是有效的H-Index, 如果一个数是H-Index，那么最大的H-Index一定在它后面（因为是升序的），根据这点就可以进行二分搜索了。

代码

```
// H-Index II
// Time complexity: O(logn), Space complexity: O(1)
public class Solution {
    public int hIndex(int[] citations) {
        final int n = citations.length;
        int begin = 0;
        int end = citations.length;

        while (begin < end) {
            final int mid = begin + (end - begin) / 2;
            if (citations[mid] < n - mid) {
                begin = mid + 1;
            } else {
                end = mid;
            }
        }
        return n - begin;
    }
}
```

本章的题目都能用暴力枚举法解决。

Subsets

描述

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],
  []
]
```

递归

增量构造法

每个元素，都有两种选择，选或者不选。

```java
// Subsets
// 增量构造法，深搜，时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        Arrays.sort(nums); // 输出要求有序
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> path = new ArrayList<>();
        subsets(nums, path, 0, result);
        return result;
    }

    private static void subsets(int[] nums, List<Integer> path, int step,
                                List<List<Integer>> result) {
        if (step == nums.length) {
            result.add(new ArrayList<Integer>(path));
            return;
        }
        // 不选nums[step]
        subsets(nums, path, step + 1, result);
        // 选nums[step]
        path.add(nums[step]);
        subsets(nums, path, step + 1, result);
        path.remove(path.size() - 1);
    }
}
```

位向量法

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

```java
// Subsets
// 位向量法，深搜，时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        Arrays.sort(nums);  // 输出要求有序

        List<List<Integer>> result = new ArrayList<>();
        boolean[] selected = new boolean[nums.length];
        subsets(nums, selected, 0, result);
        return result;
    }

    private static void subsets(int[] nums, boolean[] selected, int step,
                        List<List<Integer>> result) {
        if (step == nums.length) {
            ArrayList<Integer> subset = new ArrayList<>();
            for (int i = 0; i < nums.length; i++) {
                if (selected[i]) subset.add(nums[i]);
            }
            result.add(subset);
            return;
        }
        // 不选S[step]
        selected[step] = false;
        subsets(nums, selected, step + 1, result);
        // 选S[step]
        selected[step] = true;
        subsets(nums, selected, step + 1, result);
    }
}
```

迭代

增量构造法

```
// Subsets
// 迭代版，时间复杂度O(2^n)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        Arrays.sort(nums); // 输出要求有序
        List<List<Integer>> result = new ArrayList<>();
        result.add(new ArrayList<>()); // empty subset
        for (int elem : nums) {
            final int n = result.size();
            for (int i = 0; i < n; ++i) { // copy itself
                result.add(new ArrayList<>(result.get(i)));
            }
            for (int i = n; i < result.size(); ++i) {
                result.get(i).add(elem);
            }
        }
        return result;
    }
}
```

二进制法

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

这种方法最巧妙。因为它不仅能生成子集，还能方便的表示集合的并、交、差等集合运算。设两个集合的位向量分别为B_1和B_2，则$B_1 \cup B_2, B_1 \cap B_2, B_1 \triangle B_2$分别对应集合的并、交、对称差。

二进制法，也可以看做是位向量法，只不过更加优化。

```
// Subsets
// 二进制法，时间复杂度O(2^n)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        Arrays.sort(nums); // 输出要求有序
        List<List<Integer>> result = new ArrayList<>();
        final int n = nums.length;
        ArrayList<Integer> v = new ArrayList<>();

        for (int i = 0; i < 1 << n; i++) {
            for (int j = 0; j < n; j++) {
                if ((i & 1 << j) > 0) v.add(nums[j]);
            }
            result.add(new ArrayList<>(v));
            v.clear();
        }
        return result;
    }
}
```

相关题目

- Subsets II

Subsets II

描述

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],
  []
]
```

分析

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

递归

增量构造法

```java
// Subsets II
// 增量构造法，版本1，时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);  // 必须排序

        List<List<Integer>> result = new ArrayList<>();
        List<Integer> path = new ArrayList<>();

        dfs(nums, 0, path, result);
        return result;
    }

    private static void dfs(int[] nums, int start, List<Integer> path,
                            List<List<Integer>> result) {
        result.add(new ArrayList<Integer>(path));

        for (int i = start; i < nums.length; i++) {
            if (i != start && nums[i] == nums[i-1]) continue;
            path.add(nums[i]);
            dfs(nums, i + 1, path, result);
            path.remove(path.size() - 1);
        }
    }
}
```

```java
// Subsets II
// 增量构造法，版本2，时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);  // 必须排序
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> path = new ArrayList<>(); // 中间结果

        // 记录每个元素的出现次数
        HashMap<Integer, Integer> counterMap = new HashMap<>();
        for (int i : nums) {
            counterMap.put(i, counterMap.getOrDefault(i, 0) + 1);
        }
        // 将HashMap里的pair拷贝到一个数组里
        Pair[] counters = new Pair[counterMap.size()];
        int i = 0;
        for (Map.Entry<Integer, Integer> entry : counterMap.entrySet()) {
            counters[i++] = new Pair(entry.getKey(), entry.getValue());
        }
        Arrays.sort(counters);

        dfs(counters, 0, path, result);
        return result;
    }
```

```
    private static void dfs(Pair[] counters, int step, List<Integer> path,
                            List<List<Integer>> result) {
        if (step == counters.length) {
            result.add(new ArrayList<>(path));
            return;
        }

        for (int i = 0; i <= counters[step].value; i++) {
            for (int j = 0; j < i; ++j) {
                path.add(counters[step].key);
            }
            dfs(counters, step + 1, path, result);
            for (int j = 0; j < i; ++j) {
                path.remove(path.size() - 1);
            }
        }
    }

    static class Pair implements Comparable<Pair> {
        int key;
        int value;
        public Pair(int key, int value) {
            this.key = key;
            this.value = value;
        }
        @Override
        public int compareTo(Pair o) {
            if (this.key < o.key) return -1;
            else if (this.key > o.key) return 1;
            else {
                return this.value - o.value;
            }
        }
    }
}
```

位向量法

```
// Subsets II
// 位向量法，时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);  // 必须排序
        List<List<Integer>> result = new ArrayList<>();
        // 记录每个元素的出现次数
        HashMap<Integer, Integer> counterMap = new HashMap<>();
        for (int i : nums) {
            counterMap.put(i, counterMap.getOrDefault(i, 0) + 1);
        }
        // 将HashMap里的pair拷贝到一个数组里
        Pair[] counters = new Pair[counterMap.size()];
        int i = 0;
```

```
        for (Map.Entry<Integer, Integer> entry : counterMap.entrySet()) {
            counters[i++] = new Pair(entry.getKey(), entry.getValue());
        }
        Arrays.sort(counters);

        // 每个元素选择了多少个
        HashMap<Integer, Integer> selected = new HashMap<>();
        for (Pair p : counters) {
            selected.put(p.key, 0 );
        }

        dfs(nums, counters, selected, 0, result);
        return result;
    }

    private static void dfs(int[] nums, Pair[] counters, HashMap<Integer, Integer> sele
cted,
                            int step, List<List<Integer>> result) {
        if (step == counters.length) {
            ArrayList<Integer> subset = new ArrayList<>();
            for (Pair p : counters) {
                for (int i = 0; i < selected.get(p.key); ++i) {
                    subset.add(p.key);
                }
            }
            result.add(subset);
            return;
        }

        for (int i = 0; i <= counters[step].value; i++) {
            selected.put(counters[step].key, i);
            dfs(nums, counters, selected, step + 1, result);
        }
    }
    static class Pair implements Comparable<Pair> {
        int key;
        int value;
        public Pair(int key, int value) {
            this.key = key;
            this.value = value;
        }

        @Override
        public int compareTo(Pair o) {
            if (this.key < o.key) return -1;
            else if (this.key > o.key) return 1;
            else {
                return this.value - o.value;
            }
        }
    }
}
```

迭代

增量构造法

```java
// Subsets II
// 增量构造法
// 时间复杂度O(2^n)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);  // 必须排序
        List<List<Integer>> result = new ArrayList<>();
        result.add(new ArrayList<Integer>());

        int previous_size = 0;
        for (int i = 0; i < nums.length; ++i) {
            final int size = result.size();
            for (int j = 0; j < size; ++j) {
                if (i == 0 || nums[i] != nums[i-1] || j >= previous_size) {
                    result.add(new ArrayList<>(result.get(j)));
                    result.get(result.size() - 1).add(nums[i]);
                }
            }
            previous_size = size;
        }
        return result;
    }
}
```

二进制法

```
// Subsets II
// 二进制法，时间复杂度O(2^n)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);  // 必须排序
        // 用 set 去重，不能用 unordered_set，因为输出要求有序
        LinkedHashSet<ArrayList<Integer>> result = new LinkedHashSet<>();
        final int n = nums.length;
        ArrayList<Integer> v = new ArrayList<>();

        for (int i = 0; i < 1 << n; ++i) {
            for (int j = 0; j < n; ++j) {
                if ((i & 1 << j) > 0)
                    v.add(nums[j]);
            }
            result.add(new ArrayList<>(v));
            v.clear();
        }
        List<List<Integer>> realResult = new ArrayList<>();
        for (ArrayList<Integer> list : result) {
            realResult.add(list);
        }
        return realResult;
    }
}
```

相关题目

- Subsets

Permutations

描述

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]`.

next_permutation()

函数 `next_permutation()` 的具体实现见这节 Next Permutation。

```
// Permutations
// 重新实现 next_permutation()
// 时间复杂度O(n!)，空间复杂度O(1)
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);

        do {
            ArrayList<Integer> one = new ArrayList<>();
            for (int i : nums) {
                one.add(i);
            }
            result.add(one);
            // 调用的是 2.1.12 节的 next_permutation()
            // 而不是 std::next_permutation()
        } while(nextPermutation(nums, 0, nums.length));
        return result;
    }
    // 代码来自 2.1.12 节的 next_permutation()
    private static boolean nextPermutation(int[] nums, int begin, int end) {
        // From right to left, find the first digit(partitionNumber)
        // which violates the increase trend
        int p = end - 2;
        while (p > -1 && nums[p] >= nums[p + 1]) --p;

        // If not found, which means current sequence is already the largest
        // permutation, then rearrange to the first permutation and return false
        if(p == -1) {
            reverse(nums, begin, end);
            return false;
        }

        // From right to left, find the first digit which is greater
        // than the partition number, call it changeNumber
        int c = end - 1;
        while (c > 0 && nums[c] <= nums[p]) --c;
```

```
        // Swap the partitionNumber and changeNumber
        swap(nums, p, c);
        // Reverse all the digits on the right of partitionNumber
        reverse(nums, p+1, end);
        return true;
    }
    private static void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
    private static void reverse(int[] nums, int begin, int end) {
        end--;
        while (begin < end) {
            swap(nums, begin++, end--);
        }
    }
}
```

递归

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

扩展节点，每次从左到右，选一个没有出现过的元素。

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

代码

```java
// Permutations
// 深搜，增量构造法
// 时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        Arrays.sort(nums);

        List<List<Integer>> result = new ArrayList<>();
        List<Integer> path = new ArrayList<>(); // 中间结果

        dfs(nums, path, result);
        return result;
    }
    private static void dfs(int[] nums, List<Integer> path,
                            List<List<Integer>> result) {
        if (path.size() == nums.length) {  // 收敛条件
            result.add(new ArrayList<Integer>(path));
            return;
        }

        // 扩展状态
        for (int i : nums) {
            // 查找 i 是否在path 中出现过
            int pos = path.indexOf(i);

            if (pos == -1) {
                path.add(i);
                dfs(nums, path, result);
                path.remove(path.size() - 1);
            }
        }
    }
}
```

相关题目

- Next Permutation
- Permutation Sequence
- Permutations II
- Combinations

Permutations II

描述

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]`.

next_permutation()

直接使用 `std::next_permutation()`，代码与上一题相同。

重新实现next_permutation()

重新实现 `std::next_permutation()`，代码与上一题相同。

递归

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

扩展节点，每次从小到大，选一个没有被用光的元素，直到所有元素被用光。

本题不需要判重，因为状态装换图是一颗有层次的树。

```
// Permutations II
// 深搜，时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        Arrays.sort(nums);  // 必须排序
        List<List<Integer>> result = new ArrayList<>(); // 最终结果
        List<Integer> path = new ArrayList<>(); // 中间结果

        // 记录每个元素的出现次数
        HashMap<Integer, Integer> counterMap = new HashMap<>();
        for (int i : nums) {
            counterMap.put(i, counterMap.getOrDefault(i, 0) + 1);
        }
        // 将HashMap里的pair拷贝到一个数组里
        Pair[] counters = new Pair[counterMap.size()];
        int i = 0;
        for (Map.Entry<Integer, Integer> entry : counterMap.entrySet()) {
            counters[i++] = new Pair(entry.getKey(), entry.getValue());
        }
        Arrays.sort(counters);

        // 每个元素选择了多少个
        HashMap<Integer, Integer> selected = new HashMap<>();
        for (Pair p : counters) {
```

```
            selected.put(p.key, 0 );
        }

        n = nums.length;
        permute(counters, selected, path, result);
        return result;
    }

    private int n;

    void permute(Pair[] counters, HashMap<Integer,Integer> selected,
                 List<Integer> path, List<List<Integer>> result) {
        if (n == path.size()) {  // 收敛条件
            result.add(new ArrayList<>(path));
        }

        // 扩展状态
        for (Pair counter : counters) {
            if (selected.get(counter.key) < counter.value) {
                path.add(counter.key);
                selected.put(counter.key, selected.get(counter.key) + 1);
                permute(counters, selected, path, result);
                path.remove(path.size() - 1);
                selected.put(counter.key, selected.get(counter.key) - 1);
            }
        }
    }
    static class Pair implements Comparable<Pair> {
        int key;
        int value;
        public Pair(int key, int value) {
            this.key = key;
            this.value = value;
        }

        @Override
        public int compareTo(Pair o) {
            if (this.key < o.key) return -1;
            else if (this.key > o.key) return 1;
            else {
                return this.value - o.value;
            }
        }
    }
}
```

相关题目

- Next Permutation
- Permutation Sequence
- Permutations

- Combinations

Combinations

描述

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

For example, If `n = 4` and `k = 2`, a solution is:

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

递归

```
// Combinations
// 深搜，递归
// 时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> path = new ArrayList<>();
        dfs(n, k, 1, 0, path, result);
        return result;
    }
    // start，开始的数, cur，已经选择的数目
    private static void dfs(int n, int k, int start, int cur,
                            List<Integer> path, List<List<Integer>> result) {
        if (cur == k) {
            result.add(new ArrayList<>(path));
        }
        for (int i = start; i <= n; ++i) {
            path.add(i);
            dfs(n, k, i + 1, cur + 1, path, result);
            path.remove(path.size() - 1);
        }
    }
}
```

相关题目

- Next Permutation
- Permutation Sequence
- Permutations

- Permutations II

Letter Combinations of a Phone Number

描述

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: Phone Keyboard

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.

分析

无

递归

```java
// Letter Combinations of a Phone Number
// 时间复杂度O(3^n)，空间复杂度O(n)
public class Solution {
    private static final String[] keyboard =
            new String[]{ " ", "", "abc", "def", // '0','1','2',...
            "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    public List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<>();
        if (digits.isEmpty()) return result;
        dfs(digits, 0, "", result);
        return result;
    }

    private static void dfs(String digits, int cur, String path,
                            List<String> result) {
        if (cur == digits.length()) {
            result.add(path);
            return;
        }
        final String str = keyboard[digits.charAt(cur) - '0'];

        for (char c : keyboard[digits.charAt(cur) - '0'].toCharArray()) {
            dfs(digits, cur + 1, path + c, result);
        }
    }
}
```

迭代

```java
// Letter Combinations of a Phone Number
// 时间复杂度O(3^n)，空间复杂度O(1)
public class Solution {
    private static final String[] keyboard =
            new String[]{ " ", "", "abc", "def", // '0','1','2',...
                    "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    public List<String> letterCombinations(String digits) {
        if (digits.isEmpty()) return new ArrayList<>();
        List<String> result = new ArrayList<>();
        result.add("");
        for (char d : digits.toCharArray()) {
            final int n = result.size();
            final int m = keyboard[d - '0'].length();

            // resize to n * m
            for (int i = 1; i < m; ++i) {
                for (int j = 0; j < n; ++j) {
                    result.add(result.get(j));
                }
            }

            for (int i = 0; i < result.size(); ++i) {
                result.set(i, result.get(i) + keyboard[d - '0'].charAt(i/n));
            }
        }
        return result;
    }
}
```

广度优先搜索

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

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

Word Ladder

描述

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.

分析

求最短路径，用广搜。

单队列

```
// Word Ladder
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int ladderLength(String beginWord, String endWord, Set<String> wordList) {
        Queue<State> q = new LinkedList<>();
        HashSet<State> visited = new HashSet<>(); // 判重

        final Function<State, Boolean> stateIsValid = (State s) ->
                wordList.contains(s.word) || s.word.equals(endWord);
        final Function<State, Boolean> stateIsTarget = (State s) ->
                s.word.equals(endWord);

        final Function<State, HashSet<State> > stateExtend = (State s) -> {
            HashSet<State> result = new HashSet<>();

            char[] array = s.word.toCharArray();
            for (int i = 0; i < array.length; ++i) {
                final char old = array[i];
                for (char c = 'a'; c <= 'z'; c++) {
```

```
                    // 防止同字母替换
                    if (c == array[i]) continue;

                    array[i] = c;
                    State newState = new State(new String(array), s.level+1);

                    if (stateIsValid.apply(newState) &&
                            !visited.contains(newState)) {
                        result.add(newState);
                    }
                    array[i] = old; // 恢复该单词
                }
            }

            return result;
        };

        State startState = new State(beginWord, 0);
        q.offer(startState);
        visited.add(startState);
        while (!q.isEmpty()) {
            State state = q.poll();

            if (stateIsTarget.apply(state)) {
                return state.level + 1;
            }


            HashSet<State> newStates = stateExtend.apply(state);
            for (State newState : newStates) {
                q.offer(newState);
                visited.add(newState);
            }
        }
        return 0;
    }

    static class State {
        String word;
        int level;

        public State(String word, int level) {
            this.word = word;
            this.level = level;
        }

        @Override
        public int hashCode() {
            return word.hashCode();
        }

        @Override
        public boolean equals(Object other) {
```

```
            if (this == other) return true;
            if (this.hashCode() != other.hashCode()) return false;
            if (!(other instanceof State)) return false;

            return this.word.equals(((State) other).word);
        }
    }
}
```

双队列

```
// Word Ladder
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int ladderLength(String beginWord, String endWord, Set<String> wordList) {
        Queue<String> current = new LinkedList<>(); // 当前层
        Queue<String> next = new LinkedList<>();    // 下一层
        HashSet<String> visited = new HashSet<>();  // 判重

        int level = -1;  // 层次

        final Function<String, Boolean> stateIsValid = (String s) ->
                wordList.contains(s) || s.equals(endWord);
        final Function<String, Boolean> stateIsTarget = (String s) ->
                s.equals(endWord);

        final Function<String, HashSet<String> > stateExtend = (String s) -> {
            HashSet<String> result = new HashSet<>();

            char[] array = s.toCharArray();
            for (int i = 0; i < array.length; ++i) {
                final char old = array[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替换
                    if (c == array[i]) continue;

                    array[i] = c;
                    String newState = new String(array);

                    if (stateIsValid.apply(newState) &&
                            !visited.contains(newState)) {
                        result.add(newState);
                    }
                    array[i] = old; // 恢复该单词
                }
            }

            return result;
        };

        current.offer(beginWord);
```

```
        visited.add(beginWord);
        while (!current.isEmpty()) {
            ++level;
            while (!current.isEmpty()) {
                // 千万不能用 const auto&，pop() 会删除元素，
                // 引用就变成了悬空引用
                String state = current.poll();

                if (stateIsTarget.apply(state)) {
                    return level + 1;
                }

                HashSet<String> newStates = stateExtend.apply(state);
                for (String newState : newStates) {
                    next.offer(newState);
                    visited.add(newState);
                }
            }
            // swap
            Queue<String> tmp = current;
            current = next;
            next = tmp;
        }
        return 0;
    }
}
```

相关题目

- Word Ladder II

Word Ladder II

描述

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.

分析

跟 Word Ladder比，这题是求路径本身，不是路径长度，也是BFS，略微麻烦点。

求一条路径和求所有路径有很大的不同，求一条路径，每个状态节点只需要记录一个前驱即可；求所有路径时，有的状态节点可能有多个父节点，即要记录多个前驱。

如果当前路径长度已经超过当前最短路径长度，可以中止对该路径的处理，因为我们要找的是最短路径。

单队列

```
// Word Ladder II
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord,
                                          Set<String> wordList) {
        Queue<String> q = new LinkedList<>();
        HashMap<String, Integer> visited = new HashMap<>(); // 判重
        HashMap<String, ArrayList<String>> father = new HashMap<>(); // DAG
```

```
        final Function<String, Boolean> stateIsValid = (String s) ->
                wordList.contains(s) || s.equals(endWord);
        final Function<String, Boolean> stateIsTarget = (String s) ->
                s.equals(endWord);

        final Function<String, HashSet<String> > stateExtend = (String s) -> {
            HashSet<String> result = new HashSet<>();

            char[] array = s.toCharArray();
            for (int i = 0; i < array.length; ++i) {
                final char old = array[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替换
                    if (c == array[i]) continue;

                    array[i] = c;
                    String newState = new String(array);
                    final int newDepth = visited.get(s) + 1;

                    if (stateIsValid.apply(newState)) {
                        if (visited.containsKey(newState)) {
                            final int depth = visited.get(newState);
                            if (depth < newDepth) {
                                // do nothing
                            } else if (depth == newDepth) {
                                result.add(newState);
                            } else {
                                throw new IllegalStateException("not possible to get he
re");

                            }

                        } else {
                            result.add(newState);
                        }
                    }
                    array[i] = old; // 恢复该单词
                }
            }

            return result;
        };

        List<List<String>> result = new ArrayList<>();
        q.offer(beginWord);
        visited.put(beginWord, 0);
        while (!q.isEmpty()) {
            String state = q.poll();

            // 如果当前路径长度已经超过当前最短路径长度，
            // 可以中止对该路径的处理，因为我们要找的是最短路径
            if (!result.isEmpty() && (visited.get(state) + 1) > result.get(0).size()) b
reak;
```

```
            if (stateIsTarget.apply(state)) {
                ArrayList<String> path = new ArrayList<>();
                genPath(father, beginWord, state, path, result);
                continue;
            }
            // 必须挪到下面，比如同一层A和B两个节点均指向了目标节点，
            // 那么目标节点就会在q中出现两次，输出路径就会翻倍
            // visited.insert(state);

            // 扩展节点
            HashSet<String> newStates = stateExtend.apply(state);
            for (String newState : newStates) {
                if (!visited.containsKey(newState)) {
                    q.offer(newState);
                    visited.put(newState, visited.get(state)+1);
                }
                ArrayList<String> parents = father.getOrDefault(newState, new ArrayList
<>());
                parents.add(state);
                father.put(newState, parents);
            }
        }
        return result;
    }
    private static void genPath(HashMap<String, ArrayList<String>> father,
                                String start, String state, List<String> path,
                                List<List<String>> result) {
        path.add(state);
        if (state.equals(start)) {
            if (!result.isEmpty()) {
                if (path.size() < result.get(0).size()) {
                    result.clear();
                } else if (path.size() == result.get(0).size()) {
                    // do nothing
                } else {
                    throw new IllegalStateException("not possible to get here");
                }
            }
            ArrayList<String> tmp = new ArrayList<>(path);
            Collections.reverse(tmp);
            result.add(tmp);
        } else {
            for (String f : father.get(state)) {
                genPath(father, start, f, path, result);
            }
        }
        path.remove(path.size() - 1);
    }
}
```

双队列

```java
// Word Ladder II
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord,
                                          Set<String> wordList) {
        // 当前层，下一层，用unordered_set是为了去重，例如两个父节点指向
        // 同一个子节点，如果用vector, 子节点就会在next里出现两次，其实此
        // 时 father 已经记录了两个父节点，next里重复出现两次是没必要的
        HashSet<String> current = new HashSet<>();
        HashSet<String> next = new HashSet<>();
        HashSet<String> visited = new HashSet<>(); // 判重
        HashMap<String, ArrayList<String>> father = new HashMap<>(); // DAG
        int level = -1; // 层次

        final Function<String, Boolean> stateIsValid = (String s) ->
                wordList.contains(s) || s.equals(endWord);
        final Function<String, Boolean> stateIsTarget = (String s) ->
                s.equals(endWord);

        final Function<String, HashSet<String> > stateExtend = (String s) -> {
            HashSet<String> result = new HashSet<>();

            char[] array = s.toCharArray();
            for (int i = 0; i < array.length; ++i) {
                final char old = array[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替换
                    if (c == array[i]) continue;

                    array[i] = c;
                    String newState = new String(array);

                    if (stateIsValid.apply(newState) &&
                            !visited.contains(newState)) {
                        result.add(newState);
                    }
                    array[i] = old; // 恢复该单词
                }
            }

            return result;
        };

        List<List<String>> result = new ArrayList<>();
        current.add(beginWord);
        while (!current.isEmpty()) {
            ++ level;
            // 如果当前路径长度已经超过当前最短路径长度，
            // 可以中止对该路径的处理，因为我们要找的是最短路径
            if (!result.isEmpty() && level + 1 > result.get(0).size()) break;

            // 1. 延迟加入visited, 这样才能允许两个父节点指向同一个子节点
            // 2. 一股脑current 全部加入visited, 是防止本层前一个节点扩展
```

```
            // 节点时，指向了本层后面尚未处理的节点，这条路径必然不是最短的
            for (String state : current)
                visited.add(state);

            for (String state : current) {
                if (stateIsTarget.apply(state)) {
                    ArrayList<String> path = new ArrayList<>();
                    genPath(father, beginWord, state, path, result);
                    continue;
                }
                // 扩展节点
                HashSet<String> newStates = stateExtend.apply(state);
                for (String newState : newStates) {
                    next.add(newState);
                    ArrayList<String> parents = father.getOrDefault(newState, new Array
List<>());
                    parents.add(state);
                    father.put(newState, parents);
                }
            }
            current.clear();
            // swap
            HashSet<String> tmp = current;
            current = next;
            next = tmp;

        }
        return result;
    }
    private static void genPath(HashMap<String, ArrayList<String>> father,
                                String start, String state, List<String> path,
                                List<List<String>> result) {
        path.add(state);
        if (state.equals(start)) {
            if (!result.isEmpty()) {
                if (path.size() < result.get(0).size()) {
                    result.clear();
                } else if (path.size() == result.get(0).size()) {
                    // do nothing
                } else {
                    throw new IllegalStateException("not possible to get here");
                }
            }
            ArrayList<String> tmp = new ArrayList<>(path);
            Collections.reverse(tmp);
            result.add(tmp);
        } else {
            for (String f : father.get(state)) {
                genPath(father, start, f, path, result);
            }
        }
        path.remove(path.size() - 1);
    }
```

```
}
```

图的广搜

前面的解法，在状态扩展的时候，每次都是从'a'到'z'全部枚举一遍，重复计算，比较浪费，其实当给定字典 `dict` 后，单词与单词之间的路径就固定下来了，本质上单词与单词之间构成了一个无向图。如果事先把这个图构建出来，那么状态扩展就会大大加快。

```
import java.util.*;
import java.util.function.Predicate;
import java.util.function.Function;

// Word Ladder II
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord,
                                          Set<String> wordList) {
        Queue<String> q = new LinkedList<>();
        HashMap<String, Integer> visited = new HashMap<>(); // 判重
        HashMap<String, ArrayList<String>> father = new HashMap<>(); // DAG
        // only used by stateExtend()
        final HashMap<String, HashSet<String>> g = buildGraph(wordList);

        final Function<String, Boolean> stateIsValid = (String s) ->
                wordList.contains(s) || s.equals(endWord);
        final Function<String, Boolean> stateIsTarget = (String s) ->
                s.equals(endWord);

        final Function<String, List<String> > stateExtend = (String s) -> {
            List<String> result = new ArrayList<>();
            final int newDepth = visited.get(s) + 1;
            HashSet<String> list = g.get(s);
            if (list == null) return result;

            for (String newState : list) {
                if (stateIsValid.apply(newState)) {
                    if (visited.containsKey(newState)) {
                        final int depth = visited.get(newState);
                        if (depth < newDepth) {
                            // do nothing
                        } else if (depth == newDepth) {
                            result.add(newState);
                        } else {
                            throw new IllegalStateException("not possible to get here")
;
                        }

                    } else {
                        result.add(newState);
                    }
                }
```

```
            }

            return result;
        };

        List<List<String>> result = new ArrayList<>();
        q.offer(beginWord);
        visited.put(beginWord, 0);
        while (!q.isEmpty()) {
            String state = q.poll();

            // 如果当前路径长度已经超过当前最短路径长度，
            // 可以中止对该路径的处理，因为我们要找的是最短路径
            if (!result.isEmpty() && (visited.get(state) + 1) > result.get(0).size()) b
reak;

            if (stateIsTarget.apply(state)) {
                ArrayList<String> path = new ArrayList<>();
                genPath(father, beginWord, state, path, result);
                continue;
            }
            // 必须挪到下面，比如同一层A和B两个节点均指向了目标节点，
            // 那么目标节点就会在q中出现两次，输出路径就会翻倍
            // visited.insert(state);

            // 扩展节点
            List<String> newStates = stateExtend.apply(state);
            for (String newState : newStates) {
                if (!visited.containsKey(newState)) {
                    q.offer(newState);
                    visited.put(newState, visited.get(state)+1);
                }
                ArrayList<String> parents = father.getOrDefault(newState, new ArrayList
<>());
                parents.add(state);
                father.put(newState, parents);
            }
        }
        return result;
    }
    private static void genPath(HashMap<String, ArrayList<String>> father,
                                String start, String state, List<String> path,
                                List<List<String>> result) {
        path.add(state);
        if (state.equals(start)) {
            if (!result.isEmpty()) {
                if (path.size() < result.get(0).size()) {
                    result.clear();
                } else if (path.size() == result.get(0).size()) {
                    // do nothing
                } else {
                    throw new IllegalStateException("not possible to get here");
                }
```

```
            }
            ArrayList<String> tmp = new ArrayList<>(path);
            Collections.reverse(tmp);
            result.add(tmp);
        } else {
            for (String f : father.get(state)) {
                genPath(father, start, f, path, result);
            }
        }
        path.remove(path.size() - 1);
    }

    private static HashMap<String, HashSet<String>> buildGraph(Set<String> dict) {
        HashMap<String, HashSet<String>> adjacency_list = new HashMap<>();
        for (String word: dict) {
            char[] array = word.toCharArray();
            for (int i = 0; i < array.length; ++i) {
                final char old = array[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替换
                    if (c == array[i]) continue;

                    array[i] = c;
                    String newWord = new String(array);

                    if (dict.contains(newWord)) {
                        HashSet<String> list = adjacency_list.getOrDefault(
                                word, new HashSet<>());
                        list.add(newWord);
                        adjacency_list.put(word, list);
                    }
                    array[i] = old; // 恢复该单词
                }
            }
        }
        return adjacency_list;
    }
}
```

相关题目

- Word Ladder

Surrounded Regions

描述

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

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

For example,

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

After running your function, the board should be:

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

分析

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

代码

```
// Surrounded Regions
// BFS，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public void solve(char[][] board) {
        if (board.length == 0) return;

        final int m = board.length;
        final int n = board[0].length;
        for (int i = 0; i < n; i++) {
            bfs(board, 0, i);
            bfs(board, m - 1, i);
        }
        for (int j = 1; j < m - 1; j++) {
            bfs(board, j, 0);
            bfs(board, j, n - 1);
        }
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                if (board[i][j] == 'O')
                    board[i][j] = 'X';
```

```
                else if (board[i][j] == '+')
                    board[i][j] = 'O';
    }
    private static void bfs(char[][] board, int i, int j) {
        Queue<State> q = new LinkedList<>();
        final int m = board.length;
        final int n = board[0].length;

        final Function<State, Boolean> stateIsValid = (State s) -> {
            if (s.x < 0 || s.x >= m || s.y < 0 || s.y >= n ||
                    board[s.x][s.y] != 'O')
                return false;
            return true;
        };
        final Function<State, ArrayList<State>> stateExtend = (State s) -> {
            ArrayList<State> result = new ArrayList<>();
            final int x = s.x;
            final int y = s.y;
            // 上下左右
            State[] newStates = new State[]{new State(x-1, y),
                    new State(x+1,y),
                    new State(x,y-1),
                    new State(x,y+1)
            };
            for (int k = 0; k < 4; ++k) {
                if (stateIsValid.apply(newStates[k])) {
                    // 既有标记功能又有去重功能
                    board[newStates[k].x][newStates[k].y] = '+';
                    result.add(newStates[k]);
                }
            }
            return result;
        };
        State start = new State(i, j);
        if (stateIsValid.apply(start)) {
            board[i][j] = '+';
            q.offer(start);
        }
        while (!q.isEmpty()) {
            State cur = q.poll();
            ArrayList<State> newStates = stateExtend.apply(cur);
            for (State s : newStates) q.offer(s);
        }
    }
    static class State {
        private int x;
        private int y;
        public State(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
}
```

小结

适用场景

输入数据：没什么特征，不像深搜，需要有“递归”的性质。如果是树或者图，概率更大。

状态转换图：树或者DAG图。

求解目标：多阶段最优化问题。

思考的步骤

1. 是求路径长度，还是路径本身（或动作序列）？
    i. 如果是求路径长度，则状态里面要存路径长度（或双队列+一个全局变量）
    ii. 如果是求路径本身或动作序列
        i. 要用一棵树存储宽搜过程中的路径
        ii. 是否可以预估状态个数的上限？能够预估状态总数，则开一个大数组，用树的双亲表示法；如果不能预估状态总数，则要使用一棵通用的树。这一步也是第4步的必要不充分条件。
2. 如何表示状态？即一个状态需要存储哪些些必要的数据，才能够完整提供如何扩展到下一步状态的所有信息。一般记录当前位置或整体局面。
3. 如何扩展状态？这一步跟第2步相关。状态里记录的数据不同，扩展方法就不同。对于固定不变的数据结构（一般题目直接给出，作为输入数据），如二叉树，图等，扩展方法很简单，直接往下一层走，对于隐式图，要先在第1步里想清楚状态所带的数据，想清楚了这点，那如何扩展就很简单了。
4. 如何判断重复？如果状态转换图是一颗树，则永远不会出现回路，不需要判重；如果状态转换图是一个图（这时候是一个图上的BFS），则需要判重。
    i. 如果是求最短路径长度或一条路径，则只需要让“点”（即状态）不重复出现，即可保证不出现回路
    ii. 如果是求所有路径，注意此时，状态转换图是DAG，即允许两个父节点指向同一个子节点。具体实现时，每个节点要“延迟”加入到已访问集合 `visited` ，要等一层全部访问完后，再加入到 `visited` 集合。
    iii. 具体实现
        i. 状态是否存在完美哈希方案？即将状态一一映射到整数，互相之间不会冲突。
        ii. 如果不存在，则需要使用通用的哈希表（自己实现或用标准库，例如 `unordered_set` ）来判重；自己实现哈希表的话，如果能够预估状态个数的上限，则可以开两个数组，head和next，表示哈希表，参考第 ??? 节方案2。
        iii. 如果存在，则可以开一个大布尔数组，来判重，且此时可以精确计算出状态总数，而不仅仅是预估上限。
5. 目标状态是否已知？如果题目已经给出了目标状态，可以带来很大便利，这时候可以从起始状态出发，正向广搜；也可以从目标状态出发，逆向广搜；也可以同时出发，双向广搜。

代码模板

广搜需要一个队列，用于一层一层扩展，一个hashset，用于判重，一棵树（只求长度时不需要），用于存储整棵树。

对于队列，可以用 `queue` ，也可以把 `vector` 当做队列使用。当求长度时，有两种做法：

1. 只用一个队列，但在状态结构体 `state_t` 里增加一个整数字段 `level` ，表示当前所在的层次，当碰到目标状态，直接输出 `level` 即可。这个方案，可以很容易的变成A*算法，把 `queue` 替换为 `priority_queue` 即可。
2. 用两个队列， `current, next` ，分别表示当前层次和下一层，另设一个全局整数 `level` ，表示层数（也即路径长度），当碰到目标状态，输出 `level` 即可。这个方案，状态里可以不存路径长度，只需全局设置一个整数 `level` ，比较节省内存；

对于hashset，如果有完美哈希方案，用布尔数组(`bool visited[STATE_MAX]` 或 `vector<bool> visited(STATE_MAX, false)`)来表示；如果没有，可以用STL里的 `set` 或 `unordered_set` 。

对于树，如果用STL，可以用 `unordered_map<state_t, state_t > father` 表示一颗树，代码非常简洁。如果能够预估状态总数的上限（设为STATE_MAX），可以用数组(`state_t nodes[STATE_MAX]`)，即树的双亲表示法来表示树，效率更高，当然，需要写更多代码。

如何表示状态

```
/** 状态 */
struct state_t {
    int data1;  /** 状态的数据，可以有多个字段. */
    int data2;  /** 状态的数据，可以有多个字段. */
    // dataN;   /** 其他字段 */
    int action; /** 由父状态移动到本状态的动作，求动作序列时需要. */
    int level;  /** 所在的层次（从0开始），也即路径长度-1，求路径长度时需要；
                    不过，采用双队列时不需要本字段，只需全局设一个整数 */
    bool operator==(const state_t &other) const {
        return true;  // 根据具体问题实现
    }
};

// 定义hash函数

// 方法1：模板特化，当hash函数只需要状态本身，不需要其他数据时，用这个方法比较简洁
namespace std {
template<> struct hash<state_t> {
    size_t operator()(const state_t & x) const {
        return 0; // 根据具体问题实现
    }
};
}

// 方法2：函数对象，如果hash函数需要运行时数据，则用这种方法
class Hasher {
public:
    Hasher(int _m) : m(_m) {};
    size_t operator()(const state_t &s) const {
        return 0; // 根据具体问题实现
```

```
    }
private:
    int m; // 存放外面传入的数据
};

/**
 * @brief 反向生成路径，求一条路径.
 * @param[in] father 树
 * @param[in] target 目标节点
 * @return 从起点到target的路径
 */
vector<state_t> gen_path(const unordered_map<state_t, state_t> &father,
        const state_t &target) {
    vector<state_t> path;
    path.push_back(target);

    for (state_t cur = target; father.find(cur) != father.end();
            cur = father.at(cur))
        path.push_back(cur);

    reverse(path.begin(), path.end());

    return path;
}

/**
 * 反向生成路径，求所有路径.
 * @param[in] father 存放了所有路径的树
 * @param[in] start 起点
 * @param[in] state 终点
 * @return 从起点到终点的所有路径
 */
void gen_path(unordered_map<state_t, vector<state_t> > &father,
        const string &start, const state_t& state, vector<state_t> &path,
        vector<vector<state_t> > &result) {
    path.push_back(state);
    if (state == start) {
        if (!result.empty()) {
            if (path.size() < result[0].size()) {
                result.clear();
                result.push_back(path);
            } else if(path.size() == result[0].size()) {
                result.push_back(path);
            } else {
                // not possible
                throw std::logic_error("not possible to get here");
            }
        } else {
            result.push_back(path);
        }
        reverse(result.back().begin(), result.back().end());
    } else {
        for (const auto& f : father[state]) {
```

```
            gen_path(father, start, f, path, result);
        }
    }
    path.pop_back();
}
```

求最短路径长度或一条路径

单队列的写法

```
#include "bfs_common.h"

/**
 * @brief 广搜，只用一个队列.
 * @param[in] start 起点
 * @param[in] data 输入数据
 * @return 从起点到目标状态的一条最短路径
 */
vector<state_t> bfs(state_t &start, const vector<vector<int>> &grid) {
    queue<state_t> q; // 队列
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father; // 树，求路径本身时才需要

    // 判断状态是否合法
    auto state_is_valid = [&](const state_t &s) { /*...*/ };

    // 判断当前状态是否为所求目标
    auto state_is_target = [&](const state_t &s) { /*...*/ };

    // 扩展当前状态
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state) &&
                    visited.find(new_state) != visited.end()) {
                result.insert(new_state);
            }
        }
        return result;
    };

    assert (start.level == 0);
    q.push(start);
    while (!q.empty()) {
        // 千万不能用 const auto&，pop() 会删除元素，
        // 引用就变成了悬空引用
        const state_t state = q.front();
        q.pop();
        visited.insert(state);

        // 访问节点
```

```
        if (state_is_target(state)) {
            return return gen_path(father, target); // 求一条路径
            // return state.level + 1; // 求路径长度
        }

        // 扩展节点
        vector<state_t> new_states = state_extend(state);
        for (const auto& new_state : new_states) {
            q.push(new_state);
            father[new_state] = state;  // 求一条路径
            // visited.insert(state); // 优化：可以提前加入 visited 集合，
            // 从而缩小状态扩展。这时 q 的含义略有变化，里面存放的是处理了一半
            // 的节点：已经加入了visited，但还没有扩展。别忘记 while循环开始
            // 前，要加一行代码, visited.insert(start)
        }
    }

    return vector<state_t>();
    //return 0;
}
```

双队列的写法

```
#include "bfs_common.h"

/**
 * @brief 广搜，使用两个队列.
 * @param[in] start 起点
 * @param[in] data 输入数据
 * @return 从起点到目标状态的一条最短路径
 */
vector<state_t> bfs(const state_t &start, const type& data) {
    queue<state_t> next, current; // 当前层，下一层
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father; // 树，求路径本身时才需要

    int level = -1;  // 层次

    // 判断状态是否合法
    auto state_is_valid = [&](const state_t &s) { /*...*/ };

    // 判断当前状态是否为所求目标
    auto state_is_target = [&](const state_t &s) { /*...*/ };

    // 扩展当前状态
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state) &&
                    visited.find(new_state) != visited.end()) {
                result.insert(new_state);
```

```
            }
        }
        return result;
    };

    current.push(start);
    while (!current.empty()) {
        ++level;
        while (!current.empty()) {
            // 千万不能用 const auto&，pop() 会删除元素，
            // 引用就变成了悬空引用
            const auto state = current.front();
            current.pop();
            visited.insert(state);

            if (state_is_target(state)) {
                return return gen_path(father, state); // 求一条路径
                // return state.level + 1; // 求路径长度
            }

            const auto& new_states = state_extend(state);
            for (const auto& new_state : new_states) {
                next.push(new_state);
                father[new_state] = state;
                // visited.insert(state); // 优化：可以提前加入 visited 集合，
                // 从而缩小状态扩展。这时 current 的含义略有变化，里面存放的是处
                // 理了一半的节点：已经加入了visited，但还没有扩展。别忘记 while
                // 循环开始前，要加一行代码, visited.insert(start)
            }
        }
        swap(next, current); //!!! 交换两个队列
    }

    return vector<state_t>();
    // return 0;
}
```

求所有路径

单队列

```
/**
 * @brief 广搜，使用一个队列.
 * @param[in] start 起点
 * @param[in] data 输入数据
 * @return 从起点到目标状态的所有最短路径
 */
vector<vector<state_t> > bfs(const state_t &start, const type& data) {
    queue<state_t> q;
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, vector<state_t> > father; // DAG
```

```
    auto state_is_valid = [&](const state_t& s) { /*...*/ };
    auto state_is_target = [&](const state_t &s) { /*...*/ };
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state)) {
                auto visited_iter = visited.find(new_state);

                if (visited_iter != visited.end()) {
                    if (visited_iter->level < new_state.level) {
                        // do nothing
                    } else if (visited_iter->level == new_state.level) {
                        result.insert(new_state);
                    } else { // not possible
                        throw std::logic_error("not possible to get here");
                    }
                } else {
                    result.insert(new_state);
                }
            }
        }

        return result;
    };

    vector<vector<string>> result;
    state_t start_state(start, 0);
    q.push(start_state);
    visited.insert(start_state);
    while (!q.empty()) {
        // 千万不能用 const auto&，pop() 会删除元素，
        // 引用就变成了悬空引用
        const auto state = q.front();
        q.pop();

        // 如果当前路径长度已经超过当前最短路径长度，
        // 可以中止对该路径的处理，因为我们要找的是最短路径
        if (!result.empty() && state.level + 1 > result[0].size()) break;

        if (state_is_target(state)) {
            vector<string> path;
            gen_path(father, start_state, state, path, result);
            continue;
        }
        // 必须挪到下面，比如同一层A和B两个节点均指向了目标节点，
        // 那么目标节点就会在q中出现两次，输出路径就会翻倍
        // visited.insert(state);

        // 扩展节点
        const auto& new_states = state_extend(state);
        for (const auto& new_state : new_states) {
            if (visited.find(new_state) == visited.end()) {
```

```
                q.push(new_state);
            }
            visited.insert(new_state);
            father[new_state].push_back(state);
        }
    }

    return result;
}
```

双队列的写法

```
#include "bfs_common.h"

/**
 * @brief 广搜，使用两个队列.
 * @param[in] start 起点
 * @param[in] data 输入数据
 * @return 从起点到目标状态的所有最短路径
 */
vector<vector<state_t> > bfs(const state_t &start, const type& data) {
    // 当前层，下一层，用unordered_set是为了去重，例如两个父节点指向
    // 同一个子节点，如果用vector, 子节点就会在next里出现两次，其实此
    // 时 father 已经记录了两个父节点，next里重复出现两次是没必要的
    unordered_set<string> current, next;
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, vector<state_t> > father; // DAG

    int level = -1;  // 层次

    // 判断状态是否合法
    auto state_is_valid = [&](const state_t &s) { /*...*/ };

    // 判断当前状态是否为所求目标
    auto state_is_target = [&](const state_t &s) { /*...*/ };

    // 扩展当前状态
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state) &&
                    visited.find(new_state) != visited.end()) {
                result.insert(new_state);
            }
        }
        return result;
    };

    vector<vector<state_t> > result;
    current.insert(start);
    while (!current.empty()) {
```

```
        ++ level;
        // 如果当前路径长度已经超过当前最短路径长度，可以中止对该路径的
        // 处理，因为我们要找的是最短路径
        if (!result.empty() && level+1 > result[0].size()) break;

        // 1. 延迟加入visited, 这样才能允许两个父节点指向同一个子节点
        // 2. 一股脑current 全部加入visited, 是防止本层前一个节点扩展
        // 节点时，指向了本层后面尚未处理的节点，这条路径必然不是最短的
        for (const auto& state : current)
            visited.insert(state);
        for (const auto& state : current) {
            if (state_is_target(state)) {
                vector<string> path;
                gen_path(father, path, start, state, result);
                continue;
            }

            const auto new_states = state_extend(state);
            for (const auto& new_state : new_states) {
                next.insert(new_state);
                father[new_state].push_back(state);
            }
        }

        current.clear();
        swap(current, next);
    }

    return result;
}
```

本章主要讲各种深度优先搜索。

Additive Number

描述

Additive number is a string whose digits can form additive sequence.

A valid additive sequence should contain **at least** three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

For example:

`"112358"` is an additive number because the digits can form an additive sequence: `1, 1, 2, 3, 5, 8`.

```
1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
```

`"199100199"` is also an additive number, the additive sequence is: `1, 99, 100, 199`.

```
1 + 99 = 100, 99 + 100 = 199
```

Note: Numbers in the additive sequence cannot have leading zeros, so sequence `1, 2, 03` or `1, 02, 3` is invalid.

Given a string containing only digits `'0'-'9'`, write a function to determine if it's an additive number.

Follow up:

How would you handle overflow for very large input integers?

分析

这是一个多阶段决策问题，且必须走到字符串最后一个字符才能得出结论，因此适合用深搜或DP。

再仔细想一下状态转换图，每次索引变化一下，就跟之前的完全没有重复，因此状态转换图是一颗树，不是DAG，因此不存在重叠子问题，因此排除DP，本题应该用深搜。

代码

```java
// Additive Number
// 多入口深搜
// 时间复杂度O(n^3)，空间复杂度O(1)
public class Solution {
    public boolean isAdditiveNumber(String num) {
        for (int i = 1; i <= num.length() / 2; ++i) {
            if (num.charAt(0) == '0' && i > 1) continue;
            for (int j = i + 1; j < num.length(); ++j) {
                if (num.charAt(i) == '0' && j - i > 1) continue;
                if (dfs(num, 0, i, j)) return true;
            }
        }
        return false;
    }

    // 判断从 [i, j) 和 [j, k) 出发,能否走到尽头
    private static boolean dfs(String num, int i, int j, int k) {
        long num1 = Long.parseLong(num.substring(i, j));
        long num2 = Long.parseLong(num.substring(j, k));
        final String addition = String.valueOf(num1 + num2);

        if (!num.substring(k).startsWith(addition)) return false;
        if (k + addition.length() == num.length()) return true;
        return dfs(num, j, k, k + addition.length());
    }
}
```

Palindrome Partitioning

描述

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"]
]
```

分析

在每一步都可以判断中间结果是否为合法结果，用回溯法。

一个长度为n的字符串，有 `n-1` 个地方可以砍断，每个地方可断可不断，因此复杂度为$O(2^{n-1})$

深搜1

```java
// Palindrome Partitioning
// 时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<>();
        List<String> path = new ArrayList<>(); // 一个partition方案
        dfs(s, path, result, 0, 1);
        return result;
    }

    // prev 表示前一个隔板, start 表示当前隔板
    private static void dfs(String s, List<String> path,
                            List<List<String>> result, int prev, int start) {
        if (start == s.length()) { // 最后一个隔板
            if (isPalindrome(s, prev, start - 1)) { // 必须使用
                path.add(s.substring(prev, start));
                result.add(new ArrayList<>(path));
                path.remove(path.size() - 1);
            }
            return;
        }
        // 不断开
        dfs(s, path, result, prev, start + 1);
        // 如果[prev, start-1] 是回文，则可以断开，也可以不断开（上一行已经做了）
        if (isPalindrome(s, prev, start - 1)) {
            // 断开
            path.add(s.substring(prev, start));
            dfs(s, path, result, start, start + 1);
            path.remove(path.size() - 1);
        }
    }

    private static boolean isPalindrome(String s, int start, int end) {
        while (start < end && s.charAt(start) == s.charAt(end)) {
            ++start;
            --end;
        }
        return start >= end;
    }
}
```

深搜2

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

```java
// Palindrome Partitioning
// 时间复杂度O(2^n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<>();
        List<String> path = new ArrayList<>();  // 一个partition方案
        dfs(s, path, result, 0);
        return result;
    }
    // 搜索必须以s[start]开头的partition方案
    private static void dfs(String s, List<String> path,
                            List<List<String>> result, int start) {
        if (start == s.length()) {
            result.add(new ArrayList<>(path));
            return;
        }
        for (int i = start; i < s.length(); i++) {
            if (isPalindrome(s, start, i)) { // 从i位置砍一刀
                path.add(s.substring(start, i+1));
                dfs(s, path, result, i + 1);  // 继续往下砍
                path.remove(path.size() - 1); // 撤销上上行
            }
        }
    }
    private static boolean isPalindrome(String s, int start, int end) {
        while (start < end && s.charAt(start) == s.charAt(end)) {
            ++start;
            --end;
        }
        return start >= end;
    }
}
```

动规

```java
// Palindrome Partitioning
// 动规，时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public List<List<String>> partition(String s) {
        final int n = s.length();
        boolean[][] p = new boolean[n][n]; // whether s[i,j] is palindrome
        for (int i = n - 1; i >= 0; --i)
            for (int j = i; j < n; ++j)
                p[i][j] = s.charAt(i) == s.charAt(j) &&
                        ((j - i < 2) || p[i + 1][j - 1]);

        List<List<String>>[] subPalins = new ArrayList[n]; // sub palindromes of s[0,i]
        for (int i = 0; i < n; ++i) subPalins[i] = new ArrayList<>();
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i; j < n; ++j)
                if (p[i][j]) {
                    String palindrome = s.substring(i, j+1);
                    if (j + 1 < n) {
                        for (List<String> v : subPalins[j + 1]) {
                            ArrayList<String> tmp = new ArrayList<>(v);
                            tmp.add(0, palindrome);
                            subPalins[i].add(tmp);
                        }
                    } else {
                        ArrayList<String> tmp = new ArrayList<>();
                        tmp.add(palindrome);
                        subPalins[i].add(tmp);
                    }
                }
        }
        return subPalins[0];
    }
}
```

相关题目

- Palindrome Partitioning II

Unique Paths

描述

A robot is located at the top-left corner of a `m × 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?

Figure: Above is a `3 × 7` grid. How many possible unique paths are there?

Note: `m` and `n` will be at most 100.

深搜

```
// Unique Paths
// 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 class Solution {
    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);
    }
}
```

备忘录法

给前面的深搜，加个缓存，就可以过大集合了。即备忘录法。

```
// Unique Paths
// 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)
public class Solution {
    public int uniquePaths(int m, int n) {
        // f[x][y] 表示 从(0,0)到(x,y)的路径条数
        f = new int[m][n];
        f[0][0] = 1;
        return dfs(m - 1, n - 1);
    }

    int dfs(int x, int y) {
        if (x < 0 || y < 0) return 0; // 数据非法，终止条件

        if (x == 0 && y == 0) return f[0][0]; // 回到起点，收敛条件

        if (f[x][y] > 0) {
            return f[x][y];
        } else {
            return f[x][y] = dfs(x - 1, y) +  dfs(x, y - 1);
        }
    }
    private int[][] f;  // 缓存
}
```

动规

既然可以用备忘录法自顶向下解决，也一定可以用动规自底向上解决。

设状态为 `f[i][j]` ，表示从起点 `(1,1)` 到达 `(i,j)` 的路线条数，则状态转移方程为：

```
f[i][j]=f[i-1][j]+f[i][j-1]
```

```
// Unique Paths
// 动规，滚动数组
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public int uniquePaths(int m, int n) {
        int[] f = new int[n];
        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] + f[j - 1];
            }
        }
        return f[n - 1];
    }
}
```

数学公式

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} 。

```
// LeetCode, Unique Paths
// 数学公式
class Solution {
public:
    typedef long long int64_t;
    // 求阶乘, n!/(start-1)!，即 n*(n-1)...start，要求 n >= 1
    static int64_t factor(int n, int start = 1) {
        int64_t  ret = 1;
        for(int i = start; i <= n; ++i)
            ret *= i;
        return ret;
    }
    // 求组合数 C_n^k
    static int64_t combination(int n, int k) {
        // 常数优化
        if (k == 0) return 1;
        if (k == 1) return n;

        int64_t ret = factor(n, k+1);
        ret /= factor(n - k);
        return ret;
    }

    int uniquePaths(int m, int n) {
        // max 可以防止n和k差距过大，从而防止combination()溢出
        return combination(m+n-2, max(m-1, n-1));
    }
};
```

相关题目

- Unique Paths II
- Minimum Path Sum

Unique Paths II

描述

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 `3 × 3` 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.

备忘录法

在上一题的基础上改一下即可。相比动规，简单得多。

```
// Unique Paths II
// 深搜 + 缓存，即备忘录法
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        final int m = obstacleGrid.length;
        final int n = obstacleGrid[0].length;
        if (obstacleGrid[0][0] != 0 ||
                obstacleGrid[m - 1][n - 1] != 0) return 0;

        f = new int[m][n];
        f[0][0] = obstacleGrid[0][0] != 0 ? 0 : 1;
        return dfs(obstacleGrid, m - 1, n - 1);
    }

    // @return 从 (0, 0) 到 (x, y) 的路径总数
    int dfs(int[][] obstacleGrid, int x, int y) {
        if (x < 0 || y < 0) return 0; // 数据非法，终止条件

        // (x,y)是障碍
        if (obstacleGrid[x][y] != 0) return 0;

        if (x == 0 && y == 0) return f[0][0]; // 回到起点，收敛条件

        if (f[x][y] > 0) {
            return f[x][y];
        } else {
            return f[x][y] = dfs(obstacleGrid, x - 1, y) +
                    dfs(obstacleGrid, x, y - 1);
        }
    }
    private int[][] f;  // 缓存
}
```

动规

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，那么后面的行全为0。

```java
// Unique Paths II
// 动规，滚动数组
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        final int m = obstacleGrid.length;
        final int n = obstacleGrid[0].length;
        if (obstacleGrid[0][0] != 0 ||
                obstacleGrid[m-1][n-1] != 0) return 0;

        int[] f = new int[n];
        f[0] = obstacleGrid[0][0] != 0 ? 0 : 1;

        for (int i = 0; i < m; i++) {
            f[0] = f[0] == 0 ? 0 : (obstacleGrid[i][0] != 0 ? 0 : 1);
            for (int j = 1; j < n; j++)
                f[j] = obstacleGrid[i][j] != 0 ? 0 : (f[j] + f[j - 1]);
        }

        return f[n - 1];
    }
}
```

相关题目

- Unique Paths
- Minimum Path Sum

N-Queens

描述

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

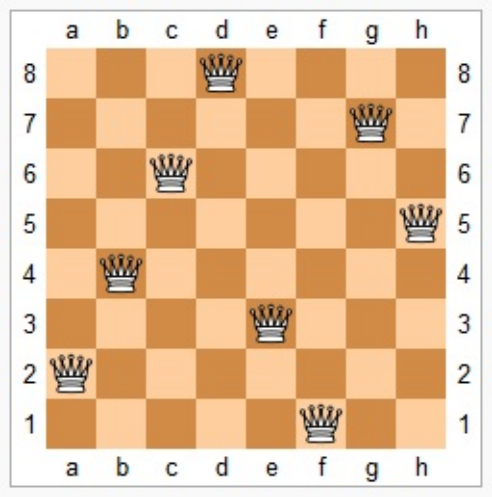


Figure: Eight Queens

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

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

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

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

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

分析

经典的深搜题。

设置一个数组 `vector<int> C(n, 0)`，`C[i]` 表示第i行皇后所在的列编号，即在位置 `(i, C[i])` 上放了一个皇后，这样用一个一维数组，就能记录整个棋盘。

代码1

```
// N-Queens
// 深搜+剪枝
// 时间复杂度O(n!*n)，空间复杂度O(n)
public class Solution {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> result = new ArrayList<>();
        int[] C = new int[n]; // C[i]表示第i行皇后所在的列编号
        dfs(C, 0, result);
        return result;
    }
    private static void dfs(int[] C, int row, List<List<String>> result) {
        final int N = C.length;
        if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            List<String> solution = new ArrayList<>();
            for (int i = 0; i < N; ++i) {
                char[] charArray = new char[N];
                Arrays.fill(charArray, '.');
                for (int j = 0; j < N; ++j) {
                    if (j == C[i]) charArray[j] = 'Q';
                }
                solution.add(new String(charArray));
            }
            result.add(solution);
            return;
        }

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            final boolean ok = isValid(C, row, j);
            if (!ok) continue;  // 剪枝，如果非法，继续尝试下一列
            // 执行扩展动作
            C[row] = j;
            dfs(C, row + 1, result);
            // 撤销动作
            // C[row] = -1;
        }
    }

    /**
     * 能否在 (row, col) 位置放一个皇后.
     *
     * @param C 棋局
     * @param row 当前正在处理的行，前面的行都已经放了皇后了
     * @param col 当前列
     * @return 能否放一个皇后
     */
    private static boolean isValid(int[] C, int row, int col) {
        for (int i = 0; i < row; ++i) {
```

```
            // 在同一列
            if (C[i] == col) return false;
            // 在同一对角线上
            if (Math.abs(i - row) == Math.abs(C[i] - col)) return false;
        }
        return true;
    }
}
```

代码2

```java
// N-Queens
// 深搜+剪枝
// 时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public List<List<String>> solveNQueens(int n) {
        this.columns = new boolean[n];
        this.main_diag = new boolean[2 * n - 1];
        this.anti_diag = new boolean[2 * n - 1];

        List<List<String>> result = new ArrayList<>();
        int[] C = new int[n];
        Arrays.fill(C, -1); // C[i]表示第i行皇后所在的列编号
        dfs(C, 0, result);
        return result;
    }


    private void dfs(int[] C, int row, List<List<String>> result) {
        final int N = C.length;
        if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            List<String> solution = new ArrayList<>();
            for (int i = 0; i < N; ++i) {
                char[] charArray = new char[N];
                Arrays.fill(charArray, '.');
                for (int j = 0; j < N; ++j) {
                    if (j == C[i]) charArray[j] = 'Q';
                }
                solution.add(new String(charArray));
            }
            result.add(solution);
            return;
        }

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            final boolean ok = !columns[j] && !main_diag[row - j + N - 1]  &&
                    !anti_diag[row + j];
            if (!ok) continue;  // 剪枝，如果非法，继续尝试下一列
            // 执行扩展动作
            C[row] = j;
            columns[j] = main_diag[row - j + N - 1] = anti_diag[row + j] = true;
            dfs(C, row + 1, result);
            // 撤销动作
            // C[row] = -1;
            columns[j] = main_diag[row - j + N - 1] = anti_diag[row + j] = false;
        }
    }
    // 这三个变量用于剪枝
    private boolean[] columns;  // 表示已经放置的皇后占据了哪些列
    private boolean[] main_diag;  // 占据了哪些主对角线
    private boolean[] anti_diag;  // 占据了哪些副对角线
}
```

相关题目

- N-Queens II

N-Queens II

描述

Follow up for N-Queens problem.

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

分析

只需要输出解的个数，不需要输出所有解，代码要比上一题简化很多。设一个全局计数器，每找到一个解就增1。

代码1

```java
// N-Queens II
// 深搜+剪枝
// 时间复杂度O(n!*n)，空间复杂度O(n)
public class Solution {
    public int totalNQueens(int n) {
        this.count = 0;

        int[] C = new int[n]; // C[i]表示第i行皇后所在的列编号
        dfs(C, 0);
        return this.count;
    }

    void dfs(int[] C, int row) {
        final int n = C.length;
        if (row == n) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            ++this.count;
            return;
        }

        for (int j = 0; j < n; ++j) {  // 扩展状态，一列一列的试
            final boolean ok = isValid(C, row, j);
            if (!ok) continue;  // 剪枝：如果合法，继续递归
            // 执行扩展动作
            C[row] = j;
            dfs(C, row + 1);
            // 撤销动作
            // C[row] = -1;
        }
    }
    /**
     * 能否在 (row, col) 位置放一个皇后.
     *
     * @param C 棋局
     * @param row 当前正在处理的行，前面的行都已经放了皇后了
     * @param col 当前列
     * @return 能否放一个皇后
     */
    private static boolean isValid(int[] C, int row, int col) {
        for (int i = 0; i < row; ++i) {
            // 在同一列
            if (C[i] == col) return false;
            // 在同一对角线上
            if (Math.abs(i - row) == Math.abs(C[i] - col)) return false;
        }
        return true;
    }
    private int count; // 解的个数
}
```

代码2

```java
// N-Queens II
// 深搜+剪枝
// 时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public int totalNQueens(int n) {
        this.count = 0;
        this.columns = new boolean[n];
        this.main_diag = new boolean[2 * n - 1];
        this.anti_diag = new boolean[2 * n - 1];

        int[] C = new int[n]; // C[i]表示第i行皇后所在的列编号
        dfs(C, 0);
        return this.count;
    }

    void dfs(int[] C, int row) {
        final int N = C.length;
        if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            ++this.count;
            return;
        }

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            final boolean ok = !columns[j] &&
                    !main_diag[row - j + N - 1] &&
                    !anti_diag[row + j];
            if (!ok) continue;  // 剪枝：如果合法，继续递归
            // 执行扩展动作
            C[row] = j;
            columns[j] = main_diag[row - j + N - 1] =
                    anti_diag[row + j] = true;
            dfs(C, row + 1);
            // 撤销动作
            // C[row] = -1;
            columns[j] = main_diag[row - j + N - 1] =
                    anti_diag[row + j] = false;
        }
    }

    private int count; // 解的个数
    // 这三个变量用于剪枝
    private boolean[] columns;   // 表示已经放置的皇后占据了哪些列
    private boolean[] main_diag; // 占据了哪些主对角线
    private boolean[] anti_diag; // 占据了哪些副对角线

    public static void main(String[] args) {
        Solution s = new Solution();
        s.totalNQueens(1);
    }
}
```

相关题目

- N-Queens

Restore IP Addresses

描述

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)

分析

必须要走到底部才能判断解是否合法，深搜。

代码

```java
// Restore IP Addresses
// 时间复杂度O(n^4)，空间复杂度O(n)
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> result = new ArrayList<>();
        List<String> ip = new ArrayList<>();; // 存放中间结果
        dfs(s, ip, result, 0);
        return result;
    }

    /**
     * 解析字符串
     * @param[in] s 字符串，输入数据
     * @param[out] ip 存放中间结果
     * @param[out] result 存放所有可能的IP地址
     * @param[in] start 当前正在处理的 index
     * @return 无
     */
    private static void dfs(String s, List<String> ip,
                            List<String> result, int start) {
        if (ip.size() == 4 && start == s.length()) {  // 找到一个合法解
            result.add(ip.get(0) + '.' + ip.get(1) + '.' +
                    ip.get(2) + '.' + ip.get(3));
            return;
        }

        if (s.length() - start > (4 - ip.size()) * 3)
            return;  // 剪枝
        if (s.length() - start < (4 - ip.size()))
            return;  // 剪枝

        int num = 0;
        for (int i = start; i < start + 3 && i < s.length(); i++) {
            num = num * 10 + (s.charAt(i) - '0');

            if (num < 0 || num > 255) continue;  // 剪枝

            ip.add(s.substring(start, i + 1));
            dfs(s, ip, result, i + 1);
            ip.remove(ip.size() - 1);

            if (num == 0) break;  // 不允许前缀0，但允许单个0
        }
    }

    public static void main(String[] args) {
        Solution s = new Solution();
        s.restoreIpAddresses("1111");
    }
}
```

Combination Sum

描述

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

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

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination $(a_1, a_2, ..., a_k)$ must be in non-descending order. (ie, $a_1 \leq a_2 \leq ... \leq a_k$).
- 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]
```

分析

无

代码

```java
// Combination Sum
// 时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> combinationSum(int[] nums, int target) {
        Arrays.sort(nums);
        List<List<Integer>> result = new ArrayList<>(); // 最终结果
        List<Integer> path = new ArrayList<>(); // 中间结果
        dfs(nums, path, result, target, 0);
        return result;
    }

    private static void dfs(int[] nums, List<Integer> path,
                            List<List<Integer>> result, int gap, int start) {
        if (gap == 0) {  // 找到一个合法解
            result.add(new ArrayList<Integer>(path));
            return;
        }
        for (int i = start; i < nums.length; i++) { // 扩展状态
            if (gap < nums[i]) return; // 剪枝

            path.add(nums[i]); // 执行扩展动作
            dfs(nums, path, result, gap - nums[i], i);
            path.remove(path.size() - 1);  // 撤销动作
        }
    }
}
```

相关题目

- Combination Sum II
- Combination Sum III

Combination Sum II

描述

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

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

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination $(a_1, a_2, ..., a_k)$ must be in non-descending order. (ie, $a_1 > a_2 > ... > a_k$).
- 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]
```

分析

无

代码

```java
// Combination Sum II
// 时间复杂度O(n!)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> combinationSum2(int[] nums, int target) {
        Arrays.sort(nums); // 跟第 50 行配合，
        // 确保每个元素最多只用一次
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> path = new ArrayList<>();
        dfs(nums, path, result, target, 0);
        return result;
    }
    // 使用nums[start, nums.size())之间的元素，能找到的所有可行解
    private static void dfs(int[] nums, List<Integer> path,
                            List<List<Integer>> result, int gap, int start) {
        if (gap == 0) {  //  找到一个合法解
            result.add(new ArrayList<>(path));
            return;
        }

        int previous = -1;
        for (int i = start; i < nums.length; i++) {
            // 如果上一轮循环已经使用了nums[i]，则本次循环就不能再选nums[i]，
            // 确保nums[i]最多只用一次
            if (previous == nums[i]) continue;

            if (gap < nums[i]) return;  // 剪枝

            previous = nums[i];

            path.add(nums[i]);
            dfs(nums, path, result, gap - nums[i], i + 1);
            path.remove(path.size() - 1);  // 恢复环境
        }
    }
}
```

相关题目

- Combination Sum
- Combination Sum III

Combination Sum III

描述

Find all possible combinations of `k` numbers that add up to a number `n`, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Ensure that numbers within the set are sorted in ascending order.

Example 1:

Input: `k` =3, `n` =7

Output: `[[1,2,4]]`

Example 2:

Input: `k` =3, `n` =9

Output: `[[1,2,6], [1,3,5], [2,3,4]]`

分析

这是一个多阶段问题，目标是求所有解，显然用深搜+剪枝，即回溯法。

代码

```
// Combination Sum III
// Time Complexity: O(9*8*...*(10-k)), Space Complexity: O(k)
public class Solution {
    public List> combinationSum3(int k, int n) {
        final List> result = new ArrayList<>();
        final List path = new ArrayList<>();
        dfs(k, n, path, result);
        return result;
    }

    private static void dfs(int step, int gap, List path,
                            List> result) {
        if (step == 0) {
            if (gap == 0) {
                result.add(new ArrayList<>(path));
            }
            return;
        }

        if (gap < 1) return;

        final int start = path.isEmpty() ? 1 : path.get(path.size() - 1)+1;
        for (int i = start; i < 10; ++i) {
            path.add(i);
            dfs(step - 1, gap - i, path, result);
            path.remove(path.size() - 1);
        }
    }
}
```

相关题目

- Combination Sum
- Combination Sum II

Generate Parentheses

描述

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:

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

分析

小括号串是一个递归结构，跟单链表、二叉树等递归结构一样，首先想到用递归。

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

代码1

```java
// Generate Parentheses
// 时间复杂度O(TODO)，空间复杂度O(n)
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        StringBuilder path = new StringBuilder();
        if (n > 0) generate(n, path, result, 0, 0);
        return result;
    }
    // l 表示 ( 出现的次数, r 表示 ) 出现的次数
    private static void generate(int n, StringBuilder path,
                                 List<String> result, int l, int r) {
        if (l == n) {
            StringBuilder sb = new StringBuilder(path);
            for (int i = 0; i < n - r; ++i) sb.append(')');
            result.add(sb.toString());
            return;
        }

        path.append('(');
        generate(n, path, result, l + 1, r);
        path.deleteCharAt(path.length() - 1);

        if (l > r) {
            path.append(')');
            generate(n, path, result, l, r + 1);
            path.deleteCharAt(path.length() - 1);
        }
    }
}
```

代码2

另一种递归写法，更加简洁。

```java
// Generate Parentheses
// @author 连城 (http://weibo.com/lianchengzju)
public class Solution {
    public List<String> generateParenthesis(int n) {
        if (n == 0) return new ArrayList<>(Arrays.asList(""));
        if (n == 1) return new ArrayList<>(Arrays.asList("()"));
        List<String> result = new ArrayList<>();

        for (int i = 0; i < n; ++i)
            for (String inner : generateParenthesis (i))
                for (String outer : generateParenthesis (n - 1 - i))
                    result.add ("(" + inner + ")" + outer);

        return result;
    }
}
```

相关题目

- Valid Parentheses
- Longest Valid Parentheses

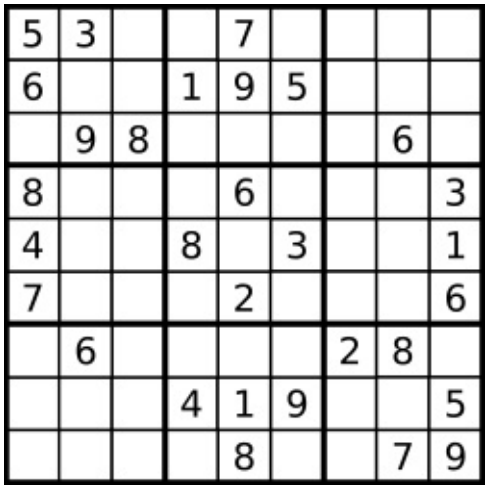
Sudoku Solver

描述

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

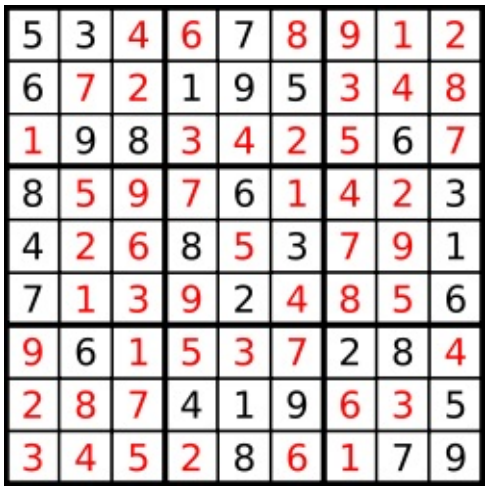
Empty cells are indicated by the character `'.'`.

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



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: A sudoku puzzle...



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: ...and its solution numbers marked in red

分析

无。

代码

```java
// Sudoku Solver
// 时间复杂度O(9^4)，空间复杂度O(1)
public class Solution {
    public void solveSudoku(char[][] board) {
        _solveSudoku(board);
    }
    private static boolean _solveSudoku(char[][] board) {
        for (int i = 0; i < 9; ++i)
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    for (int k = 0; k < 9; ++k) {
                        board[i][j] = Character.forDigit(k+1, 10);
                        if (isValid(board, i, j) && _solveSudoku(board))
                            return true;
                        board[i][j] = '.';
                    }
                    return false;
                }
            }
        return true;
    }
    // 检查 (x, y) 是否合法
    private static boolean isValid(char[][] board, int x, int y) {
        int i, j;
        for (i = 0; i < 9; i++) // 检查 y 列
            if (i != x && board[i][y] == board[x][y])
                return false;
        for (j = 0; j < 9; j++) // 检查 x 行
            if (j != y && board[x][j] == board[x][y])
                return false;
        for (i = 3 * (x / 3); i < 3 * (x / 3 + 1); i++)
            for (j = 3 * (y / 3); j < 3 * (y / 3 + 1); j++)
                if ((i != x || j != y) && board[i][j] == board[x][y])
                    return false;
        return true;
    }
}
```

相关题目

- Valid Sudoku

Word Search

描述

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 neighbouring. 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`.

分析

无。

代码

```java
// Word Search
// 深搜，递归
// 时间复杂度O(n^2*m^2)，空间复杂度O(n^2)
public class Solution {
    public boolean exist(char[][] board, String word) {
        final int m = board.length;
        final int n = board[0].length;
        boolean[][] visited = new boolean[m][n];
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                if (dfs(board, word, 0, i, j, visited))
                    return true;
        return false;
    }
    private static boolean dfs(char[][] board, String word,
                    int index, int x, int y, boolean[][] visited) {
        if (index == word.length())
            return true; // 收敛条件

        if (x < 0 || y < 0 || x >= board.length || y >= board[0].length)
            return false;  // 越界，终止条件

        if (visited[x][y]) return false; // 已经访问过，剪枝

        if (board[x][y] != word.charAt(index)) return false; // 不相等，剪枝

        visited[x][y] = true;
        boolean ret = dfs(board, word, index + 1, x - 1, y, visited) || // 上
                dfs(board, word, index + 1, x + 1, y, visited)    || // 下
                dfs(board, word, index + 1, x, y - 1, visited)    || // 左
                dfs(board, word, index + 1, x, y + 1, visited);      // 右
        visited[x][y] = false;
        return ret;
    }
}
```

小结

适用场景

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

状态转换图：树或者DAG。

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

思考的步骤

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

    i. 如果是路径条数，则不需要存储路径。
    ii. 如果是求路径本身，则要用一个数组 `path[]` 存储路径。跟宽搜不同，宽搜虽然最终求的也是一条路径，但是需要存储扩展过程中的所有路径，在没找到答案之前所有路径都不能放弃；而深搜，在搜索过程中始终只有一条路径，因此用一个数组就足够了。
2. 只要求一个解，还是要求所有解？如果只要求一个解，那找到一个就可以返回；如果要求所有解，找到了一个后，还要继续扩展，直到遍历完。广搜一般只要求一个解，因而不需要考虑这个问题（广搜当然也可以求所有解，这时需要扩展到所有叶子节点，相当于在内存中存储整个状态转换图，非常占内存，因此广搜不适合解这类问题）。

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

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

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

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

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

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

1. 关于判重

    i. 是否需要判重？如果状态转换图是一棵树，则不需要判重，因为在遍历过程中不可能重复；如果状态转换图是一个DAG，则需要判重。这一点跟BFS不一样，BFS的状态转换图总是DAG，必须要判重。

    ii. 怎样判重？跟广搜相同，见第 ??? 节。同时，DAG说明存在重叠子问题，此时可以用缓存加速，见第8步。 \end{enumerate}

2. 如何加速？

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

    ii. 缓存。

        i. 前提条件：状态转换图是一个DAG。DAG=>存在重叠子问题=>子问题的解会被重复利用，用缓存自然会有加速效果。如果依赖关系是树状的（例如树，单链表等），没必要加缓存，因为子问题只会一层层往下，用一次就再也不会用到，加了缓存也没什么加速效果。

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

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

代码模板

```
/**
 * dfs模板.
 * @param[in] input 输入数据指针
 * @param[out] path 当前路径，也是中间结果
 * @param[out] result 存放最终结果
 * @param[inout] cur or gap 标记当前位置或距离目标的距离
 * @return 路径长度，如果是求路径本身，则不需要返回长度
 */
void dfs(type &input, type &path, type &result, int cur or gap) {
    if (数据非法) return 0;   // 终止条件
    if (cur == input.size()) { // 收敛条件
    // if (gap == 0) {
        将path放入result
    }

    if (可以剪枝) return;

    for(...) { // 执行所有可能的扩展动作
        执行动作，修改path
        dfs(input, step + 1 or gap--, result);
        恢复path
    }
}
```

深搜与回溯法的区别

深搜(Depth-first search, DFS)的定义见 http://en.wikipedia.org/wiki/Depth_first_search，回溯法(backtracking)的定义见 http://en.wikipedia.org/wiki/Backtracking

回溯法 = 深搜 + 剪枝。一般大家用深搜时，或多或少会剪枝，因此深搜与回溯法没有什么不同，可以在它们之间画上一个等号。本书同时使用深搜和回溯法两个术语，但读者可以认为二者等价。

深搜一般用递归(recursion)来实现，这样比较简洁。

深搜能够在候选答案生成到一半时，就进行判断，抛弃不满足要求的答案，所以深搜比暴力搜索法要快。

深搜与递归的区别

深搜经常用递归(recursion)来实现，二者常常同时出现，导致很多人误以为他俩是一个东西。

深搜，是逻辑意义上的算法，递归，是一种物理意义上的实现，它和迭代(iteration)是对应的。深搜，可以用递归来实现，也可以用栈来实现；而递归，一般总是用来实现深搜。可以说，递归一定是深搜，深搜不一定用递归。

递归有两种加速策略，一种是 剪枝**(prunning)**，对中间结果进行判断，提前返回；一种是缓存，缓存中间结果，防止重复计算，用空间换时间。

其实，递归+缓存，就是 memorization。所谓**memorization**（翻译为备忘录法，见第 ??? 节，就是"top-down with cache"（自顶向下+缓存），它是Donald Michie 在1968年创造的术语，表示一种优化技术，在top-down 形式的程序中，使用缓存来避免重复计算，从而达到加速的目的。

memorization 不一定用递归，就像深搜不一定用递归一样，可以在迭代(iterative)中使用 memorization 。递归也不一定用 **memorization**，可以用memorization来加速，但不是必须的。只有当递归使用了缓存，它才是 memorization 。

既然递归一定是深搜，为什么很多书籍都同时使用这两个术语呢？在递归味道更浓的地方，一般用递归这个术语，在深搜更浓的场景下，用深搜这个术语，读者心里要弄清楚他俩大部分时候是一回事。在单链表、二叉树等递归数据结构上，递归的味道更浓，这时用递归这个术语；在图、隐式图等数据结构上，深搜的味道更浓，这时用深搜这个术语。

本章主要讲分治法。

Pow(x,n)

描述

Implement `pow(x, n)` .

分析

二分法，$x^n = x^{n/2} \times x^{n/2} \times x^{n\%2}$

代码

```
// Pow(x, n)
// 二分法，$x^n = x^{n/2} * x^{n/2} * x^{n\%2}$
// 时间复杂度O(logn)，空间复杂度O(1)
public class Solution {
    public double myPow(double x, int n) {
        if (n < 0) return 1.0 / power(x, -n);
        else return power(x, n);
    }
    private static double power(double x, int n) {
        if (n == 0) return 1;
        double v = power(x, n / 2);
        if (n % 2 == 0) return v * v;
        else return v * v * x;
    }
}
```

相关题目

- Sqrt(x)

Sqrt(x)

描述

Implement int `sqrt(int x)`.

Compute and return the square root of `x`.

分析

二分查找

代码

```
// Plus One
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int[] plusOne(int[] digits) {
        return add(digits, 1);
    }
    private static int[] add(int[] digits, int digit) {
        int c = digit;  // carry, 进位

        for (int i = digits.length - 1; i >= 0; --i) {
            digits[i] += c;
            c = digits[i] / 10;
            digits[i] %= 10;
        }

        if (c > 0) { // assert (c == 1)
            int[] tmp = new int[digits.length + 1];
            System.arraycopy(digits, 0, tmp, 1, digits.length);
            tmp[0] = c;
            return tmp;
        } else {
            return digits;
        }
    }
};
```

相关题目

- Pow(x)

本章主要讲贪心法。

Jump Game

描述

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.

分析

由于每层最多可以跳`A[i]`步，也可以跳0或1步，因此如果能到达最高层，则说明每一层都可以到达。有了这个条件，说明可以用贪心法。

思路一：正向，从0出发，一层一层网上跳，看最后能不能超过最高层，能超过，说明能到达，否则不能到达。

思路二：逆向，从最高层下楼梯，一层一层下降，看最后能不能下降到第0层。

思路三：如果不敢用贪心，可以用动规，设状态为`f[i]`，表示从第0层出发，走到`A[i]`时剩余的最大步数，则状态转移方程为：

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

代码1

```
// Jump Game
// 思路1，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public boolean canJump(int[] nums) {
        int reach = 1; // 最右能跳到哪里
        for (int i = 0; i < reach && reach < nums.length; ++i)
            reach = Math.max(reach,  i + 1 + nums[i]);
        return reach >= nums.length;
    }
}
```

代码2

```
// Jump Game
// 思路2，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public boolean canJump(int[] nums) {
        if (nums.length == 0) return true;
        // 逆向下楼梯，最左能下降到第几层
        int left_most = nums.length - 1;

        for (int i = nums.length - 2; i >= 0; --i)
            if (i + nums[i] >= left_most)
                left_most = i;

        return left_most == 0;
    }
}
```

代码3

```
// Jump Game
// 思路三，动规，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public boolean canJump(int[] nums) {
        int[] f = new int[nums.length];
        for (int i = 1; i < nums.length; i++) {
            f[i] = Math.max(f[i - 1], nums[i - 1]) - 1;
            if (f[i] < 0) return false;;
        }
        return f[nums.length - 1] >= 0;
    }
}
```

相关题目

- Jump Game II

Jump Game II

描述

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.)

分析

贪心法。

代码1

```
// Jump Game II
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int jump(int[] nums) {
        int step = 0; // 最小步数
        int left = 0;
        int right = 0;  // [left, right]是当前能覆盖的区间
        if (nums.length == 1) return 0;

        while (left <= right) { // 尝试从每一层跳最远
            ++step;
            final int old_right = right;
            for (int i = left; i <= old_right; ++i) {
                int new_right = i + nums[i];
                if (new_right >= nums.length - 1) return step;

                if (new_right > right) right = new_right;
            }
            left = old_right + 1;
        }
        return 0;
    }
}
```

代码2

```
// Jump Game II
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int jump(int[] nums) {
        int result = 0;
        // the maximum distance that has been reached
        int last = 0;
        // the maximum distance that can be reached by using "ret+1" steps
        int cur = 0;
        for (int i = 0; i < nums.length; ++i) {
            if (i > last) {
                last = cur;
                ++result;
            }
            cur = Math.max(cur, i + nums[i]);
        }

        return result;
    }
}
```

相关题目

- Jump Game

Best Time to Buy and Sell Stock

描述

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.

分析

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` 。

代码

```
// Best Time to Buy and Sell Stock
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) return 0;
        int profit = 0;  // 差价，也就是利润
        int cur_min = prices[0]; // 当前最小

        for (int i = 1; i < prices.length; i++) {
            profit = Math.max(profit, prices[i] - cur_min);
            cur_min = Math.min(cur_min, prices[i]);
        }
        return profit;
    }
}
```

相关题目

- Best Time to Buy and Sell Stock II
- Best Time to Buy and Sell Stock III

Best Time to Buy and Sell Stock II

描述

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).

分析

贪心法，低进高出，把所有正的价格差价相加起来。

把原始价格序列变成差分序列，本题也可以做是最大 `m` 子段和， `m=` 数组长度。

代码

```
// Best Time to Buy and Sell Stock II
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int maxProfit(int[] prices) {
        int sum = 0;
        for (int i = 1; i < prices.length; i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) sum += diff;
        }
        return sum;
    }
}
```

相关题目

- Best Time to Buy and Sell Stock
- Best Time to Buy and Sell Stock III

Longest Substring Without Repeating Characters

描述

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for `"abcabcbb"` is `"abc"`, which the length is 3. For `"bbbbb"` the longest substring is `"b"`, with the length of 1.

分析

假设子串里含有重复字符，则父串一定含有重复字符，单个子问题就可以决定父问题，因此可以用贪心法。跟动规不同，动规里，单个子问题只能影响父问题，不足以决定父问题。

从左往右扫描，当遇到重复字母时，以上一个重复字母的`index+1`，作为新的搜索起始位置，直到最后一个字母，复杂度是`O(n)`。如下图所示。

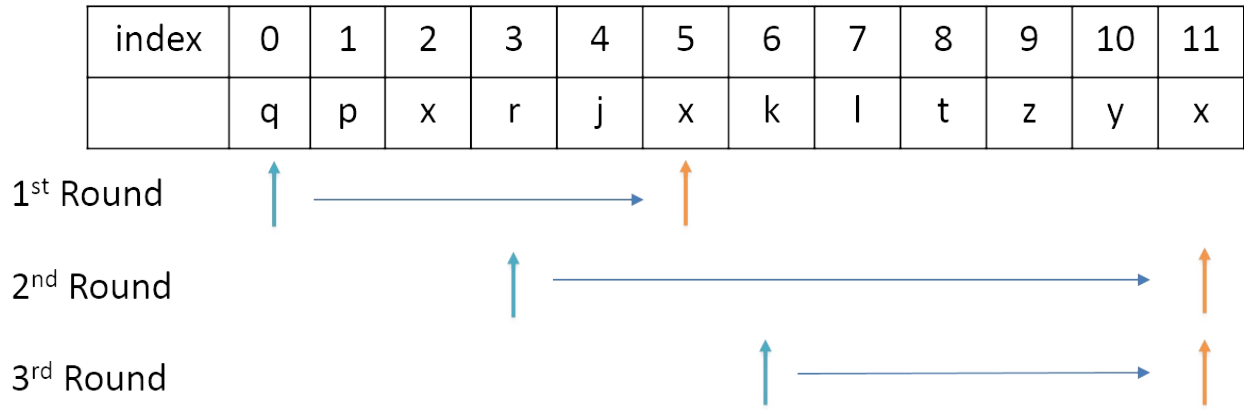


Figure: 不含重复字符的最长子串

代码

```java
// Longest Substring Without Repeating Characters
// 时间复杂度O(n)，空间复杂度O(1)
// 考虑非字母的情况
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        final int ASCII_MAX = 255;
        int[] last = new int[ASCII_MAX]; // 记录字符上次出现过的位置
        int start = 0; // 记录当前子串的起始位置

        Arrays.fill(last, -1); // 0也是有效位置，因此初始化为-1
        int max_len = 0;
        for (int i = 0; i < s.length(); i++) {
            if (last[s.charAt(i)] >= start) {
                max_len = Math.max(i - start, max_len);
                start = last[s.charAt(i)] + 1;
            }
            last[s.charAt(i)] = i;
        }
        return Math.max((int)s.length() - start, max_len);  // 别忘了最后一次，例如"abcd"
    }
}
```

Container With Most Water

描述

Given `n` non-negative integers $a_1, a_2, ..., a_n$, where each represents a point at coordinate (i, a_i). n vertical lines are drawn such that the two endpoints of line `i` is at (i, a_i) and `(i, 0)`. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

分析

每个容器的面积，取决于最短的木板。

代码

```
// Container With Most Water
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int maxArea(int[] height) {
        int start = 0;
        int end = height.length - 1;
        int result = Integer.MIN_VALUE;
        while (start < end) {
            int area = Math.min(height[end], height[start]) * (end - start);
            result = Math.max(result, area);
            if (height[start] <= height[end]) {
                start++;
            } else {
                end--;
            }
        }
        return result;
    }
}
```

相关题目

- Trapping Rain Water
- Largest Rectangle in Histogram

Patching Array

描述

Given a sorted positive integer array `nums` and an integer `n`, add/patch elements to the array such that any number in range `[1, n]` inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

Example 1:

`nums = [1, 3], n = 6`, return 1.

Combinations of `nums` are `[1], [3], [1,3]`, which form possible sums of: `1, 3, 4`.

Now if we add/patch `2` to nums, the combinations are: `[1], [2], [3], [1,3], [2,3], [1,2,3]`.

Possible sums are `1, 2, 3, 4, 5, 6`, which now covers the range `[1, 6]`.

So we only need 1 patch.

Example 2:

`nums = [1, 5, 10], n = 20`, return `2`.

The two patches can be `[2, 4]`.

Example 3:

`nums = [1, 2, 2], n = 5`, return 0.

分析

首先可以确定的是，

- `nums` 中必然包含1，如果不包含1，那么 `[1,n]` 这个范围中的1就没法实现
- 其次数组中的元素不能重复使用，如果允许重复使用，那么把1重复多次，就可以组成任意整数。

令 `miss` 为 `[0,n]` 中缺少的最小整数，意味着我们可以实现 `[0,miss)` 范围内的任意整数。

1. 如果数组中有某个整数 `x<=miss`，那么我们可以把 `[0,miss)` 区间的所有整数加上 `x`，区间变成了 `[x, miss+x)`，由于区间 `[0,miss)` 和 `[x, miss+x)` 重叠，两个区间可以无缝连接起来，意味着我们可以把区间 `[0,miss)` 扩展到 `[0, miss+x)`。
2. 如果数组中不存在小于或等于 `miss` 的元素，则区间 `[0,miss)` 和 `[x, miss+x)` 脱节了，连不起来。此时我们需要添加一个数，最大限度的扩展区间 `[0, miss)`。那添加哪个数呢？当然是添加 `miss` 本身，这样区间 `[0,miss)` 和 `[miss, miss+miss)` 恰好可以无缝拼接。

举个例子，令 `nums=[1, 2, 4, 13, 43]`，`n=100`，我们需要让 `[1,100]` 内的数都能够组合出来。

使用数字 `1,2,4`，我们可以组合出 `[0, 8)` 内的所有数，但无法组合出8，由于下一个数是13，比8大，根据规则2，我们添加8，把区间从 `[0,8)` 扩展到 `[0,16)`。

下一个数是13，比16小，根据规则1，我们可以把区间从 `[0,16)` 扩展到 `[0,29)`。

下一个数是43，比29大，根据规则2，添加29，把区间从 `[0,29)` 扩大到 `[0,58)` 。

由于43比58小，根据规则1，可以把区间从 `[0,58)` 扩展到 `[0,101)` ，刚好覆盖了 `[1,100]` 内的所有数。

最终结果是添加2个数，8和29，就可以组合出 `[1,100]` 内的所有整数。

参考资料：https://leetcode.com/discuss/82822/solution-explanation

代码

```
// Patching Array
// Time complexity: O(n), Space complexity: O(1)
public class Solution {
    public int minPatches(int[] nums, int n) {
        long miss = 1;
        int added = 0;
        int i = 0;
        while (miss
```

动态规划

本章主要讲各种动态规划题。

Triangle

描述

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.

分析

设状态为 `f(i, j)` ，表示从从位置 `(i,j)` 出发，路径的最小和，则状态转移方程为

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

代码

```
// Triangle
// 时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        for (int i = triangle.size() - 2; i >= 0; --i)
            for (int j = 0; j < i + 1; ++j) {
                int old = triangle.get(i).get(j);
                triangle.get(i).set(j, old + Math.min(triangle.get(i + 1).get(j),
                        triangle.get(i + 1).get(j + 1)));
            }

        return triangle.get(0).get(0);
    }
}
```

Maximum Subarray

描述

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`.

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：

$f[j] = \max\{f[j-1] + S[j], S[j]\}, 1 \le j \le n$

$target = \max\{f[j]\}, 1 \le j \le n$

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(nlog 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

动规

```
// Maximum Subarray
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int maxSubArray(int[] nums) {
        int maxLocal = nums[0];
        int global = nums[0];
        for (int i = 1; i < nums.length; ++i) {
            maxLocal = Math.max(nums[i], nums[i] + maxLocal);
            global = Math.max(global, maxLocal);
        }
        return global;
    }
}
```

思路5

```
// Maximum Subarray
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int maxSubArray(int[] nums) {
        return mcss(nums, 0, nums.length);
    }
    // 思路5，求最大连续子序列和
    private static int mcss(int[] nums, int begin, int end) {
        final int n = end - begin;
        int[] sum = new int[n + 1];  // 前n项和

        int result = Integer.MIN_VALUE;
        int cur_min = sum[0];
        for (int i = 1; i <= n; i++) {
            sum[i] = sum[i - 1] + nums[begin  + i - 1];
        }
        for (int i = 1; i <= n; i++) {
            result = Math.max(result, sum[i] - cur_min);
            cur_min = Math.min(cur_min, sum[i]);
        }
        return result;
    }
}
```

相关题目

- Maximum Subarray
- Binary Tree Maximum Path Sum

Maximum Product Subarray

描述

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`.

分析

这题跟“最大连续子序列和”非常类似，只不过变成了“最大连续子序列积”，所以解决思路也很类似。

仅仅有一个小细节需要注意，就是负负得正，两个负数的乘积是正数，因此我们不仅要跟踪最大值，还要跟踪最小值。

动规

```
// maximum-product-subarray
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int maxProduct(int[] nums) {
        int maxLocal = nums[0];
        int minLocal = nums[0];
        int global = nums[0];

        for(int i = 1; i < nums.length; i++){
            int temp = maxLocal;
            maxLocal = Math.max(Math.max(nums[i] * maxLocal, nums[i]), nums[i] * minLoc
al);
            minLocal = Math.min(Math.min(nums[i] * temp, nums[i]), nums[i] * minLocal);
            global = Math.max(global, maxLocal);
        }
        return global;
    }
}
```

相关题目

- Maximum Subarray
- Binary Tree Maximum Path Sum

Longest Increasing Subsequence

描述

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example,

Given `[10, 9, 2, 5, 3, 7, 101, 18]`,

The longest increasing subsequence is `[2, 3, 7, 101]`, therefore the length is `4`. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in `O(n^2)` complexity.

Follow up: Could you improve it to O(n log n) time complexity?

解法1 动规

这是一个多阶段决策问题，求最长，是一个最优化问题，用 BFS, 贪心或DP。

如果用BFS，首先用数组中的所有元素作为根节点，形成n颗树，每棵树开始往下扩展，出现逆序则终止，最后计算每棵树的高度，取最大，就是最终结果。算法复杂度为 `O(n!)`。

本题中，一个节点往下扩展的时候，没有一个确定的准则，让你走哪些边，本题不具有贪心选择性质，因此不能用贪心法。

我们来尝试用DP来解决这题。最重要的是要定义出状态。首先从状态扩展这方面看，对于数组中的一个元素，它往后走，凡是比它大的元素，都可以作为下一步，因此这里找不到突破口。

我们换一个角度，从结果来入手，我们要求的最长递增子序列，一个递增子序列，肯定是有首尾两个端点的，假设我们定义 f[i] 为以第 `i` 个元素为起点的最长递增子序列，那么 `f[i]` 和 `f[j]` 之间没有必然联系，这个状态不好用。

假设定义 `f[i]` 为以第 `i` 个元素为终点的最长递增子序列，那么如果 `i<j` 且 `nums[i]<nums[j]`，则 `f[i]` 一定是 `f[j]` 的前缀。这个状态能表示子问题之间的关系，可以接着深入下去。

现在正式开始定义，我们定义状态 `f[i]` 为第 `i` 个元素为终点的最长递增子序列的长度，那么状态转移方程是

`f[j] = max{f[i], 0 <= i < j && f[i] < f[j]} + 1`

有了状态和状态转移方程，代码就不难写了。

```java
// Longest Increasing Subsequence
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        // f[i]表示以i结尾的最长递增子序列的长度
        int[] f = new int[nums.length];
        Arrays.fill(f, 1);
        int global = 1;
        for (int j = 1; j < nums.length; ++j) {
            for (int i = 0; i < j; ++i) {
                if (nums[i] < nums[j]) {
                    f[j] = Math.max(f[j], f[i] + 1);
                }
            }
            global = Math.max(global, f[j]);
        }
        return global;
    }
}
```

解法2 Insert Position

本题最后有一个进阶问题，能不能 `O(n log n)` 解决？有。

维护一个单调递增序列，遍历数组，二分查找每一个数在单调序列中的位置，然后替换之。

```java
// Longest Increasing Subsequence
// 时间复杂度O(nlogn)，空间复杂度O(n)
public class Solution {
    public int lengthOfLIS(int[] nums) {
        ArrayList<Integer> lis = new ArrayList<>();
        for (int x : nums) {
            int insertPos = lowerBound(lis, 0, lis.size(), x);
            if (insertPos >= lis.size()) {
                lis.add(x);
            } else {
                lis.set(insertPos, x);
            }
        }
        return lis.size();
    }
    private static int lowerBound (ArrayList<Integer> A,
                                   int first, int last, int target) {
        while (first != last) {
            int mid = first + (last - first) / 2;
            if (target > A.get(mid)) first = ++mid;
            else                 last = mid;
        }

        return first;
    }
}
```

Palindrome Partitioning II

描述

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

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

For example, given `s = "aab"`,

Return 1 since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

分析

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

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

这是一个二维函数，实际写代码比较麻烦。

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

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

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

定义状态`P[i][j] = true if [i,j]为回文`，那么

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

代码

```java
// Palindrome Partitioning II
// 时间复杂度O(n^2)，空间复杂度O(n^2)
public class Solution {
    public int minCut(String s) {
        final int n = s.length();
        int[] f = new int[n+1];
        boolean[][] p = new boolean[n][n];
        //the worst case is cutting by each char
        for (int i = 0; i = 0; i--) {
            for (int j = i; j < n; j++) {
                if (s.charAt(i) == s.charAt(j) &&
                        (j - i < 2 || p[i + 1][j - 1])) {
                    p[i][j] = true;
                    f[i] = Math.min(f[i], f[j + 1] + 1);
                }
            }
        }
        return f[0];
    }
}
```

相关题目

- Palindrome Partitioning

Maximal Rectangle

描述

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

分析

无

代码

```java
// Maximal Rectangle
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public int maximalRectangle(char[][] matrix) {
        if (matrix.length == 0)  return 0;

        final int m = matrix.length;
        final int n = matrix[0].length;
        int[] H = new int[n];
        int[] L = new int[n];
        int[] R = new int[n];
        Arrays.fill(R, n);

        int ret = 0;
        for (int i = 0; i < m; ++i) {
            int left = 0, right = n;
            // calculate L(i, j) from left to right
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == '1') {
                    ++H[j];
                    L[j] = Math.max(L[j], left);
                } else {
                    left = j+1;
                    H[j] = 0; L[j] = 0; R[j] = n;
                }
            }
            // calculate R(i, j) from right to left
            for (int j = n-1; j >= 0; --j) {
                if (matrix[i][j] == '1') {
                    R[j] = Math.min(R[j], right);
                    ret = Math.max(ret, H[j]*(R[j]-L[j]));
                } else {
                    right = j;
                }
            }
        }
        return ret;
    }
}
```

Best Time to Buy and Sell Stock III

描述

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).

分析

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

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` ，参考代码：https://gist.github.com/soulmachine/5906637

代码

```java
// Best Time to Buy and Sell Stock III
// 时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) return 0;

        final int n = prices.length;
        int[] f = new int[n];
        int[] g = new int[n];

        for (int i = 1, valley = prices[0]; i < n; ++i) {
            valley = Math.min(valley, prices[i]);
            f[i] = Math.max(f[i - 1], prices[i] - valley);
        }

        for (int i = n - 2, peak = prices[n - 1]; i >= 0; --i) {
            peak = Math.max(peak, prices[i]);
            g[i] = Math.max(g[i], peak - prices[i]);
        }

        int max_profit = 0;
        for (int i = 0; i < n; ++i)
            max_profit = Math.max(max_profit, f[i] + g[i]);

        return max_profit;
    }
}
```

相关题目

- Best Time to Buy and Sell Stock
- Best Time to Buy and Sell Stock II

Best Time to Buy and Sell Stock IV

描述

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 `k` transactions.

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

分析

设两个状态，`global[i][j]` 表示i天前最多可以进行j次交易的最大利润，`local[i][j]` 表示i天前最多可以进行j次交易，且在第i天进行了第j次交易的最大利润。状态转移方程如下：

```
local[i][j] = max(global[i-1][j-1] + max(diff,0), local[i-1][j]+diff)
global[i][j] = max(local[i][j], global[i-1][j])
```

关于 `global` 的状态转移方程比较简单，不断地和已经计算出的local进行比较，把大的保存在global中。

关于 `local` 的状态转移方程，取下面二者中较大的一个：

- 全局前 `i-1` 天进行了 `j-1` 次交易，然后然后加上今天的交易产生的利润（如果赚钱就交易，不赚钱就不交易，什么也不发生，利润是0）
- 局部前 `i-1` 天进行了 `j` 次交易，然后加上今天的差价（ `local[i-1][j]` 是第 `i-1` 天卖出的交易，它加上diff后变成第 `i` 天卖出，并不会增加交易次数。无论 `diff` 是正还是负都要加上，否则就不满足 `local[i][j]` 必须在最后一天卖出的条件了）

注意，当 `k` 大于数组的大小时，上述算法将变得低效，此时可以改为不限交易次数的方式解决，即等价于"Best Time to Buy and Sell Stock II"。

解法1

```java
// Best Time to Buy and Sell Stock IV
// Time Complexity: O(nk), Space Complexity: O(nk)
public class Solution {
    public int maxProfit(final int k, final int[] prices) {
        if (prices.length < 2 || k < 1) return 0;
        if (k >= prices.length) return maxProfit(prices);

        final int[][] local = new int[prices.length][k + 1];
        final int[][] global = new int[prices.length][k + 1];

        for (int i = 1; i < prices.length; i++) {
            final int diff = prices[i] - prices[i - 1];
            for (int j = 1; j < k+1; j++) {
                local[i][j] = Math.max(
                        global[i - 1][j - 1] + Math.max(diff, 0),
                        local[i - 1][j] + diff);
                global[i][j] = Math.max(global[i - 1][j], local[i][j]);
            }
        }

        return global[prices.length - 1][k];
    }

    // Best Time to Buy and Sell Stock II
    public static int maxProfit(final int[] prices) {
        int sum = 0;
        for (int i = 1; i < prices.length; i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) sum += diff;
        }
        return sum;
    }
}
```

解法2 最长m段子段和

Best Time to Buy and Sell Stock with Cooldown

描述

Almost the ame as Best Time to Buy and Sell Stock II but with one restriction: after you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day).

Example:

```
prices = [1, 2, 3, 0, 2]
maxProfit = 3
transactions = [buy, sell, cooldown, buy, sell]
```

分析

这题比Best Time to Buy and Sell Stock II多了一个cooldown的条件，就变得麻烦多了。这题是一个多阶段优化问题，首先范围缩小到广搜，贪心或者动规。因为每步之间互相牵连，贪心显然不行。广搜固然可以，不过是 `O(2^n)` 复杂度，所以我们先考虑用动规。

对于每一天，有三种动作，buy, sell, cooldown, sell 和 cooldown 可以合并成一种状态，因为手里最终没有股票。最终需要的结果是 sell，即手里股票卖了获得最大利润。我们可以用两个数组来记录当前持股和未持股的状态，令 `sell[i]` 表示第i天未持股时，获得的最大利润， `buy[i]` 表示第i天持有股票时，获得的最大利润。

对于 `sell[i]` ，最大利润有两种可能，一是今天没动作跟昨天未持股状态一样，二是今天卖了股票，所以状态转移方程如下：

`sell[i] = max{sell[i - 1], buy[i-1] + prices[i]}`

对于 `buy[i]` ，最大利润有两种可能，一是今天没动作跟昨天持股状态一样，二是前天卖了股票，今天买了股票，因为 cooldown 只能隔天交易，所以今天买股票要追溯到前天的状态。状态转移方程如下：

`buy[i] = max{buy[i-1], sell[i-2] - prices[i]}`

最终我们要求的结果是 `sell[n - 1]` ，表示最后一天结束时，手里没有股票时的最大利润。

这个算法的空间复杂度是 `O(n)` ，不过由于 `sell[i]` 仅仅依赖前一项， `buy[i]` 仅仅依赖前两项，所以可以优化到 `O(1)` ，具体见第二种代码实现。

代码1 O(n)空间

```java
// Best Time to Buy and Sell Stock with Cooldown
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) return 0;

        int[] sell = new int[prices.length];
        int[] buy = new int[prices.length];
        sell[0] = 0;
        buy[0] = -prices[0];

        for (int i = 1; i < prices.length; ++i) {
            sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
            buy[i] = Math.max(buy[i - 1], (i > 1 ? sell[i - 2] : 0) - prices[i]);
        }
        return sell[prices.length - 1];
    }
}
```

代码2 O(1)空间

```java
// Best Time to Buy and Sell Stock with Cooldown
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) return 0;

        int curSell = 0;   // sell[i]
        int prevSell = 0;  // sell[i-2]
        int buy = -prices[0]; // buy[i]

        for (int i = 1; i < prices.length; ++i) {
            final int tmp = curSell;
            curSell = Math.max(curSell, buy + prices[i]);
            buy = Math.max(buy, (i > 1 ? prevSell : 0) - prices[i]);
            prevSell = tmp;
        }
        return curSell;
    }
}
```

Interleaving String

描述

Given `s1, s2, s3`, find whether `s3` is formed by the interleaving of `s1` and `s2`.

For example, Given: `s1 = "aabcc", s2 = "dbbca"`,

When `s3 = "aadbbcbcac"`, return true.

When `s3 = "aadbbbaccc"`, return false.

分析

设状态`f[i][j]`，表示`s1[0,i]`和`s2[0,j]`，匹配`s3[0, i+j]`。如果s1的最后一个字符等于s3的最后一个字符，则`f[i][j]=f[i-1][j]`；如果s2的最后一个字符等于s3的最后一个字符，则`f[i][j]=f[i][j-1]`。因此状态转移方程如下：

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

递归

```
// Interleaving String
// 递归，会超时，仅用来帮助理解
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if (s3.length() != s1.length() + s2.length())
            return false;
        if (s1.isEmpty() || s2.isEmpty()) return true;

        return isInterleave(s1, 0, s1.length(),
                s2, 0, s2.length(), s3, 0, s3.length());
    }

    private static boolean isInterleave(String s1, int begin1, int end1,
                                        String s2, int begin2, int end2,
                                        String s3, int begin3, int end3) {
        if (begin3 == end3)
            return begin1 == end1 && begin2 == end2;

        return (begin1 < end1 && s1.charAt(begin1) == s3.charAt(begin3) &&
                isInterleave(s1, begin1 + 1, end1, s2, begin2, end2,
                        s3, begin3 + 1, end3)) ||
                (begin2 < end2 && s2.charAt(begin2) == s3.charAt(begin3) &&
                        isInterleave(s1, begin1, end1,
                                s2, begin2 + 1, end2, s3, begin3 + 1, end3));
    }
}
```

动规

```
// Interleaving String
// 二维动规，时间复杂度O(n^2)，空间复杂度O(n^2)
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if (s3.length() != s1.length() + s2.length())
            return false;

        boolean[][] f = new boolean[s1.length() + 1][s2.length() + 1];
        for (int i = 0; i < s1.length() + 1; ++i)
            Arrays.fill(f[i], true);

        for (int i = 1; i
```

动规+滚动数组

```java
// Interleaving String
// 二维动规+滚动数组，时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if (s1.length() + s2.length() != s3.length())
            return false;

        if (s1.length() < s2.length())
            return isInterleave(s2, s1, s3);

        boolean[] f = new boolean[s2.length() + 1];
        Arrays.fill(f, true);

        for (int i = 1; i
```

Scramble String

描述

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

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

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

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

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

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

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

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

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

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

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

分析

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

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

加缓存，可以用数组或HashMap。本题维数较高，用HashMap，`map`和`unordered_map`均可。

既然可以用记忆化搜索，这题也一定可以用动规。设状态为`f[n][i][j]`，表示长度为`n`，起点为`s1[i]`和起点为`s2[j]`两个字符串是否互为scramble，则状态转移方程为

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

递归

```
// Scramble String
// 递归，会超时，仅用来帮助理解
// 时间复杂度O(n^6)，空间复杂度O(1)
public class Solution {
    public boolean isScramble(String s1, String s2) {
        return isScramble(s1, 0, s1.length(), s2, 0);
    }
    private static boolean isScramble(String s1, int begin1, int end1,
                                      String s2, int begin2) {
        final int length = end1 - begin1;
        final int end2 = begin2 + length;

        if (length == 1) return s1.charAt(begin1) == s2.charAt(begin2);

        for (int i = 1; i < length; ++i)
            if ((isScramble(s1, begin1, begin1 + i, s2, begin2)
                    && isScramble(s1, begin1 + i, end1, s2, begin2 + i))
                    || (isScramble(s1, begin1, begin1 + i, s2, end2 - i)
                    && isScramble(s1, begin1 + i, end1, s2, begin2)))
                return true;

        return false;
    }
}
```

动规

```
// Scramble String
// 动规，时间复杂度O(n^3)，空间复杂度O(n^3)
public class Solution {
    public boolean isScramble(String s1, String s2) {
        final int N = s1.length();
        if (N != s2.length()) return false;

        // f[n][i][j]，表示长度为n，起点为s1[i]和
        // 起点为s2[j]两个字符串是否互为scramble
        boolean[][][] f = new boolean[N+1][N][N];

        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                f[1][i][j] = s1.charAt(i) == s2.charAt(j);

        for (int n = 1; n
```

递归+剪枝

```
// Scramble String
// 递归+剪枝
// 时间复杂度O(n^6)，空间复杂度O(1)
public class Solution {
    public boolean isScramble(String s1, String s2) {
        return isScramble(s1, 0, s1.length(), s2, 0);
    }
    private static boolean isScramble(String s1, int begin1, int end1,
                                      String s2, int begin2) {
        final int length = end1 - begin1;
        final int end2 = begin2 + length;
        if (length == 1) return s1.charAt(begin1) == s2.charAt(begin2);

        // 剪枝，提前返回
        int[] A = new int[26]; // 每个字符的计数器
        for(int i = 0; i < length; i++) A[s1.charAt(begin1+i)-'a']++;
        for(int i = 0; i < length; i++) A[s2.charAt(begin2+i)-'a']--;
        for(int i = 0; i < 26; i++) if (A[i] != 0) return false;

        for (int i = 1; i < length; ++i)
            if ((isScramble(s1, begin1, begin1 + i, s2, begin2)
                    && isScramble(s1, begin1 + i, end1, s2, begin2 + i))
                    || (isScramble(s1, begin1, begin1 + i, s2, end2 - i)
                    && isScramble(s1, begin1 + i, end1, s2, begin2)))
                return true;

        return false;
    }
}
```

备忘录法

```
// Scramble String
// 递归+map做cache
// 时间复杂度O(n^3)，空间复杂度O(n^3), TLE
public class Solution {
    public boolean isScramble(String s1, String s2) {
        cache.clear();
        return isScramble(s1, 0, s1.length(), s2, 0);
    }

    final private HashMap cache = new HashMap<>();

    private boolean isScramble(String s1, int begin1, int end1,
                               String s2, int begin2) {
        final int length = end1 - begin1;
        final int end2 = begin2 + length;

        if (length == 1) return s1.charAt(begin1) == s2.charAt(begin2);

        for (int i = 1; i < length; ++i)
            if ((getOrUpdate(s1, begin1, begin1 + i, s2, begin2)
                    && getOrUpdate(s1, begin1 + i, end1, s2, begin2 + i))
                    || (getOrUpdate(s1, begin1, begin1 + i, s2, end2 - i)
                    && getOrUpdate(s1, begin1 + i, end1, s2, begin2)))
                return true;

        return false;
    }

    boolean getOrUpdate(String s1, int begin1, int end1,
                        String s2, int begin2) {
        Triple key = new Triple(begin1, end1, begin2);
        if (!cache.containsKey(key)) {
            boolean result = isScramble(s1, begin1, end1, s2, begin2);
            cache.put(key, result);
            return result;
        } else {
            return cache.get(key);
        }
    }
    static class Triple {
        private int i;
        private int j;
        private int k;

        public Triple(int i, int j, int k) {
            this.i = i;
            this.j = j;
            this.k = k;
        }
        @Override
        public int hashCode() {
            int hash = 0;
```

```
            hash = i * 31 + j;
            hash = hash * 31 * k;
            return hash;
        }
        @Override
        public boolean equals(Object other) {
            if (this == other) return true;
            if (this.hashCode() != other.hashCode()) return false;
            if (!(other instanceof Triple)) return false;
            Triple o = (Triple)other;
            return this.i == o.i && this.j == o.j && this.k == o.k;
        }
    }
}
```

Minimum Path Sum

描述

Given a `m × 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

分析

跟第 ??? 节 Unique Paths 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：

```
f[i][j]=min(f[i-1][j], f[i][j-1])+grid[i][j]
```

备忘录法

```
// Minimum Path Sum
// 备忘录法
public class Solution {
    public int minPathSum(int[][] grid) {
        final int m = grid.length;
        final int n = grid[0].length;
        this.f = new int[m][n];
        for (int i = 0; i < m; ++i) Arrays.fill(f[i], -1);
        return dfs(grid, m-1, n-1);
    }

    private int dfs(int[][] grid, int x, int y) {
        if (x < 0 || y < 0) return Integer.MAX_VALUE; // 越界，终止条件，注意，不是0

        if (x == 0 && y == 0) return grid[0][0]; // 回到起点，收敛条件

        return Math.min(getOrUpdate(grid, x - 1, y),
                getOrUpdate(grid, x, y - 1)) + grid[x][y];
    }

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

动规

```java
// Minimum Path Sum
// 二维动规
public class Solution {
    public int minPathSum(int[][] grid) {
        final int m = grid.length;
        final int n = grid[0].length;
        if (m == 0) return 0;

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

动规+滚动数组

```java
// Minimum Path Sum
// 二维动规+滚动数组
public class Solution {
    public int minPathSum(int[][] grid) {
        final int m = grid.length;
        final int n = grid[0].length;

        int[] f = new int[n];
        Arrays.fill(f, Integer.MAX_VALUE); // 初始值是 INT_MAX，因为后面用了min函数。
        f[0] = 0;

        for (int i = 0; i < m; i++) {
            f[0] += grid[i][0];
            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] = Math.min(f[j - 1], f[j]) + grid[i][j];
            }
        }
        return f[n - 1];
    }
}
```

相关题目

- Unique Paths
- Unique Paths II

Edit Distance

描述

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:

- Insert a character
- Delete a character
- Replace a character

分析

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`，

1. 如果 `c==d`，则 `f[i][j]=f[i-1][j-1]`；
2. 如果 `c!=d`，
    i. 如果将c替换成d，则 `f[i][j]=f[i-1][j-1]+1`；
    ii. 如果在c后面添加一个d，则 `f[i][j]=f[i][j-1]+1`；
    iii. 如果将c删除，则 `f[i][j]=f[i-1][j]+1`；

动规

```java
// Edit Distance
// 二维动规，时间复杂度O(n*m)，空间复杂度O(n*m)
public class Solution {
    public int minDistance(String word1, String word2) {
        final int n = word1.length();
        final int m = word2.length();
        // 长度为n的字符串，有n+1个隔板
        int[][] f = new int[n+1][m+1];
        for (int i = 0; i <= n; i++)
            f[i][0] = i;
        for (int j = 0; j <= m; j++)
            f[0][j] = j;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1))
                    f[i][j] = f[i - 1][j - 1];
                else {
                    int mn = Math.min(f[i - 1][j], f[i][j - 1]);
                    f[i][j] = 1 + Math.min(f[i - 1][j - 1], mn);
                }
            }
        }
        return f[n][m];
    }
}
```

动规+滚动数组

```java
// Edit Distance
// 二维动规+滚动数组
// 时间复杂度O(n*m)，空间复杂度O(n)
public class Solution {
    public int minDistance(String word1, String word2) {
        if (word1.length() < word2.length())
            return minDistance(word2, word1);

        int[] f = new int[word2.length() + 1];
        int upper_left = 0; // 额外用一个变量记录f[i-1][j-1]

        for (int i = 0; i <= word2.length(); ++i)
            f[i] = i;

        for (int i = 1; i <= word1.length(); ++i) {
            upper_left = f[0];
            f[0] = i;

            for (int j = 1; j <= word2.length(); ++j) {
                int upper = f[j];

                if (word1.charAt(i - 1) == word2.charAt(j - 1))
                    f[j] = upper_left;
                else
                    f[j] = 1 + Math.min(upper_left, Math.min(f[j], f[j - 1]));

                upper_left = upper;
            }
        }

        return f[word2.length()];
    }
}
```

Decode Ways

描述

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.

分析

跟第???节Climbing Stairs很类似，不过多加一些判断逻辑。

代码

```
// Decode Ways
// 动规，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int numDecodings(String s) {
        if (s.isEmpty() || s.charAt(0) == '0') return 0;

        int prev = 0;
        int cur = 1;
        // 长度为n的字符串，有 n+1个阶梯
        for (int i = 1; i <= s.length(); ++i) {
            if (s.charAt(i-1) == '0') cur = 0;

            if (i < 2 || !(s.charAt(i - 2) == '1' ||
                    (s.charAt(i - 2) == '2' && s.charAt(i - 1) <= '6')))
                prev = 0;

            int tmp = cur;
            cur = prev + cur;
            prev = tmp;
        }
        return cur;
    }
}
```

相关题目

- Climbing Stairs

Distinct Subsequences

描述

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.

分析

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]`，此时 `f(i,j)=f(i-1,j)+f(i-1, j-1)`。

代码

```
// Distinct Subsequences
// 二维动规+滚动数组
// 时间复杂度O(m*n)，空间复杂度O(n)
public class Solution {
    public int numDistinct(String s, String t) {
        int[] f = new int[t.length() + 1];
        f[0] = 1;
        for (int i = 0; i < s.length(); ++i) {
            for (int j = t.length() - 1; j >= 0; --j) {
                f[j + 1] += s.charAt(i) == t.charAt(j) ? f[j] : 0;
            }
        }

        return f[t.length()];
    }
}
```

Word Break

描述

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"`.

分析

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

`f(i) = any_of(f(j) && s[j,i] in dict), 0 <= j < i`

深搜

```
// Word Break
// 深搜，超时
// 时间复杂度O(2^n)，空间复杂度O(n)
class Solution {
    public boolean wordBreak(String s, Set<String> dict) {
        return dfs(s, dict, 0, 1);
    }
    private static boolean dfs(String s, Set<String> dict,
                    int start, int cur) {
        if (cur == s.length()) {
            return dict.contains(s.substring(start, cur));
        }
        if (dfs(s, dict, start, cur+1)) return true; // no cut
        if (dict.contains(s.substring(start, cur))) // cut here
            if (dfs(s, dict, cur+1, cur+1)) return true;
        return false;
    }
}
```

动规

```
// Word Break
// 动规，时间复杂度O(n^2)，空间复杂度O(n)
class Solution {
    public boolean wordBreak(String s, Set<String> dict) {
        // 长度为n的字符串有n+1个隔板
        boolean[] f = new boolean[s.length() + 1];
        f[0] = true; // 空字符串
        for (int i = 1; i <= s.length(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (f[j] && dict.contains(s.substring(j, i))) {
                    f[i] = true;
                    break;
                }
            }
        }
        return f[s.length()];
    }
}
```

相关题目

- Word Break II

Word Break II

描述

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"]`.

分析

在上一题的基础上，要返回解本身。

代码

```java
// Word Break II
// 动规，时间复杂度O(n^2)，空间复杂度O(n^2)
public class Solution {
    public List<String> wordBreak(String s, Set<String> wordDict) {
        // 长度为n的字符串有n+1个隔板
        boolean[] f = new boolean[s.length() + 1];
        // prev[i][j]为true，表示s[j, i)是一个合法单词，可以从j处切开
        // 第一行未用
        boolean[][] prev = new boolean[s.length() + 1][s.length()];
        f[0] = true;
        for (int i = 1; i <= s.length(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (f[j] && wordDict.contains(s.substring(j, i))) {
                    f[i] = true;
                    prev[i][j] = true;
                }
            }
        }
        List<String> result = new ArrayList<>();
        List<String> path = new ArrayList<>();
        gen_path(s, prev, s.length(), path, result);
        return result;

    }
    // DFS遍历树，生成路径
    private static void gen_path(String s, boolean[][] prev,
                  int cur, List<String> path, List<String> result) {
        if (cur == 0) {
            StringBuilder sb = new StringBuilder();
            for (int i = path.size() - 1; i >= 0; --i)
                sb.append(path.get(i)).append(' ');
            sb.deleteCharAt(sb.length()-1);
            result.add(sb.toString());
        }
        for (int i = 0; i < s.length(); ++i) {
            if (prev[cur][i]) {
                path.add(s.substring(i, cur));
                gen_path(s, prev, i, path, result);
                path.remove(path.size()-1);
            }
        }
    }
}
```

相关题目

- Word Break

Dungeon Game

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN`.

```
+--------+----+--------+
| -2 (K) | -3 |    3   |
+--------+----+--------+
| -5     | -10|    1   |
+--------+----+--------+
| 10     | 30 | -5 (P) |
+--------+----+--------+
```

Notes:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

分析

这是一个多阶段优化问题，有广搜，贪心，动规。这题显然贪心不行，于是范围缩小到广搜和动规。本题求最小健康点数，跟路径长度无关，因此广搜不适合。最后只剩下了动规这个方向。

先定义状态，`f[i][j]` 表示进入 `(i,j)` 这个格子前所需要的最小健康点数。

再考虑状态的初始值，我们发现右下角那个坐标才是初始值，起点 `f[0][0]` 反而不知道，于是大致可以知道，这题是从右下角向左上角来填表。右下角格子，如果该格子的值为负数，那么进入这个格子前骑士需要的健康点数是 `-dungeon[i][j]+1`，即初始值 `f[i][j]=-dungeon[i][j]+1`，如果格子的值是非负数，那么初始值 `f[i][j]=1`。

接下来寻找状态转移方程。从初始值我们可以推测出状态转移是从右下角反向的，可以得到状态转移方程如下：

`f[i][j]=max(1, -dungeon[i][j]+min(f[i+1][j],f[i][j+1])`

代码

```java
// Dungeon Game
// Time Complexity: O(mxn), Space Complexity: O(mxn)
public class Solution {
    public int calculateMinimumHP(int[][] dungeon) {
        final int m = dungeon.length;
        final int n = dungeon[0].length;
        if (m == 0 || n == 0) return 0;

        final int[][] f = new int[m][n];
        f[m-1][n-1] = Math.max(1, -dungeon[m-1][n-1]+1);

        for (int i = m - 2; i >= 0; --i) {
            f[i][n-1] = Math.max(1, -dungeon[i][n-1] + f[i+1][n-1]);
        }
        for (int j = n - 2; j >= 0; --j) {
            f[m-1][j] = Math.max(1, -dungeon[m-1][j] + f[m-1][j+1]);
        }

        for (int i = m - 2; i >= 0; --i) {
            for (int j = n - 2; j >= 0; --j) {
                f[i][j] = Math.max(1, -dungeon[i][j] + Math.min(f[i+1][j], f[i][j+1]));
            }
        }
        return f[0][0];
    }
}
```

House Robber

描述

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

分析

这是一个多阶段最优化问题，且要走到最底部才能知道答案，因此广搜排除，只剩下贪心和动规。贪心明显要排除，只剩下动规。

设状态 `f[i]` 为到位置 `i` 时能抢到的金钱最大和，那么状态转移方程如下：

`f[i]=max(f[i-1], f[i-2] + nums[i])`

其含义是，如果不选择 `i` ，则抢到的钱是 `f[i-1]` ，如果选择 `i` ，则能抢到的钱是 `f[i-2] + nums[i]` 。

解法1

```
// House Robber
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        if (nums.length == 1) return nums[0];

        int[] f = new int[nums.length];
        f[0] = nums[0];
        f[1] = Math.max(nums[0], nums[1]);

        for (int i = 2; i < nums.length; ++i) {
            f[i] = Math.max(f[i-1], f[i-2] + nums[i]);
        }
        return f[nums.length - 1];
    }
}
```

解法2

在状态转移方程中，我们可以发现 `f[i]` 仅仅依赖前两项，因此用两个整数变量即可代替一位数组，将空间复杂度降为 `O(1)` 。

```java
// House Robber
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        if (nums.length == 1) return nums[0];

        int even = nums[0];
        int odd = Math.max(nums[0], nums[1]);

        for (int i = 2; i < nums.length; ++i) {
            if (i % 2 == 0) {
                even = Math.max(even + nums[i], odd);
            } else {
                odd = Math.max(odd + nums[i], even);
            }
        }
        return Math.max(even, odd);
    }
}
```

House Robber II

描述

This time, all houses at this place are **arranged in a circle**.

分析

如果抢劫第一家，则不可以抢最后一家；否则，可以抢最后一家。因此，这个问题就转化成为了两趟动规，可以复用 "House Robber" 的代码。

代码

```
// House Robber II
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int rob(int[] nums) {
        if (nums.length == 1) return nums[0];
        return Math.max(rob1(nums, 0, nums.length - 1),
                rob1(nums, 1, nums.length));
    }

    private static int rob1(int[] nums, int begin, int end) {
        if (nums == null || begin >= end) return 0;
        if ((end - begin) == 1) return nums[begin];

        int even = nums[begin];
        int odd = Math.max(nums[begin], nums[begin + 1]);

        for (int i = begin + 2; i < end; ++i) {
            if ((i-begin) % 2 == 0) {
                even = Math.max(even + nums[i], odd);
            } else {
                odd = Math.max(odd + nums[i], even);
            }
        }
        return Math.max(even, odd);
    }
}
```

House Robber III

描述

All houses in this place **forms a binary tree**. It will automatically contact the police if two directly-linked houses were broken into on the same night.

Example 1:

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

Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.

Example 2:

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

Maximum amount of money the thief can rob = 4 + 5 = 9.

分析

树形动规。设状态 `f(root)` 表示抢劫root为根节点的二叉树，root可抢也可能不抢，能得到的最大金钱， `g(root)` 表示抢劫root为根节点的二叉树，但不抢root，能得到的最大金钱，则状态转移方程为

`f(root) = max{f(root.left) + f(root.right), g(root.left)+g(root.right) + root.val}`

`g(root) = f(root.left) + f(root.right)`

代码

```java
// House Robber III
// Time Complexity: O(n), Space Complexity: O(h)
public class Solution {
    public int rob(TreeNode root) {
        return dfs(root)[0];
    }

    private static int[] dfs(TreeNode root) {
        int[] dp = new int[] {0, 0}; // f, g
        if (root != null) {
            int[] dpL = dfs(root.left);
            int[] dpR = dfs(root.right);
            dp[1] = dpL[0] + dpR[0];
            dp[0] = Math.max(dp[1], dpL[1] + dpR[1] + root.val);
        }
        return dp;
    }
}
```

Range Sum Query - Immutable

描述

Given an integer array `nums`, find the sum of the elements between indices `i` and `j` (`i ≤ j`), inclusive.

Example:

Given `nums = [-2, 0, 3, -5, 2, -1]`

```
sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

Note:

- You may assume that the array does not change.
- There are many calls to sumRange function.

分析

令状态`f[i]`为0到`i`元素之间的和，则状态转移方程为`f[i] = f[i-1] + nums[i]`。`f[i]`本质上是累加和，有了`f[i]`，则范围[i,j]之间的和等于`f[j] - f[i-1]`。

代码

```
// Range Sum Query - Immutable
public class NumArray {
    // Time Complexity: O(n), Space Complexity: O(1)
    public NumArray(int[] nums) {
        this.f = new int[nums.length];
        int sum = 0;
        for (int i = 0; i < nums.length; ++i) {
            sum += nums[i];
            f[i] = sum;
        }
    }

    // Time Complexity: O(1), Space Complexity: O(1)
    public int sumRange(int i, int j) {
        return f[j] - (i == 0 ? 0 : f[i - 1]);
    }
    private final int[] f;
}
```

相关题目

- Range Sum Query 2D - Immutable
- Range Sum Query - Mmutable

Range Sum Query 2D - Immutable

描述

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner `(row1, col1)` and lower right corner `(row2, col2)`.

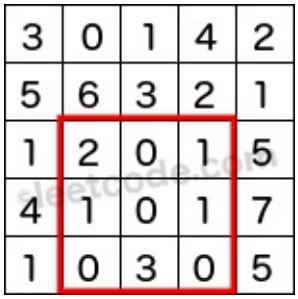


Figure: The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

Given `matrix =`

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

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

Note:

- You may assume that the matrix does not change.
- There are many calls to sumRegion function.
- You may assume that row1 ≤ row2 and col1 ≤ col2.

分析

思路跟一维的类似，建立一个累加和矩阵。令状态 `f[i][j]` 表示从(0,0)到(i,j)的子矩阵的和，则状态转移方程为

`f[i][j] = f[i-1][j] + rowSum`

其中 `rowSum` 是矩阵 `matrix[i][0]` 到 `matrix[i][j]` 这一行的和。

有了 `f[i][j]`，则

`sumRange(i1,j1,i2,j2) = f[i2][j2] + f[i1-1][j1-1] - f[i1-1][j2]-f[i2][j1-1]`

将辅助矩阵 `f[i][j]` 的行数和列数增1，可以简化对矩阵边界的处理。

代码

```
// Range Sum Query 2D - Immutable
public class NumMatrix {
    // Time Complexity: O(n*m), Space Complexity: O(1)
    public NumMatrix(int[][] matrix) {
        final int m = matrix.length;
        final int n = matrix.length > 0 ? matrix[0].length : 0;
        this.f = new int[m + 1][n + 1];

        for (int i = 1; i < m + 1; ++i) {
            int rowSum = 0;
            for (int j = 1; j < n + 1; ++j) {
                f[i][j] += rowSum + matrix[i-1][j-1];
                if (i > 1) {
                    f[i][j] += f[i-1][j];
                }
                rowSum += matrix[i-1][j-1];
            }
        }
    }

    // Time Complexity: O(1), Space Complexity: O(1)
    public int sumRegion(int row1, int col1, int row2, int col2) {
        return f[row2 + 1][col2 + 1] + f[row1][col1] -
                f[row1][col2 + 1] - f[row2 + 1][col1];
    }
    private final int[][] f;
}
```

相关题目

- Range Sum Query - Immutable
- Range Sum Query - Mmutable

图

无向图的节点定义如下：

```
// 无向图的节点
class UndirectedGraphNode {
    int label;
    ArrayList<UndirectedGraphNode> neighbors;
    UndirectedGraphNode(int x) { label = x;}
};
```

Clone Graph

描述

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

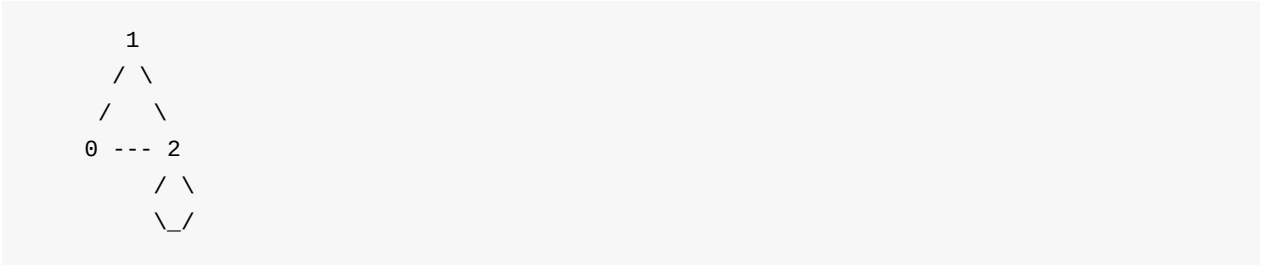
OJ's undirected graph serialization: Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbour 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:



分析

广度优先遍历或深度优先遍历都可以。

DFS

```
// Clone Graph
// DFS，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if(node == null) return null;
        // key is original node，value is copied node
        HashMap<UndirectedGraphNode, UndirectedGraphNode> visited = new HashMap<>();
        clone(node, visited);
        return visited.get(node);
    }
    // DFS
    private static UndirectedGraphNode clone(UndirectedGraphNode node,
                                             HashMap<UndirectedGraphNode,
                                                     UndirectedGraphNode> visited) {
        // a copy already exists
        if (visited.containsKey(node)) return visited.get(node);

        UndirectedGraphNode new_node = new UndirectedGraphNode(node.label);
        visited.put(node, new_node);
        for (UndirectedGraphNode nbr : node.neighbors)
            new_node.neighbors.add(clone(nbr, visited));
        return new_node;
    }
}
```

BFS

```java
// Clone Graph
// BFS，时间复杂度O(n)，空间复杂度O(n)
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null) return null;
        // key is original node，value is copied node
        HashMap<UndirectedGraphNode,UndirectedGraphNode> visited = new HashMap<>();
        // each node in queue is already copied itself
        // but neighbors are not copied yet
        Queue<UndirectedGraphNode> q = new LinkedList<>();
        q.offer(node);
        visited.put(node, new UndirectedGraphNode(node.label));
        while (!q.isEmpty()) {
            UndirectedGraphNode cur = q.poll();
            for (UndirectedGraphNode nbr : cur.neighbors) {
                // a copy already exists
                if (visited.containsKey(nbr)) {
                    visited.get(cur).neighbors.add(visited.get(nbr));
                } else {
                    UndirectedGraphNode new_node =
                            new UndirectedGraphNode(nbr.label);
                    visited.put(nbr, new_node);
                    visited.get(cur).neighbors.add(new_node);
                    q.offer(nbr);
                }
            }
        }
        return visited.get(node);
    }
}
```

本章主要讲位操作相关的题目。

Reverse Bits

描述

Reverse bits of a given 32 bits unsigned integer.

For example, given input `43261596` (represented in binary as `00000010100101000001111010011100`), return `964176192` (represented in binary as `00111001011110000010100101000000`).

Follow up: If this function is called many times, how would you optimize it?

分析

最简单直接的做法，从右向左把一位位取出来，添加到新生成的整数的最低位即可。

第二个简单的方法，左右不断交换位，直到相遇。

解法1

```
// Reverse Bits
// Time Complexity: O(logn), Space Complexity: O(1)
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int result = 0;
        for (int i = 0; i < 32; ++i) {
            if ((n & 1) == 1) {
                result = (result << 1) + 1;
            } else {
                result = result << 1;
            }
            n = n >> 1;
        }
        return result;
    }
}
```

解法2

```java
// Reverse Bits
// Time Complexity: O(logn), Space Complexity: O(1)
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int left = 0;
        int right = 31;
        while (left < right) {
            // swap bit
            int x = (n >> left) & 1;
            int y = (n >> right) & 1;

            if (x != y) {
                n ^= (1 << left) | (1 << right);
            }
            ++left;
            --right;
        }
        return n;
    }
}
```

Repeated DNA Sequences

描述

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

Given s = `"AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"`,

Return:

`["AAAAACCCCC", "CCCCCAAAAA"]`.

分析

首先能想到一个简单直接的方法，用一个长度为10的窗口，从左到右扫描，放入 HashMap，并把计数器增一。最后，把 HashMap 中所有计数器大于1的字符串输出来。时间复杂度 `O(n)`, 由于HashMap中存储了所有长度为10的子串，所以空间复杂度 `O(10n)`。

由于字符串中只存在 A, C, G, T 四种字符，我们可以把每个字符映射为2个bit:

```
A -> 00
C -> 01
G -> 10
T -> 11
```

每个长度为10的字符串，可以映射为 20 bits, 小于32位，因此可以把这个字符串映射到一个整数。这个方法时间复杂度依旧是 `O(n)`，但空间复杂度下降到了 `O(n)`。

解法1 简单粗暴

```
// Repeated DNA Sequences
// Time Complexity: O(n), Space Complexity: O(10n)
public class Solution {
    public List<String> findRepeatedDnaSequences(String s) {
        final List<String> result = new ArrayList<>();
        if (s.length() < 10) return result;

        final Map<String, Integer> counter = new HashMap<>();

        for (int i = 0; i < s.length() - 9; ++i) {
            final String key = s.substring(i, i + 10);
            int value = counter.getOrDefault(key, 0);
            counter.put(key, value + 1);
        }

        for (Map.Entry<String, Integer> entry : counter.entrySet()) {
            if (entry.getValue() > 1) {
                result.add(entry.getKey());
            }
        }
        return result;
    }
}
```

解法2 完美哈希

```
// Repeated DNA Sequences
// Time Complexity: O(n), Space Complexity: O(n)
public class Solution {
    public List<String> findRepeatedDnaSequences(String s) {
        final List<String> result = new ArrayList<>();
        if (s.length() < LEN) return result;

        final Map<Character, Integer> charMap = new HashMap<>();
        charMap.put('A', 0);
        charMap.put('C', 1);
        charMap.put('G', 2);
        charMap.put('T', 3);

        final Map<Integer, Character> intMap = new HashMap<>();
        intMap.put(0, 'A');
        intMap.put(1, 'C');
        intMap.put(2, 'G');
        intMap.put(3, 'T');

        final Map<Integer, Integer> counter = new HashMap<>();

        for (int i = 0; i < s.length() - LEN + 1; ++i) {
            final String key = s.substring(i, i + 10);
            final int hashValue = strToInt(key, charMap);
            counter.put(hashValue, counter.getOrDefault(hashValue, 0) + 1);
```

```
        }

        for (Map.Entry<Integer, Integer> entry : counter.entrySet()) {
            if (entry.getValue() > 1) {
                result.add(intToStr(entry.getKey(), intMap));
            }
        }
        return result;
    }

    // perfect hash, no collisions
    private static int strToInt(String s, Map<Character, Integer> charMap) {
        assert s.length() == LEN;
        int x = 0;
        for (int i = 0; i < LEN; ++i) {
            final char ch = s.charAt(i);
            x = (x << 2) + charMap.get(ch);
        }
        return x;
    }
    private String intToStr(int x, Map<Integer, Character> intMap) {
        final StringBuilder sb = new StringBuilder();

        while (x > 0) {
            final char ch = intMap.get(x & 3);
            sb.append(ch);
            x >>= 2;
        }
        while (sb.length() < LEN) sb.append(intMap.get(0));
        return sb.reverse().toString();
    }
    private static final int LEN = 10;
}
```

Number of 1 Bits

描述

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer `11` has binary representation `00000000000000000000000000001011`, so the function should return 3.

分析

最直接的方法，做32次右移，统计出1的个数。

第二个方法，来自 "Hacker's Delight" 这本书。

使用Java解题时，需要注意：

1. 输入值n可能为负数，但应视其为无符号整数，因为Java中实际上是没有无符号整数的
2. 使用无符号右移操作符 `>>>`，可以忽略符号位

解法1

```
// Number of 1 Bits
// Time Complexity: O(32), Space Complexity: O(1)
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        for (int i = 0; i < 32; ++i) {
            count += n & 1;
            n >>>= 1;
        }
        return count;
    }
}
```

解法2

```java
// Number of 1 Bits
// Time Complexity: O(number of 1), Space Complexity: O(1)
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            ++count;
            n &= n - 1;
        }
        return count;
    }
}
```

Gray Code

描述

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.

分析

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

自然二进制码转换为格雷码：$g_0 = b_0, g_i = b_i \oplus b_{i-1}$

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

格雷码转换为自然二进制码：$b_0 = g_0, b_i = g_i \oplus b_{i-1}$

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

格雷码有数学公式，整数 `n` 的格雷码是$n \oplus (n/2)$。

这题要求生成 `n` 比特的所有格雷码。

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

方法2， `n` 比特的格雷码，可以递归地从 `n-1` 比特的格雷码生成。如下图所示。

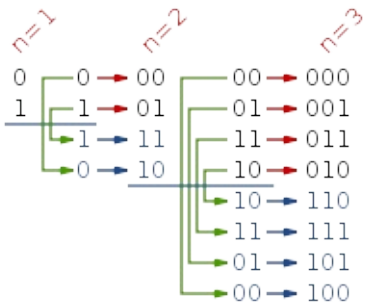


Figure: The first few steps of the reflect-and-prefix method.

数学公式

```
// Gray Code
// 数学公式，时间复杂度O(2^n)，空间复杂度O(1)
public class Solution {
    public ArrayList<Integer> grayCode(int n) {
        final int size = 1 << n;  // 2^n
        ArrayList<Integer> result = new ArrayList<>(size);

        for (int i = 0; i < size; ++i)
            result.add(binary_to_gray(i));
        return result;
    }
    private static int binary_to_gray(int n) {
        return n ^ (n >> 1);
    }
}
```

Reflect-and-prefix method

```
// Gray Code
// reflect-and-prefix method
// 时间复杂度O(2^n)，空间复杂度O(1)
public class Solution {
    public ArrayList<Integer> grayCode(int n) {
        final int size = 1 << n;
        ArrayList<Integer> result = new ArrayList<>(size);

        result.add(0);
        for (int i = 0; i < n; i++) {
            final int highest_bit = 1 << i;
            for (int j = result.size() - 1; j >= 0; j--) // 要反着遍历，才能对称
                result.add(highest_bit | result.get(j));
        }
        return result;
    }
}
```

Single Number

描述

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?

分析

异或，不仅能处理两次的情况，只要出现偶数次，都可以清零。

代码

```
// Single Number
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int singleNumber(int[] nums) {
        int x = 0;
        for (int i : nums) {
            x ^= i;
        }
        return x;
    }
};
```

相关题目

- Single Number II

Single Number II

描述

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?

分析

本题和上一题 Single Number，考察的是位运算。

方法1：创建一个长度为 `sizeof(int)` 的数组 `count[sizeof(int)]` ， `count[i]` 表示在在 `i` 位出现的1的次数。如果 `count[i]` 是3的整数倍，则忽略；否则就把该位取出来组成答案。

方法2：用 `one` 记录到当前处理的元素为止，二进制1出现“1次”（mod 3 之后的 1）的有哪些二进制位；用 `two` 记录到当前计算的变量为止，二进制1出现“2次”（mod 3 之后的 2）的有哪些二进制位。
当 `one` 和 `two` 中的某一位同时为1时表示该二进制位上1出现了3次，此时需要清零。即用二进制模拟三进制运算。最终 `one` 记录的是最终结果。

代码1

```
// Single Number II
// 方法1，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int singleNumber(int[] nums) {
        final int W = Integer.SIZE; // 一个整数的bit数，即整数字长
        int[] count = new int[W];  // count[i]表示在在i位出现的1的次数
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < W; j++) {
                count[j] += (nums[i] >> j) & 1;
                count[j] %= 3;
            }
        }
        int result = 0;
        for (int i = 0; i < W; i++) {
            result += (count[i] << i);
        }
        return result;
    }
};
```

代码2

```
// Single Number II
// 方法2，时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public int singleNumber(int[] nums) {
        int one = 0, two = 0, three = 0;
        for (int i : nums) {
            two |= (one & i);
            one ^= i;
            three = ~(one & two);
            one &= three;
            two &= three;
        }

        return one;
    }
};
```

相关题目

- Single Number

Single Number III

描述

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

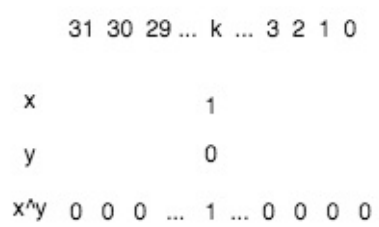
1. The order of the result is not important. So in the above example, [5, 3] is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

分析

本题是有两个未知数，是 "Single Number" 这道题的扩展，直接做异或肯定是不行的。有没有办法把两个未知数分开，使得可以应用Single Number 的解法呢？

设 `x`, `y` 是那两个未知数，那么如果对这个数组做异或的话，结果实质上等于 `x ^ y`，因为其他数都出现了两次，被抵消了。

但是仅仅是通过最后异或出来的值，是没办法区分出 `x` 和 `y` 的，但是足以帮助我们把 `x` 和 `y` 划分到不停地子数组中去。对于 `x` 和 `y`，由于二者不相等，那么二者至少存在一位不相同，如下图所示：



当找到这个 `k` 以后，就可以按照第 `k` 位是否等于1，将 `nums` 数组划分为两个子数组，然后把Single Number的算法直接应用到两个子数组上，就可以求出 `x` 和 `y` 了。

代码

```java
// Single Number III
// Time Complexity: O(log n), Space Complexity: O(1)
public class Solution {
    public int[] singleNumber(int[] nums) {
        int xorResult = 0;
        for (int x : nums) {
            xorResult ^= x;
        }

        // get the index of first 1
        int k = 0;
        for (k = 0; k < Integer.SIZE; ++ k) {
            if (((xorResult >>> k) & 1) == 1) {
                break;
            }
        }

        // use k to split the array into two subarrays
        // XOR result of the first subarray
        int xorResult2 = 0;
        for (int x : nums) {
            if (((x >>> k) & 1) == 1) {
                xorResult2 ^= x;
            }
        }
        return new int[] {xorResult2, xorResult ^ xorResult2};
    }
}
```

Power of Two

描述

Given an integer, write a function to determine if it is a power of two.

分析

如果是2的幂，则二进制的所有位中，有且仅有一个1。

可以复用 "Number of 1 Bits" 中的函数，计算出1的个数，如果为1，则返回true, 不为1，返回 false。

还有更巧妙的办法。如果一个数是2的幂，则它的二进制最高位必然为1，其余为0，此时如果我们减1的话，最高位降为0，其余位变为1，如果把两个数按位与，结果必然为0。

代码

```
// Power of Two
// Time Complexity: O(1), Space Complexity: O(1)
public class Solution {
    public boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n-1)) == 0;
    }
}
```

Missing Number

描述

Given an array containing `n` distinct numbers taken from `0, 1, 2, ..., n`, find the one that is missing from the array.

For example,

Given nums = `[0, 1, 3]` return `2`.

Note:

Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

分析

本题的意思是，从1到n的整数，其中某个数丢失了，替代它的是0。要我们找出这个丢失的数。

方法1，我们可以用公式计算出从1到n的和，减去实际数组的总和，差值就是那个丢失的数。

方法2，利用异或位运算，把数组中的每一个数，与1到n进行按位异或，最后剩下的，就是丢失的数。

方法3，二分查找。首先把数组排序，设中间元素为 `nums[mid]`，如果 `nums[mid]` 的值大于其下标，说明丢失的数字在左边，反之则在右边。时间复杂度 `O(nlogn)`，比前面两个方法慢，但是如果题目给的数组是事先排好序的，那么复杂度就是 `O(log n)`，所以这个方法还是很有意义的。

解法1

```
// Missing Number
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int missingNumber(int[] nums) {
        int sum = 0;
        for (int x : nums) {
            sum += x;
        }
        final int n = nums.length;
        final int sumExpected = (n * (n + 1)) / 2;
        return sumExpected - sum;
    }
}
```

解法2

```java
// Missing Number
// Time Complexity: O(n), Space Complexity: O(1)
public class Solution {
    public int missingNumber(int[] nums) {
        int result = 0;
        for (int i = 0; i < nums.length; ++i) {
            result ^= (i+1) ^ nums[i];
        }
        return result;
    }
}
```

解法3

```java
// Missing Number
// Time Complexity: O(nlogn), Space Complexity: O(1)
public class Solution {
    public int missingNumber(int[] nums) {
        Arrays.sort(nums);
        int begin = 0;
        int end = nums.length;
        while (begin != end) {
            final int mid = begin + (end - begin) / 2;
            if (mid < nums[mid]) end = mid;
            else begin = mid + 1;
        }
        return end;
    }
}
```

Maximum Product of Word Lengths

描述

Given a string array `words`, find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given `["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]`

Return `16`

The two words can be `"abcw"`, `"xtfn"`.

Example 2:

Given `["a", "ab", "abc", "d", "cd", "bcd", "abcd"]`

Return 4

The two words can be `"ab"`, `"cd"`.

Example 3:

Given `["a", "aa", "aaa", "aaaa"]`

Return 0

No such pair of words.

分析

由于只有26个小写字母，所以，我们可以为数组中的每个`word`开辟一个长为26的布尔数组作为哈希表，然后用一个两重for循环，两两比较，如果不存在公共的字母，则计算二者的长度的乘积，取最大作为最终结果。时间复杂度`O(26n^2)`，空间复杂度`O(26n)`。

上面的方法可以进一步优化，即长度为26的布尔数组，小于32位，可以编码为一个整数，这样两个整数按位与，如果结果为1，说明存在公共字母，如果结果为0，说明不存在公共字母。时间复杂度`O(n^2)`，空间复杂度`O(n)`。

解法1

```java
// Maximum Product of Word Lengths
// Time Complexity: O(26n^2), Space Complexity: O(26n)
public class Solution {
    public int maxProduct(String[] words) {
        final int n = words.length;
        final boolean[][] hashset = new boolean[n][ALPHABET_SIZE];

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < words[i].length(); ++j) {
                hashset[i][words[i].charAt(j) - 'a'] = true;
            }
        }

        int result = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                boolean hasCommon = false;
                for (int k = 0; k < ALPHABET_SIZE; ++k) {
                    if (hashset[i][k] && hashset[j][k]) {
                        hasCommon = true;
                        break;
                    }
                }
                int tmp = words[i].length() * words[j].length();
                if (!hasCommon && tmp > result) {
                    result = tmp;
                }
            }
        }
        return result;
    }
    private static final int ALPHABET_SIZE = 26;
}
```

解法2

```java
// Maximum Product of Word Lengths
// Time Complexity: O(n^2), Space Complexity: O(n)
public class Solution {
    public int maxProduct(String[] words) {
        final int n = words.length;
        final int[] hashset = new int[n];

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < words[i].length(); ++j) {
                hashset[i] |= 1 << (words[i].charAt(j) - 'a');
            }
        }

        int result = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                int tmp = words[i].length() * words[j].length();
                if ((hashset[i] & hashset[j]) == 0 && tmp > result) {
                    result = tmp;
                }
            }
        }
        return result;
    }
}
```

Bitwise AND of Numbers Range

描述

Given a range `[m, n]` where `0 <= m <= n <= 2147483647`, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range `[5, 7]`, you should return `4`.

分析

最简单的解法，遍历所有数，不断进行按位与，时间复杂度`O(n)`。这个做法太慢，不可接受。

先观察`[5,7]`这个例子，`5,6,7`的二进制如下：

```
101
110
111
```

三个数按位与后结果为0，仔细观察这个过程我们可以得出，最后的结果是该范围内所有数的左边的共同部分，即公共左边首部(left header)。

我们再举一个更大的例子来感受一下，例如`[26,30]`，这些数的二进制如下：

```
11010
11011
11100
11101
11110
```

最后的结果是`11000`，果然是公共左边首部。

发现了规律后，这题就简单了，我们只需要写代码找到公共左边首部即可。

解法1

```java
// Bitwise AND of Numbers Range
// Time Complexity: O(log n), Space Complexity: O(1)
public class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        int mask = Integer.MAX_VALUE;

        while ((m & mask) != (n & mask)) {
            mask = mask << 1;
        }
        return m & mask;
    }
}
```

解法2

```java
// Bitwise AND of Numbers Range
// Time Complexity: O(log n), Space Complexity: O(1)
public class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        while (n > m) {
            n &= n - 1;
        }
        return m & n;
    }
}
```

Power of Three

描述

Given an integer, write a function to determine if it is a power of three.

Follow up:

Could you do it without using any loop / recursion?

分析

最简单的方法，不断除以3，看最后能否得到1。

如果不用循环和递归，那么就需要找数学方法了。最简单的，我们可以把该整数对3取对数，如果结果是整数，说明该整数是3的幂。

代码

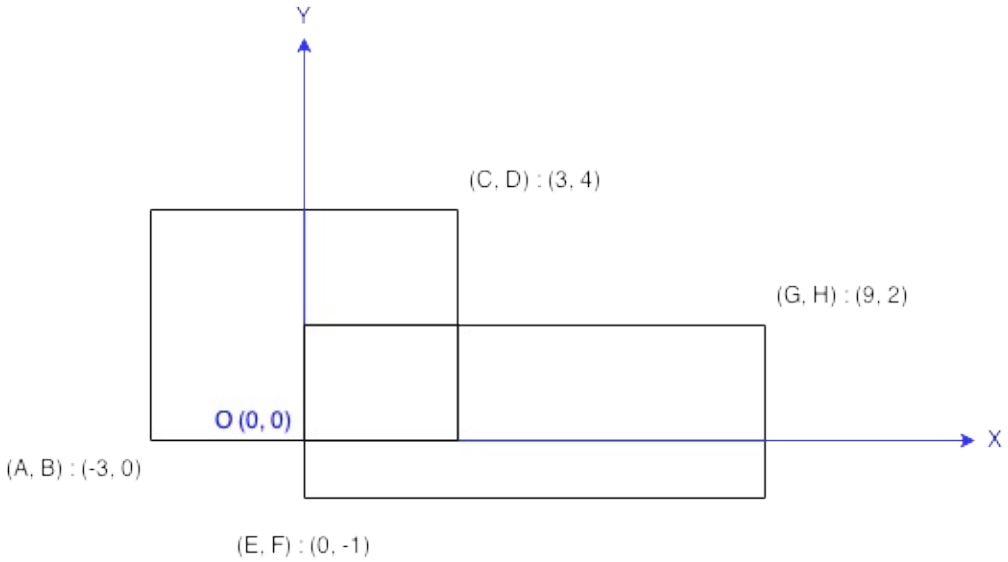
```
// Power of Three
// Time Complexity: O(1), Space Complexity: O(1)
public class Solution {
    public boolean isPowerOfThree(int n) {
        return (Math.log10(n) / Math.log10(3)) % 1 == 0;
    }
}
```

Rectangle Area

描述

Find the total area covered by two rectilinear rectangles in a 2D plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Assume that the total area is never beyond the maximum possible value of int.

分析

简单平面几何。根据容斥原理：S(M ∪ N) = S(M) + S(N) - S(M ∩ N)，最关键的是求出相交部分的面积。

代码

```
// Rectangle Area
// Time Complexity: O(1), Space Complexity: O(1)
public class Solution {
    public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
        final int area = (C - A) * (D- B) + (G - E) * (H - F);
        // prevent overflow
        if (C < E || G < A || D < F || H < B) return area;
        final int intersection = Math.max(Math.min(C, G) - Math.max(A, E), 0) *
                Math.max(Math.min(D, H) - Math.max(B, F), 0);
        return area - intersection;
    }
}
```

本节主要讲数论相关的题目。一般数论题不太适合用于面试中，所以面试中很少出现数论题，不过掌握一些常见的数论知识还是有好处的。

Happy Number

描述

Write a function to determine if a number is "happy number".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example: 19 is a happy number

$1^2 + 9^2 = 82$

$8^2 + 2^2 = 68$

$6^2 + 8^2 = 100$

$1^2 + 0^2 + 0^2 = 1$

分析

这题找到规律后就简单了。如果右边的出现了某个重复的数，但不是1，说明会无限循环下去，这个数就不是快乐数，如果是1，则是快乐数。

代码

```java
// Happy Number
// Time complexity: ?, Space complexity: ?
public class Solution {
    public boolean isHappy(int n) {
        final Set<Integer> existed = new HashSet<>();
        while (true) {
            int sum = 0;
            while (n > 0) {
                int digit = n % 10;
                sum += digit * digit;
                n /= 10;
            }
            if (existed.contains(sum)) {
                return sum == 1;
            } else {
                existed.add(sum);
                n = sum;
            }
        }
    }
}
```

Ugly Number

描述

Write a function to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

分析

思路很简单，把 `n` 里面的2,3,5 全部消掉，看最后能不能剩下1。

代码

```
// Ugly Number
// Time complexity: O(logn), Space complexity: O(1)
public class Solution {
    public boolean isUgly(int num) {
        if (num == 0) return false;
        while (num % 2 == 0) num /= 2;
        while (num % 3 == 0) num /= 3;
        while (num % 5 == 0) num /= 5;
        return num == 1;
    }
}
```

相关题目

- Ugly Number II

Ugly Number II

描述

Write a function to find the `n`-th ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

Note that 1 is typically treated as an ugly number.

Hint:

1. The naive approach is to call `isUgly()` for every number until you reach the `n`-th one. Most numbers are not ugly. Try to focus your effort on generating only the ugly ones.
2. An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.
3. The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L_1, L_2, and L_3.
4. Assume you have U_k, the k^{th} ugly number. Then U_{k+1} must be $Min(L_1 * 2, L_2 * 3, L_3 * 5)$.

分析

根据提示中的信息，我们知道丑陋序列可以拆分成3个子序列：

1. 1x2, 2x2, 3x2, 4x2, 5x2, ...
2. 1x3, 2x3, 3x3, 4x3, 5x3, ...
3. 1x5, 2x5, 3x5, 4x5, 5x5, ...

每次从三个列表中取出当前最小的那个加入序列，直到第 `n` 个为止。

代码

```java
// Ugly Number II
// Time complexity: O(n), Space complexity: O(n)
// TODO: 没必要保存所有的ugly number, 空间可以优化到O(1)
public class Solution {
    public int nthUglyNumber(int n) {
        final int[] nums = new int[n];
        nums[0] = 1; // 1 is the first ugly number
        int index = 0, index2 = 0, index3 = 0, index5 = 0;

        while (index + 1 < n) {
            int x2 = nums[index2] * 2;
            int x3 = nums[index3] * 3;
            int x5 = nums[index5] * 5;
            int min = Math.min(x2, Math.min(x3, x5));

            if (min == x2) ++index2;
            else if (min == x3) ++index3;
            else ++index5;

            if (min != nums[index]) { // skip duplicate
                nums[++index] = min;
            }
        }
        return nums[n - 1];
    }
}
```

相关题目

- Ugly Number
- Super Ugly Number

Super Ugly Number

描述

Write a function to find the `n`-th super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list `primes` of size `k`. For example, `[1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32]` is the sequence of the first 12 super ugly numbers given `primes = [2, 7, 13, 19]` of size 4.

Note:

1. 1 is a super ugly number for any given `primes`.
2. The given numbers in `primes` are in ascending order.
3. 0 < `k` ≤ 100, 0 < `n` ≤ 1000000, 0 < `primes[i]` < 1000.

分析

这题是 Ugly Number II 的扩展。在"Ugly Number II"中，`primes=[2,3,5]`，这题中`primes`可以自由变化。

所以这题可以用"Ugly Number II"的思路解决。每次要从多个列表中选择最小的元素，我们可以维护一个大小为`primes`长度的小根堆。

代码

```java
// Super Ugly Number
// Time complexity: O(n), Space complexity: O(n)
public class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        final int[] nums = new int[n];
        nums[0] = 1; // 1 is the first ugly number
        final Queue<Node> q = new PriorityQueue<>();
        for (int i = 0; i < primes.length; ++i) {
            q.add(new Node(0, primes[i], primes[i]));
        }

        for (int i = 1; i < n; ++i) {
            // get the min element and add to nums
            Node node = q.peek();
            nums[i] = node.val;

            // update top elements
            do {
                node = q.poll();
                node.val = nums[++node.index] * node.prime;
                q.add(node); // push it back
                // prevent duplicate
            } while (!q.isEmpty() && q.peek().val == nums[i]);
        }
        return nums[n - 1];
    }
    static class Node implements Comparable<Node> {
        private int index;
        private int val;
        private int prime;

        public Node(int index, int val, int prime) {
            this.index = index;
            this.val = val;
            this.prime = prime;
        }
        public int compareTo(Node other) {
            return this.val - other.val;
        }
    }
}
```

相关题目

- Ugly Number II

Fraction to Recurring Decimal

描述

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

- Given numerator = 1, denominator = 2, return "0.5".
- Given numerator = 2, denominator = 1, return "2".
- Given numerator = 2, denominator = 3, return "0.(6)".

分析

这题的难点是如何找到无限循环的那一段。仔细回想一下人脑进行除法的过程，会发现，当一个余数第二次重复出现时，就说明小数点后开始无限循环了。

代码

```java
// Fraction to Recurring Decimal
// Time Complexity: ?, Space Complexity: ?
public class Solution {
    public String fractionToDecimal(int numerator, int denominator) {
        if (numerator == 0) return "0";

        final StringBuilder result = new StringBuilder();
        // determine the sign
        if ((numerator < 0) ^ (denominator < 0)) result.append('-');

        // Integer.MIN_VALUE will overflow, so use long
        // Math.abs(MIN_VALUE) will overflow
        long n = numerator;
        n = Math.abs(n);
        long d = denominator;
        d = Math.abs(d);

        // append integral part
        result.append(String.valueOf(n / d));
        if (n % d == 0) return result.toString();
        result.append('.');

        final Map<Long, Integer> map = new HashMap<>();
        // simulate the division process
        for (long r = n % d; r != 0; r %= d) {
            // find a existed remainder, so we reach
            // the end of the repeating part
            if (map.containsKey(r)) {
                result.insert(map.get(r), "(");
                result.append(')');
                break;
            }

            map.put(r, result.length());

            r *= 10;
            result.append(Character.forDigit((int)(r/d), 10));
        }

        return result.toString();
    }
}
```

Factorial Trailing Zeroes

描述

Given an integer `n`, return the number of trailing zeroes in `n!`.

Note: Your solution should be in logarithmic time complexity.

分析

对任意一个正整数n进行质因数分解，$n = 2^x * 3^y * 5^z ...$，末尾0的个数M与2和5的个数即X、Z有关，每一对2和5都可以得到10，故 `M=min(X,Z)`。又因为能被2整除的数出现的频率要比能被5整除的数出现的频率高，所以 `M=Z`。所以只要计算出Z，就可以得到 `n!` 的末尾0的个数。

解法1

```
// Factorial Trailing Zeroes
// TLE
// Time Complexity: O(nlogn), Space Complexity: O(1)
public class Solution {
    public int trailingZeroes(int n) {
        int result = 0;

        for (int i = 1; i <= n; ++i) {
            int j = i;
            while (j % 5 == 0) {
                ++result;
                j /= 5;
            }
        }
        return result;
    }
}
```

解法2

上面的解法会超时，可以优化一下。

可以用公式计算出末尾0的个数，$Z = N/5 + N/5^2 + N/5^3 + ...$，$N/5$ 表示从1到N中能被5整除的数的个数，由于每个数都能贡献一个5，意味着能贡献一个0。$N/5^2$ 表示从1到N中能被 5^2 整除的数的个数，每个数都能贡献2个5，意味着能贡献两个0，不过由于其中一次已经包含在 $N/5$ 中了，只能再贡献一个0，依次类推。

```java
// Factorial Trailing Zeroes
// Time Complexity: O(logn), Space Complexity: O(1)
public class Solution {
    public int trailingZeroes(int n) {
        int result = 0;

        while (n > 0) {
            result += n / 5;
            n /= 5;
        }
        return result;
    }
}
```

Nim Game

描述

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

分析

这题是尼姆游戏的简化版。

尼姆游戏最流行的版本是用12枚硬币。

游戏规则很简单，游戏双方轮流取 1 枚或多枚硬币（只能在同一行），谁拿到最后一枚就算赢。

有趣的是，有人发现，当扩展到任意多行，每行有任意枚硬币时，利用二进制，可以把这个游戏玩得风生水起。哈佛大学的数学教授布顿在 1901 年首次发表了论文详述了这个问题，也正是他，正式将这个游戏命名为尼姆游戏。

把玩家每一步操作之后的游戏局面叫做“棋局”。在布顿的论文中，如果玩家每一步操作后的棋局能保证自己获胜，那就是“安全的”，否则就是“不安全的”。每个不安全棋局都可以一步正确的操作变成安全的，而如果没有正确地操作，一个安全的棋局就会变成不安全的。

回到我们上面说的那个流行版本上，可以看到在初始状态，它的二进制表示如下图：

3		1	1
4	1	0	0
5	1	0	1
SUM	2	1	2

可以看到，第 2 列之和为奇数，所以这个本版的初始状态是不安全的。拿掉最上面一行的 2 枚硬币，第 1 行就变成了 1 ，从而留下了一个安全棋局。通过用其他方法试验，可以看到，拿掉第 1 行的 2 枚硬币是留下安全棋局的唯一操作。

尼姆游戏深受数学家喜爱并被广泛研究，它因此产生了很多变体。1910 年美国数学家穆尔就提出了一个，它规则与尼姆游戏相同，只不过玩家可以从不超过指定数 k 的任意多行里拿掉硬币。有趣的是，它同样可以通过二进制来分析，只要把安全棋局定义为：二进制表里的每列之和都可以被 k + 1 整除就可以了。

这题是行数为1，k=3的简化版尼姆游戏。由于是先手，只需要判断当前的石头数能否被4整除，如果能整除，则一定会输，否则一定能赢。

参考资料: http://www.guokr.com/article/68595/

代码

```
// Nim Game
// Time Complexity: O(1), Space Complexity: O(1)
public class Solution {
    public boolean canWinNim(int n) {
        return n % 4 != 0;
    }
}
```

模拟

这类题目思路比较简单直白，按照问题的描述，把解题的步骤一步一步直白的翻译成代码，就行了。这类题目主要考察写代码是否熟练，是否具备基本的调试能力，编程风格是否良好等。

Reverse Integer

描述

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

Have you thought about this?

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

Throw an exception? Good, but what if throwing an exception is not an option? You would then have to re-design the function (ie, add an extra parameter).

分析

短小精悍的题，代码也可以写的很短小。

代码

```java
// Reverse Integer
// 时间复杂度O(logn)，空间复杂度O(1)
// 考虑 1.负数的情况 2. 溢出的情况(正溢出&&负溢出，比如 x = -2147483648(即-2^31) )
public class Solution {
    public int reverse(int x) {
        long r = 0;
        long t = x;
        t = t > 0 ? t : -t;
        for (; t > 0; t /= 10)
            r = r * 10 + t % 10;

        boolean sign = x > 0 ? false: true;
        if (r > 2147483647 || (sign && r > Integer.MAX_VALUE)) {
            return 0;
        } else {
            if (sign) {
                return (int)-r;
            } else {
                return (int)r;
            }
        }
    }
}
```

相关题目

- Palindrome Number

Palindrome Number

描述

Determine whether an integer is a palindrome. Do this without extra space.

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

分析

首先想到，可以利用上一题，将整数反转，然后与原来的整数比较，是否相等，相等则为 Palindrome 的。可是 reverse()会溢出。

正确的解法是，不断地取第一位和最后一位（10进制下）进行比较，相等则取第二位和倒数第二位，直到完成比较或者中途找到了不一致的位。

代码

```
// Palindrome Number
// 时间复杂度O(1)，空间复杂度O(1)
public class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0) return false;
        int d = 1; // divisor
        while (x / d >= 10) d *= 10;

        while (x > 0) {
            int q = x / d;  // quotient
            int r = x % 10;   // remainder
            if (q != r) return false;
            x = x % d / 10;
            d /= 100;
        }
        return true;
    }
}
```

相关题目

- Reverse Integer

- Valid Palindrome

Insert Interval

描述

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1: Given intervals `[1,3],[6,9]`, insert and merge `[2,5]` in as `[1,5],[6,9]`.

Example 2: Given `[1,2],[3,5],[6,7],[8,10],[12,16]`, insert and merge `[4,9]` in as `[1,2],[3,10],[12,16]`.

This is because the new interval `[4,9]` overlaps with `[3,5],[6,7],[8,10]`.

分析

无

代码

```
struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) { }
    Interval(int s, int e) : start(s), end(e) { }
};

// Insert Interval
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
        for (int i = 0; i < intervals.size();) {
            final Interval cur = intervals.get(i);
            if (newInterval.end < cur.start) {
                intervals.add(i, newInterval);
                return intervals;
            } else if (newInterval.start > cur.end) {
                ++i;
                continue;
            } else {
                newInterval.start = Math.min(newInterval.start, cur.start);
                newInterval.end = Math.max(newInterval.end, cur.end);
                intervals.remove(i);
            }
        }
        intervals.add(newInterval);
        return intervals;
    }
}
```

相关题目

- Merge Intervals

Merge Intervals

描述

Given a collection of intervals, merge all overlapping intervals.

For example, Given `[1,3],[2,6],[8,10],[15,18]`, return `[1,6],[8,10],[15,18]`

分析

复用一下Insert Intervals的解法即可，创建一个新的interval集合，然后每次从旧的里面取一个interval出来，然后插入到新的集合中。

代码

```
// Merge Interval
//复用一下Insert Intervals的解法即可
// 时间复杂度O(n1+n2+...)，空间复杂度O(1)
public class Solution {
    public List<Interval> merge(List<Interval> intervals) {
        List<Interval> result = new ArrayList<>();
        for (int i = 0; i < intervals.size(); i++) {
            insert(result, intervals.get(i));
        }
        return result;
    }
    private static List<Interval> insert(List<Interval> intervals,
                                         Interval newInterval) {
        for (int i = 0; i < intervals.size();) {
            final Interval cur = intervals.get(i);
            if (newInterval.end < cur.start) {
                intervals.add(i, newInterval);
                return intervals;
            } else if (newInterval.start > cur.end) {
                ++i;
                continue;
            } else {
                newInterval.start = Math.min(newInterval.start, cur.start);
                newInterval.end = Math.max(newInterval.end, cur.end);
                intervals.remove(i);
            }
        }
        intervals.add(newInterval);
        return intervals;
    }
}
```

相关题目

- Insert Interval

Minimum Window Substring

描述

Given a string `S` and a string `T`, find the minimum window in `S` which will contain all the characters in `T` in complexity `O(n)`.

For example, `S = "ADOBECODEBANC", T = "ABC"`

Minimum window is `"BANC"`.

Note:

- If there is no such window in `S` that covers all characters in `T`, return the emtpy string `""`.
- If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in `S`.

分析

双指针，动态维护一个区间。尾指针不断往后扫，当扫到有一个窗口包含了所有`T`的字符后，然后再收缩头指针，直到不能再收缩为止。最后记录所有可能的情况中窗口最小的

代码

```
// LeetCode, Minimum Window Substring
// 时间复杂度O(n)，空间复杂度O(1)
class Solution {
public:
    string minWindow(string S, string T) {
        if (S.empty()) return "";
        if (S.size() < T.size()) return "";

        const int ASCII_MAX = 256;
        int appeared_count[ASCII_MAX];
        int expected_count[ASCII_MAX];
        fill(appeared_count, appeared_count + ASCII_MAX, 0);
        fill(expected_count, expected_count + ASCII_MAX, 0);

        for (size_t i = 0; i < T.size(); i++) expected_count[T[i]]++;

        int minWidth = INT_MAX, min_start = 0;  // 窗口大小，起点
        int wnd_start = 0;
        int appeared = 0;  // 完整包含了一个T
        //尾指针不断往后扫
        for (size_t wnd_end = 0; wnd_end < S.size(); wnd_end++) {
            if (expected_count[S[wnd_end]] > 0)  {  // this char is a part of T
                appeared_count[S[wnd_end]]++;
                if (appeared_count[S[wnd_end]] <= expected_count[S[wnd_end]])
                    appeared++;
            }
            if (appeared == T.size()) {  // 完整包含了一个T
                // 收缩头指针
                while (appeared_count[S[wnd_start]] > expected_count[S[wnd_start]]
                        || expected_count[S[wnd_start]] == 0) {
                    appeared_count[S[wnd_start]]--;
                    wnd_start++;
                }
                if (minWidth > (wnd_end - wnd_start + 1)) {
                    minWidth = wnd_end - wnd_start + 1;
                    min_start = wnd_start;
                }
            }
        }

        if (minWidth == INT_MAX) return "";
        else return S.substr(min_start, minWidth);
    }
};
```

Multiply Strings

描述

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

分析

高精度乘法。

常见的做法是将字符转化为一个int，一一对应，形成一个int数组。但是这样很浪费空间，一个int32的最大值是 `2^{31}-1=2147483647` ，可以与9个字符对应，由于有乘法，减半，则至少可以与4个字符一一对应。一个int64可以与9个字符对应。

代码1

```
// Multiply Strings
// 一个字符对应一个int
// 时间复杂度O(n*m)，空间复杂度O(n+m)
public class Solution {
    public String multiply(String num1, String num2) {
        BigInt bigInt1 = new BigInt(num1);
        BigInt bigInt2 = new BigInt(num2);
        BigInt result = BigInt.multiply(bigInt1, bigInt2);
        return result.toString();
    }

    // 一个字符对应一个int
    static class BigInt {
        private final int[] d;

        public BigInt(String s) {
            this.d = fromString(s);
        }
        public BigInt(int[] d) {
            this.d = d;
        }

        private static int[] fromString(String s) {
            int[] d = new int[s.length()];
            for (int i = s.length() - 1, j = 0; i >= 0; --i)
                d[j++] = Character.getNumericValue(s.charAt(i));
            return d;
        }

        @Override
        public String toString() {
```

```
            final StringBuilder sb = new StringBuilder();
            for (int i = d.length - 1; i >= 0; --i) {
                sb.append(Character.forDigit(d[i], 10));
            }
            return sb.toString();
        }

        public static BigInt multiply(BigInt x, BigInt y) {
            int[] z = new int[x.d.length + y.d.length];
            for (int i = 0; i < x.d.length; ++i) {
                for (int j = 0; j < y.d.length; ++j) {
                    z[i + j] += x.d[i] * y.d[j];
                    z[i + j + 1] += z[i + j] / 10;
                    z[i + j] %= 10;
                }
            }
            // find the first 0 from right to left
            int i = z.length - 1;
            for (; i > 0 && z[i] == 0; --i) /* empty */;

            if (i == z.length - 1) {
                return new BigInt(z);
            } else { // make a copy
                int[] tmp = new int[i + 1];
                System.arraycopy(z, 0, tmp, 0, i + 1);
                return new BigInt(tmp);
            }
        }
    }
}
```

代码2

```
// Multiply Strings
// 9个字符对应一个 long
// 时间复杂度O(n*m)，空间复杂度O(n+m)
public class Solution {
    public String multiply(String num1, String num2) {
        BigInt bigInt1 = BigInt.fromString(num1);
        BigInt bigInt2 = BigInt.fromString(num2);
        BigInt result = BigInt.multiply(bigInt1, bigInt2);
        return result.toString();
    }

    // 9个字符对应一个 long
    static class BigInt {
        /** 一个数组元素对应9个十进制位，即数组是亿进制的
         * 因为 1000000000 * 1000000000 没有超过 2^63-1
         */
        final static int BIGINT_RADIX = 1000000000;
        final static int RADIX_LEN = 9;
```

```
        /** 万进制整数. */
        private final long[] digits;

        public BigInt(long[] digits) {
            this.digits = digits;
        }

        private static BigInt fromString(String s) {
            long[] digits;
            if (s.length() % RADIX_LEN == 0) {
                digits = new long[s.length() / RADIX_LEN];
            } else {
                digits = new long[s.length() / RADIX_LEN + 1];
            }
            for (int i = s.length(), k = 0; i > 0; i -= RADIX_LEN) {
                long tmp = 0;
                for (int j = Math.max(0, i - RADIX_LEN); j < i; ++j) {
                    tmp = tmp * 10 + Character.getNumericValue(s.charAt(j));
                }
                digits[k++] = tmp;
            }
            return new BigInt(digits);
        }

        @Override
        public String toString() {
            final StringBuilder sb = new StringBuilder(
                    Long.toString(digits[digits.length-1]));

            for (int i = digits.length - 2; i >= 0; --i) {
                sb.append(String.format("%0" + RADIX_LEN + "d", digits[i]));
            }
            return sb.toString();
        }

        public static BigInt multiply(BigInt x, BigInt y) {
            long[] z = new long[x.digits.length + y.digits.length];
            for (int i = 0; i < x.digits.length; ++i) {
                for (int j = 0; j < y.digits.length; ++j) {
                    z[i + j] += x.digits[i] * y.digits[j];
                    z[i + j + 1] += z[i + j] / BIGINT_RADIX;
                    z[i + j] %= BIGINT_RADIX;
                }
            }
            // find the first 0 from right to left
            int i = z.length - 1;
            for (; i > 0 && z[i] == 0; --i) /* empty */;

            if (i == z.length - 1) {
                return new BigInt(z);
            } else { // make a copy
                long[] tmp = new long[i + 1];
                System.arraycopy(z, 0, tmp, 0, i + 1);
```

```
                return new BigInt(tmp);
            }
        }
    }
}
```

Substring with Concatenation of All Words

描述

You are given a string, `S`, and a list of words, `L`, that are all of the same length. Find all starting indices of substring(s) in `S` that is a concatenation of each word in `L` exactly once and without any intervening characters.

For example, given:

```
S: "barfoothefoobarman"
L: ["foo", "bar"]
```

You should return the indices: `[0,9]`.(order does not matter).

分析

无

代码

```java
// Substring with Concatenation of All Words
// 时间复杂度O(n*m)，空间复杂度O(m)
public class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        final int wordLength = words[0].length();
        final int catLength = wordLength * words.length;
        List<Integer> result = new ArrayList<>();

        if (s.length() < catLength) return result;

        HashMap<String, Integer> wordCount = new HashMap<>();

        for (String word : words)
            wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);

        for (int i = 0; i <= s.length() - catLength; ++i) {
            HashMap<String, Integer> unused = new HashMap<>(wordCount);

            for (int j = i; j < i + catLength; j += wordLength) {
                final String key = s.substring(j, j + wordLength);
                final int pos = unused.getOrDefault(key, -1);

                if (pos == -1 || pos == 0) break;

                unused.put(key, pos - 1);
                if (pos - 1 == 0) unused.remove(key);
            }

            if (unused.size() == 0) result.add(i);
        }

        return result;
    }
}
```

Pascal's Triangle

描述

Given `numRows`, generate the first `numRows` of Pascal's triangle.

For example, given `numRows = 5`,

Return

```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

分析

本题可以用队列，计算下一行时，给上一行左右各加一个0，然后下一行的每个元素，就等于左上角和右上角之和。

另一种思路，下一行第一个元素和最后一个元素赋值为1，中间的每个元素，等于上一行的左上角和右上角元素之和。

从左到右

```java
// Pascal's Triangle
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> result = new ArrayList<>();
        if(numRows == 0) return result;

        result.add(new ArrayList<>(Arrays.asList(1))); //first row

        for(int i = 2; i <= numRows; ++i) {
            Integer[] current = new Integer[i]; // 本行
            Arrays.fill(current, 1);
            List<Integer> prev = result.get(i - 2); // 上一行

            for(int j = 1; j < i - 1; ++j) {
                current[j] = prev.get(j-1) + prev.get(j); // 左上角和右上角之和
            }
            result.add(new ArrayList<>(Arrays.asList(current)));
        }
        return result;
    }
}
```

从右到左

```java
// Pascal's Triangle
// 时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> array = new ArrayList<>();
        for (int i = 1; i <= numRows; i++) {
            for (int j = i - 2; j > 0; j--) {
                array.set(j, array.get(j - 1) + array.get(j));
            }
            array.add(1);
            result.add(new ArrayList<Integer>(array));
        }
        return result;
    }
}
```

相关题目

- Pascal's Triangle II

Pascal's Triangle II

描述

Given an index `k`, return the `k`-th row of the Pascal's triangle.

For example, given `k = 3`,

Return `[1,3,3,1]`.

Note: Could you optimize your algorithm to use only `O(k)` extra space?

分析

滚动数组。

代码

```
// Pascal's Triangle II
// 滚动数组，时间复杂度O(n^2)，空间复杂度O(n)
public class Solution {
    public List<Integer> getRow(int rowIndex) {
        List<Integer> array = new ArrayList<>();
        for (int i = 0; i <= rowIndex; i++) {
            for (int j = i - 1; j > 0; j--){
                array.set(j, array.get(j - 1) + array.get(j));
            }
            array.add(1);
        }
        return array;
    }
}
```

```
// LeetCode, Pascal's Triangle II
// 滚动数组，时间复杂度O(n^2)，空间复杂度O(n)
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> array;
        for (int i = 0; i <= rowIndex; i++) {
            for (int j = i - 1; j > 0; j--){
                array[j] = array[j - 1] + array[j];
            }
            array.push_back(1);
        }
        return array;
    }
};
```

相关题目

- Pascal's Triangle

Spiral Matrix

描述

Given a matrix of `m × n` elements (`m` rows, `n` columns), return all elements of the matrix in spiral order.

For example, Given the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
```

You should return `[1,2,3,6,9,8,7,4,5]`.

分析

模拟。

解法1 迭代

```
// Spiral Matrix
// 时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix.length == 0) return result;
        int beginX = 0, endX = matrix[0].length - 1;
        int beginY = 0, endY = matrix.length - 1;
        while (true) {
            // From left to right
            for (int j = beginX; j <= endX; ++j) result.add(matrix[beginY][j]);
            if (++beginY > endY) break;
            // From top to bottom
            for (int i = beginY; i <= endY; ++i) result.add(matrix[i][endX]);
            if (beginX > --endX) break;
            // From right to left
            for (int j = endX; j >= beginX; --j) result.add(matrix[endY][j]);
            if (beginY > --endY) break;
            // From bottom to top
            for (int i = endY; i >= beginY; --i) result.add(matrix[i][beginX]);
            if (++beginX > endX) break;
        }
        return result;
    }
}
```

解法2 递归

```
// Spiral Matrix
// 时间复杂度O(n^2)，空间复杂度O(1)
public class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix.length == 0) return result;

        left = 0;
        right = matrix[0].length - 1;
        up = 0;
        down = matrix.length - 1;
        dfs(matrix, 0, 0, 0, result);
        return result;
    }

    private void dfs(int[][] matrix, int i, int j, int direction,
                     List<Integer> result) {
        if (i < up || i > down) return;
        if (j < left || j > right) return;
        result.add(matrix[i][j]);

        int nextDirection = (direction + 1) % 4;
        switch (direction) {
            case 0:  // right
                if (j < right) {
                    dfs(matrix, i, j + 1, direction, result);
                } else {
                    ++up;
                    dfs(matrix, i + 1, j, nextDirection, result);
                }
                break;
            case 1:  // down
                if (i < down) {
                    dfs(matrix, i+1, j, direction, result);
                } else {
                    --right;
                    dfs(matrix, i, j - 1, nextDirection, result);
                }
                break;
            case 2:  // left
                if (j > left) {
                    dfs(matrix, i, j - 1, direction, result);
                } else {
                    --down;
                    dfs(matrix, i - 1, j, nextDirection, result);
                }
                break;
            default: // up
                if (i > up) {
                    dfs(matrix, i - 1, j, direction, result);
```

```
                } else {
                    ++left;
                    dfs(matrix, i, j + 1, nextDirection, result);
                }
            }

        }

        private int left;
        private int right;
        private int up;
        private int down;
    }
```

相关题目

- Spiral Matrix II

Spiral Matrix II

描述

Given an integer `n`, generate a square matrix filled with elements from 1 to `n^2` in spiral order.

For example, Given `n = 3`,

You should return the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

分析

这题比上一题要简单。

代码1

```
// Spiral Matrix II
// 时间复杂度O(n^2)，空间复杂度O(n^2)
public class Solution {
    public int[][] generateMatrix(int n) {
        int[][] matrix = new int[n][n];
        int begin = 0, end = n - 1;
        int num = 1;

        while (begin < end) {
            for (int j = begin; j < end; ++j) matrix[begin][j] = num++;
            for (int i = begin; i < end; ++i) matrix[i][end] = num++;
            for (int j = end; j > begin; --j) matrix[end][j] = num++;
            for (int i = end; i > begin; --i) matrix[i][begin] = num++;
            ++begin;
            --end;
        }

        if (begin == end) matrix[begin][begin] = num;

        return matrix;
    }
}
```

代码2

```
// LeetCode, Spiral Matrix II
// @author 龚陆安 (http://weibo.com/luangong)
// 时间复杂度O(n^2)，空间复杂度O(n^2)
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector< vector<int> > matrix(n, vector<int>(n));
        if (n == 0) return matrix;
        int beginX = 0, endX = n - 1;
        int beginY = 0, endY = n - 1;
        int num = 1;
        while (true) {
            for (int j = beginX; j <= endX; ++j) matrix[beginY][j] = num++;
            if (++beginY > endY) break;

            for (int i = beginY; i <= endY; ++i) matrix[i][endX] = num++;
            if (beginX > --endX) break;

            for (int j = endX; j >= beginX; --j) matrix[endY][j] = num++;
            if (beginY > --endY) break;

            for (int i = endY; i >= beginY; --i) matrix[i][beginX] = num++;
            if (++beginX > endX) break;
        }
        return matrix;
    }
};
```

相关题目

- Spiral Matrix

ZigZag Conversion

描述

The string `"PAYPALISHIRING"` is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: `"PAHNAPLSIIGYIR"`

Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

`convert("PAYPALISHIRING", 3)` should return `"PAHNAPLSIIGYIR"`.

分析

要找到数学规律。真正面试中，不大可能出这种问题。

n=4:

```
P     I     N
A   L S   I G
Y A   H R
P     I
```

n=5:

```
P       H
A     S I
Y   I   R
P L     I  G
A       N
```

所以，对于每一层垂直元素的坐标 `(i,j)= (j+1 )*n +i`；对于每两层垂直元素之间的插入元素（斜对角元素），`(i,j)= (j+1)*n -i`

代码

```java
// ZigZag Conversion
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public String convert(String s, int numRows) {
        if (numRows <= 1 || s.length() <= 1) return s;
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < numRows; i++) {
            for (int j = 0, index = i; index < s.length();
                 j++, index = (2 * numRows - 2) * j + i) {
                result.append(s.charAt(index));  // 垂直元素
                if (i == 0 || i == numRows - 1) continue;   // 斜对角元素
                if (index + (numRows - i - 1) * 2 < s.length())
                    result.append(s.charAt(index + (numRows - i - 1) * 2));
            }
        }
        return result.toString();
    }
}
```

Divide Two Integers

描述

Divide two integers without using multiplication, division and mod operator.

分析

不能用乘、除和取模，那剩下的，还有加、减和位运算。

最简单的方法，是不断减去被除数。在这个基础上，可以做一点优化，每次把被除数翻倍，从而加速。

注意，写代码的时候，禁止使用 long.

代码

```java
// Divide Two Integers
// 时间复杂度O(logn)，空间复杂度O(1)
public class Solution {
    public int divide(int dividend, int divisor) {
        if(dividend == 0) return 0;
        if (divisor == 0) return Integer.MAX_VALUE;

        // 当 dividend = INT_MIN，divisor = -1时，结果会溢出
        if (dividend == Integer.MIN_VALUE) {
            if (divisor == -1) return Integer.MAX_VALUE;
            else if (divisor < 0)
                return 1 + divide(dividend - divisor, divisor);
            else
                return - 1 + divide((dividend + divisor), divisor);
        }
        if(divisor == Integer.MIN_VALUE){
            return dividend == divisor ? 1 : 0;
        }

        int a = dividend > 0 ? dividend : -dividend;
        int b = divisor > 0 ? divisor : -divisor;

        int result = 0;
        while (a >= b) {
            int c = b;
            for (int i = 0; a >= c;) {
                a -= c;
                result += 1 << i;
                if (c < Integer.MAX_VALUE / 2) { // prevent overflow
                    ++i;
                    c <<= 1;
                }
            }
        }

        return ((dividend^divisor) >> 31) != 0 ? (-result) : (result);
    }
}
```

Text Justification

描述

Given an array of words and a length `L`, format the text such that each line has exactly `L` characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly `L` characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

words: `["This", "is", "an", "example", "of", "text", "justification."]`

L: 16.

Return the formatted lines as:

```
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed `L` in length.

Corner Cases:

- A line other than the last line might contain only one word. What should you do in this case?
- In this case, that line should be left

分析

无

代码

```
// Text Justification
// 时间复杂度O(n)，空间复杂度O(1)
public class Solution {
    public List fullJustify(String[] words, int maxWidth) {
        List result = new ArrayList<>();
        final int n = words.length;
        int begin = 0, len = 0; // 当前行的起点，当前长度
```

```
        for (int i = 0; i < n; ++i) {
            if (len + words[i].length() + (i - begin) > maxWidth) {
                result.add(connect(words, begin, i - 1, len, maxWidth, false));
                begin = i;
                len = 0;
            }
            len += words[i].length();
        }
        // 最后一行不足 maxWidth
        result.add(connect(words, begin, n - 1, len, maxWidth, true));
        return result;
    }
    /**
     * @brief 将 words[begin, end] 连成一行
     * @param[in] words 单词列表
     * @param[in] begin 开始
     * @param[in] end 结束
     * @param[in] len words[begin, end]所有单词加起来的长度
     * @param[in] L 题目规定的一行长度
     * @param[in] is_last 是否是最后一行
     * @return 对齐后的当前行
     */
    private static String connect(String[] words, int begin, int end,
                                  int len, int L, boolean is_last) {
        StringBuilder sb = new StringBuilder();
        final int n = end - begin + 1;
        for (int i = 0; i < n; ++i) {
            sb.append(words[begin + i]);
            addSpaces(sb, i, n - 1, L - len, is_last);
        }

        final int m = L - sb.length();
        for (int i = 0; i < m; ++i) sb.append(' ');
        return sb.toString();
    }

    /**
     * @brief 添加空格.
     * @param[inout]s 一行
     * @param[in] i 当前空隙的序号
     * @param[in] n 空隙总数
     * @param[in] L 总共需要添加的空额数
     * @param[in] is_last 是否是最后一行
     * @return 无
     */
    private static void addSpaces(StringBuilder sb, int i,
                                  int n, int L, boolean is_last) {
        if (n < 1 || i > n - 1) return;
        int spaces = is_last ? 1 : (L / n + (i < (L % n) ? 1 : 0));
        for (int j = 0; j < spaces; ++j) sb.append(' ');
    }
}
```

Max Points on a Line

描述

Given `n` points on a 2D plane, find the maximum number of points that lie on the same straight line.

分析

暴力枚举法。两点决定一条直线，`n`个点两两组合，可以得到$\frac{1}{2}n(n+1)$条直线，对每一条直线，判断`n`个点是否在该直线上，从而可以得到这条直线上的点的个数，选择最大的那条直线返回。复杂度`O(n^3)`。

上面的暴力枚举法以“边”为中心，再看另一种暴力枚举法，以每个“点”为中心，然后遍历剩余点，找到所有的斜率，如果斜率相同，那么一定共线对每个点，用一个哈希表，key为斜率，value为该直线上的点数，计算出哈希表后，取最大值，并更新全局最大值，最后就是结果。时间复杂度`O(n^2)`，空间复杂度`O(n)`。

以边为中心

```
// Max Points on a Line
// 暴力枚举法，以边为中心，时间复杂度O(n^3)，空间复杂度O(1)
public class Solution {
    public int maxPoints(Point[] points) {
        if (points.length < 3) return points.length;
        int result = 0;

        for (int i = 0; i < points.length - 1; i++) {
            for (int j = i + 1; j < points.length; j++) {
                int sign = 0;
                int a = 0, b = 0, c = 0;
                if (points[i].x == points[j].x) sign = 1;
                else {
                    a = points[j].x - points[i].x;
                    b = points[j].y - points[i].y;
                    c = a * points[i].y - b * points[i].x;
                }
                int count = 0;
                for (int k = 0; k < points.length; k++) {
                    if ((0 == sign && a * points[k].y == c +  b * points[k].x) ||
                            (1 == sign&&points[k].x == points[j].x))
                        count++;
                }
                if (count > result) result = count;
            }
        }
        return result;
    }
}
```

以点为中心

```java
// Max Points on a Line
// 暴力枚举，以点为中心，时间复杂度O(n^2)，空间复杂度O(n^2)
public class Solution {
    public int maxPoints(Point[] points) {
        if (points.length < 3) return points.length;
        int result = 0;

        HashMap<Double, Integer> slope_count = new HashMap<>();
        for (int i = 0; i < points.length-1; i++) {
            slope_count.clear();
            int samePointNum = 0; // 与i重合的点
            int point_max = 1;    // 和i共线的最大点数

            for (int j = i + 1; j < points.length; j++) {
                final double slope; // 斜率
                if (points[i].x == points[j].x) {
                    slope = Double.POSITIVE_INFINITY;
                    if (points[i].y == points[j].y) {
                        ++ samePointNum;
                        continue;
                    }
                } else {
                    if (points[i].y == points[j].y) {
                        // 0.0 and -0.0 is the same
                        slope = 0.0;
                    } else {
                        slope = 1.0 * (points[i].y - points[j].y) /
                                (points[i].x - points[j].x);
                    }
                }

                int count = 0;
                if (slope_count.containsKey(slope)) {
                    final int tmp = slope_count.get(slope);
                    slope_count.put(slope, tmp + 1);
                    count = tmp + 1;
                } else {
                    count = 2;
                    slope_count.put(slope, 2);
                }

                if (point_max < count) point_max = count;
            }
            result = Math.max(result, point_max + samePointNum);
        }
        return result;
    }
}
```

Java集合框架层次结构图