LintCode 题解

戴方勤 (soulmachine@gmail.com)

https://github.com/soulmachine/lintcode

最后更新 2015-5-1

版权声明

内容简介

本书的目标读者是准备去北美找工作的码农，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。

本书包含了 LintCode Online Judge(http://lintcode.com/onlinejudge) 所有题目的答案，所有代码经过精心编写，编码规范良好，适合读者反复揣摩，模仿，甚至在纸上默写。

全书的代码，使用 C++ 11 的编写，并在 LintCode Online Judge 上测试通过。本书中的代码规范，跟在公司中的工程规范略有不同，为了使代码短（方便迅速实现）：

- 所有代码都是单一文件。这是因为一般 OJ 网站，提交代码的时候只有一个文本框，如果还是按照标准做法，比如分为头文件.h 和源代码.cpp，无法在网站上提交；
- Shorter is better。能递归则一定不用栈；能用 STL 则一定不自己实现。
- 不提倡防御式编程。不需要检查 malloc()/new 返回的指针是否为 nullptr；不需要检查内部函数入口参数的有效性。

本手册假定读者已经学过《数据结构》[1]，《算法》[2] 这两门课，熟练掌握 C++ 或 Java。

GitHub 地址

本书是开源的，GitHub 地址：https://github.com/soulmachine/lintcode

北美求职微博群

我和我的小伙伴们在这里：http://q.weibo.com/1312378

[1] 《数据结构》，严蔚敏等著，清华大学出版社，http://book.douban.com/subject/2024655/

[2] 《Algorithms》，Robert Sedgewick, Addison-Wesley Professional, http://book.douban.com/subject/4854123/

目录

第 1 章
编程技巧

在判断两个浮点数 a 和 b 是否相等时，不要用 `a==b`，应该判断二者之差的绝对值 `fabs(a-b)` 是否小于某个阈值，例如 `1e-9`。

判断一个整数是否是为奇数，用 `x % 2 != 0`，不要用 `x % 2 == 1`，因为 x 可能是负数。

用 `char` 的值作为数组下标（例如，统计字符串中每个字符出现的次数），要考虑到 `char` 可能是负数。有的人考虑到了，先强制转型为 `unsigned int` 再用作下标，这仍然是错的。正确的做法是，先强制转型为 `unsigned char`，再用作下标。这涉及 C++ 整型提升的规则，就不详述了。

以下是关于 STL 使用技巧的，很多条款来自《Effective STL》这本书。

vector 和 string 优先于动态分配的数组

首先，在性能上，由于 `vector` 能够保证连续内存，因此一旦分配了后，它的性能跟原始数组相当；

其次，如果用 new，意味着你要确保后面进行了 delete，一旦忘记了，就会出现 BUG，且这样需要都写一行 delete，代码不够短；

再次，声明多维数组的话，只能一个一个 new，例如：

```
int** ary = new int*[row_num];
for(int i = 0; i < row_num; ++i)
    ary[i] = new int[col_num];
```

用 vector 的话一行代码搞定，

```
vector<vector<int> > ary(row_num, vector<int>(col_num, 0));
```

使用 reserve 来避免不必要的重新分配

第 2 章
线性表

这类题目考察线性表的操作，例如，数组，单链表，双向链表等。

2.1 数组

2.1.1 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 1

```
// LintCode, Remove Duplicates from Sorted Array
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int> &A) {
        const int n = A.size();
        if (n == 0) return 0;
        int index = 0;
        for (int i = 1; i < n; i++) {
            if (A[index] != A[i])
                A[++index] = A[i];
        }
        return index + 1;
    }
};
```

代码 2

```
// LintCode, Remove Duplicates from Sorted Array
// 使用 STL, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int> &A) {
        return distance(A.begin(), unique(A.begin(), A.end()));
    }
};
```

代码 3

```
// LintCode, Remove Duplicates from Sorted Array
// 使用 STL, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int> &A) {
        return distance(A.begin(),removeDuplicates(A.begin(), A.end(), A.begin()));
    }

    template<typename InIt, typename OutIt>
    OutIt removeDuplicates(InIt first, InIt last, OutIt output) {
        while (first != last) {
            *output++ = *first;
            first = upper_bound(first, last, *first);
        }

        return output;
    }
};
```

相关题目

- Remove Duplicates from Sorted Array II，见 §2.1.2

2.1.2 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

```
// LintCode, Remove Duplicates from Sorted Array II
// 时间复杂度 O(n), 空间复杂度 O(1)
// @author hex108 (https://github.com/hex108)
class Solution {
    public:
    int removeDuplicates(vector<int> &nums) {
        const int n = nums.size();
        if (n <= 2) return n;
        int index = 2;
        for (int i = 2; i < n; 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。

```
// LintCode, Remove Duplicates from Sorted Array II
// @author 虞航仲 (http://weibo.com/u/1666779725)
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int> &nums) {
        const int n = nums.size();
        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，见 §2.1.1

2.1.3 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

二分查找，难度主要在于左右边界的确定。

代码

```
// LintCode, Search in Rotated Sorted Array
// 时间复杂度 O(log n), 空间复杂度 O(1)
class Solution {
public:
    int search(vector<int> &A, int target) {
        int first = 0, last = A.size();
        while (first != last) {
            const int mid = first  + (last - first) / 2;
            if (A[mid] == target)
                return mid;
            if (A[first] <= A[mid]) {
                if (A[first] <= target && target < A[mid])
                    last = mid;
                else
                    first = mid + 1;
            } else {
                if (A[mid] < target && target <= A[last-1])
                    first = mid + 1;
                else
                    last = mid;
            }
        }
        return -1;
    }
};
```

相关题目

- Search in Rotated Sorted Array II，见 §2.1.4

2.1.4 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[m]>=A[l]`，那么 `[l,m]` 为递增序列的假设就不能成立了，比如 `[1,3,1,1,1]`。

如果 `A[m]>=A[l]` 不能确定递增，那就把它拆分成两个条件：

- 若 `A[m]>A[l]`，则区间 `[l,m]` 一定递增
- 若 `A[m]==A[l]` 确定不了，那就 `l++`，往下看一步即可。

代码

```
// LintCode, Search in Rotated Sorted Array II
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    bool search(vector<int> &A, int target) {
        int first = 0, last = A.size();
        while (first != last) {
            const int mid = first  + (last - first) / 2;
            if (A[mid] == target)
                return true;
            if (A[first] < A[mid]) {
                if (A[first] <= target && target < A[mid])
                    last = mid;
                else
                    first = mid + 1;
            } else if (A[first] > A[mid]) {
                if (A[mid] < target && target <= A[last-1])
                    first = mid + 1;
                else
                    last = mid;
            } else
                //skip duplicate one
                first++;
        }
        return false;
    }
};
```

相关题目

- Search in Rotated Sorted Array，见 §2.1.3

2.1.5 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”这么复杂的操作的。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

```
// LintCode, Median of Two Sorted Arrays
// 时间复杂度 O(log(m+n)), 空间复杂度 O(log(m+n))
class Solution {
public:
    double findMedianSortedArrays(const vector<int> &A, const vector<int> &B) {
        const int m = A.size();
        const int n = B.size();
        int total = m + n;
```

```
        if (total & 0x1)
            return find_kth(A.begin(), A.end(), B.begin(), B.end(), total / 2 + 1);
        else
            return (find_kth(A.begin(), A.end(), B.begin(), B.end(), total / 2)
                    + find_kth(A.begin(), A.end(), B.begin(), B.end(),
                               total / 2 + 1)) / 2.0;
    }

private:
    typedef vector<int>::const_iterator Iter;

    static int find_kth(Iter beginA, Iter endA, Iter beginB, Iter endB, int k) {
        //always assume that m is equal or smaller than n
        const int m = distance(beginA, endA);
        const int n = distance(beginB, endB);
        if (m > n) return find_kth(beginB, endB, beginA, endA, k);
        if (m == 0) return *(beginB + k - 1);
        if (k == 1) return min(*beginA, *beginB);

        //divide k into two parts
        int ia = min(k / 2, m), ib = k - ia;
        if (*(beginA + ia - 1) < *(beginB + ib - 1))
            return find_kth(beginA + ia, endA, beginB, endB, k - ia);
        else if (*(beginA + ia - 1) > *(beginB + ib - 1))
            return find_kth(beginA, endA, beginB + ib, endB, k - ib);
        else
            return *(beginA + ia - 1);
    }
};
```

相关题目

- 无

2.1.6 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 unordered_map<int, bool> used 记录每个元素是否使用，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

```
// Leet Code, Longest Consecutive Sequence
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
public:
    int longestConsecutive(const vector<int> &num) {
        unordered_map<int, bool> used;

        for (auto i : num) used[i] = false;

        int longest = 0;

        for (auto i : num) {
            if (used[i]) continue;

            int length = 1;

            used[i] = true;

            for (int j = i + 1; used.find(j) != used.end(); ++j) {
                used[j] = true;
                ++length;
            }

            for (int j = i - 1; used.find(j) != used.end(); --j) {
                used[j] = true;
                ++length;
            }

            longest = max(longest, length);
        }

        return longest;
    }
};
```

分析 2

第一直觉是个聚类的操作，应该有 union,find 的操作. 连续序列可以用两端和长度来表示. 本来用两端就可以表示，但考虑到查询的需求，将两端分别暴露出来. 用 unordered_map<int, int> map 来存储. 原始思路来自于http://discuss.lintcode.com/questions/1070/longest-consecutive-sequence

代码

```
// Leet Code, Longest Consecutive Sequence
// 时间复杂度 O(n), 空间复杂度 O(n)
// Author: @advancedxy
class Solution {
public:
    int longestConsecutive(vector<int> &num) {
        unordered_map<int, int> map;
        int size = num.size();
        int l = 1;
        for (int i = 0; i < size; i++) {
            if (map.find(num[i]) != map.end()) continue;
            map[num[i]] = 1;
            if (map.find(num[i] - 1) != map.end()) {
                l = max(l, mergeCluster(map, num[i] - 1, num[i]));
            }
            if (map.find(num[i] + 1) != map.end()) {
                l = max(l, mergeCluster(map, num[i], num[i] + 1));
            }
        }
        return size == 0 ? 0 : l;
    }

private:
    int mergeCluster(unordered_map<int, int> &map, int left, int right) {
        int upper = right + map[right] - 1;
        int lower = left - map[left] + 1;
        int length = upper - lower + 1;
        map[upper] = length;
        map[lower] = length;
        return length;
    }
};
```

相关题目

- 无

2.1.7 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

```
//LintCode, Two Sum
// 方法 2: hash。用一个哈希表，存储每个数对应的下标
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
public:
    vector<int> twoSum(vector<int> &num, int target) {
        unordered_map<int, int> mapping;
        vector<int> result;
        for (int i = 0; i < num.size(); i++) {
            mapping[num[i]] = i;
        }
        for (int i = 0; i < num.size(); i++) {
            const int gap = target - num[i];
            if (mapping.find(gap) != mapping.end() && mapping[gap] > i) {
                result.push_back(i + 1);
                result.push_back(mapping[gap] + 1);
                break;
            }
        }
        return result;
    }
};
```

相关题目

- 3Sum, 见 §2.1.8
- 3Sum Closest, 见 §2.1.9
- 4Sum, 见 §2.1.10

2.1.8 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

```
// LintCode, 3Sum
// 先排序，然后左右夹逼，注意跳过重复的数，时间复杂度 O(n^2)，空间复杂度 O(1)
class Solution {
    public:
    vector<vector<int>> threeSum(vector<int>& num) {
        vector<vector<int>> result;
        if (num.size() < 3) return result;
        sort(num.begin(), num.end());
        const int target = 0;

        auto last = num.end();
        for (auto i = num.begin(); i < last-2; ++i) {
            auto j = i+1;
            if (i > num.begin() && *i == *(i-1)) continue;
            auto k = last-1;
            while (j < k) {
                if (*i + *j + *k < target) {
                    ++j;
                    while(*j == *(j - 1) && j < k) ++j;
                } else if (*i + *j + *k > target) {
                    --k;
                    while(*k == *(k + 1) && j < k) --k;
                } else {
                    result.push_back({ *i, *j, *k });
                    ++j;
                    --k;
                    while(*j == *(j - 1) && *k == *(k + 1) && j < k) ++j;
                }
            }
        }
        return result;
    }
};
```

相关题目

- Two sum, 见 §2.1.7
- 3Sum Closest, 见 §2.1.9
- 4Sum, 见 §2.1.10

2.1.9 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

```
// LintCode, 3Sum Closest
// 先排序，然后左右夹逼，时间复杂度 O(n^2)，空间复杂度 O(1)
class Solution {
public:
    int threeSumClosest(vector<int>& num, int target) {
        int result = 0;
        int min_gap = INT_MAX;

        sort(num.begin(), num.end());

        for (auto a = num.begin(); a != prev(num.end(), 2); ++a) {
            auto b = next(a);
            auto c = prev(num.end());

            while (b < c) {
                const int sum = *a + *b + *c;
                const int gap = abs(sum - target);

                if (gap < min_gap) {
                    result = sum;
                    min_gap = gap;
                }

                if (sum < target) ++b;
                else              --c;
            }
        }
```

```
        return result;
    }
};
```

相关题目

- Two sum, 见 §2.1.7
- 3Sum, 见 §2.1.8
- 4Sum, 见 §2.1.10

2.1.10 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

```
// LintCode, 4Sum
// 先排序，然后左右夹逼，时间复杂度 O(n^3)，空间复杂度 O(1)
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& num, int target) {
        vector<vector<int>> result;
        if (num.size() < 4) return result;
        sort(num.begin(), num.end());

        auto last = num.end();
        for (auto a = num.begin(); a < prev(last, 3); ++a) {
```

```
            for (auto b = next(a); b < prev(last, 2); ++b) {
                auto c = next(b);
                auto d = prev(last);
                while (c < d) {
                    if (*a + *b + *c + *d < target) {
                        ++c;
                    } else if (*a + *b + *c + *d > target) {
                        --d;
                    } else {
                        result.push_back({ *a, *b, *c, *d });
                        ++c;
                        --d;
                    }
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

map 做缓存

```
// LintCode, 4Sum
// 用一个 hashmap 先缓存两个数的和
// 时间复杂度, 平均 O(n^2), 最坏 O(n^4), 空间复杂度 O(n^2)
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {
        vector<vector<int>> result;
        if (num.size() < 4) return result;
        sort(num.begin(), num.end());

        unordered_map<int, vector<pair<int, int> > > cache;
        for (size_t a = 0; a < num.size(); ++a) {
            for (size_t b = a + 1; b < num.size(); ++b) {
                cache[num[a] + num[b]].push_back(pair<int, int>(a, b));
            }
        }

        for (int c = 0; c < num.size(); ++c) {
            for (size_t d = c + 1; d < num.size(); ++d) {
                const int key = target - num[c] - num[d];
                if (cache.find(key) == cache.end()) continue;

                const auto& vec = cache[key];
                for (size_t k = 0; k < vec.size(); ++k) {
                    if (c <= vec[k].second)
                        continue; // 有重叠

                    result.push_back( { num[vec[k].first],
                            num[vec[k].second], num[c], num[d] });
```

```
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

multimap

```
// LintCode, 4Sum
// 用一个 hashmap 先缓存两个数的和
// 时间复杂度 O(n^2), 空间复杂度 O(n^2)
// @author 龚陆安 (http://weibo.com/luangong)
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& num, int target) {
        vector<vector<int>> result;
        if (num.size() < 4) return result;
        sort(num.begin(), num.end());

        unordered_multimap<int, pair<int, int>> cache;
        for (int i = 0; i + 1 < num.size(); ++i)
            for (int j = i + 1; j < num.size(); ++j)
                cache.insert(make_pair(num[i] + num[j], make_pair(i, j)));

        for (auto i = cache.begin(); i != cache.end(); ++i) {
            int x = target - i->first;
            auto range = cache.equal_range(x);
            for (auto j = range.first; j != range.second; ++j) {
                auto a = i->second.first;
                auto b = i->second.second;
                auto c = j->second.first;
                auto d = j->second.second;
                if (a != c && a != d && b != c && b != d) {
                    vector<int> vec = { num[a], num[b], num[c], num[d] };
                    sort(vec.begin(), vec.end());
                    result.push_back(vec);
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

方法 4

```
// LintCode, 4Sum
// 先排序, 然后左右夹逼, 时间复杂度 O(n^3logn), 空间复杂度 O(1), 会超时
```

```
// 跟方法 1 相比，表面上优化了，实际上更慢了，切记!
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& num, int target) {
        vector<vector<int>> result;
        if (num.size() < 4) return result;
        sort(num.begin(), num.end());

        auto last = num.end();
        for (auto a = num.begin(); a < prev(last, 3);
                a = upper_bound(a, prev(last, 3), *a)) {
            for (auto b = next(a); b < prev(last, 2);
                    b = upper_bound(b, prev(last, 2), *b)) {
                auto c = next(b);
                auto d = prev(last);
                while (c < d) {
                    if (*a + *b + *c + *d < target) {
                        c = upper_bound(c, d, *c);
                    } else if (*a + *b + *c + *d > target) {
                        d = prev(lower_bound(c, d, *d));
                    } else {
                        result.push_back({ *a, *b, *c, *d });
                        c = upper_bound(c, d, *c);
                        d = prev(lower_bound(c, d, *d));
                    }
                }
            }
        }
        return result;
    }
};
```

相关题目

- Two sum, 见 §2.1.7
- 3Sum, 见 §2.1.8
- 3Sum Closest, 见 §2.1.9

2.1.11 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 1

```
// LintCode, Remove Element
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int removeElement(vector<int> &A, int elem) {
        const int n = A.size();
        int index = 0;
        for (int i = 0; i < n; ++i) {
            if (A[i] != elem) {
                A[index++] = A[i];
            }
        }
        return index;
    }
};
```

代码 2

```
// LintCode, Remove Element
// 使用 remove(), 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int removeElement(vector<int> &A, int elem) {
        return distance(A.begin(), remove(A.begin(), A.end(), elem));
    }
};
```

相关题目

- 无

2.1.12 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).

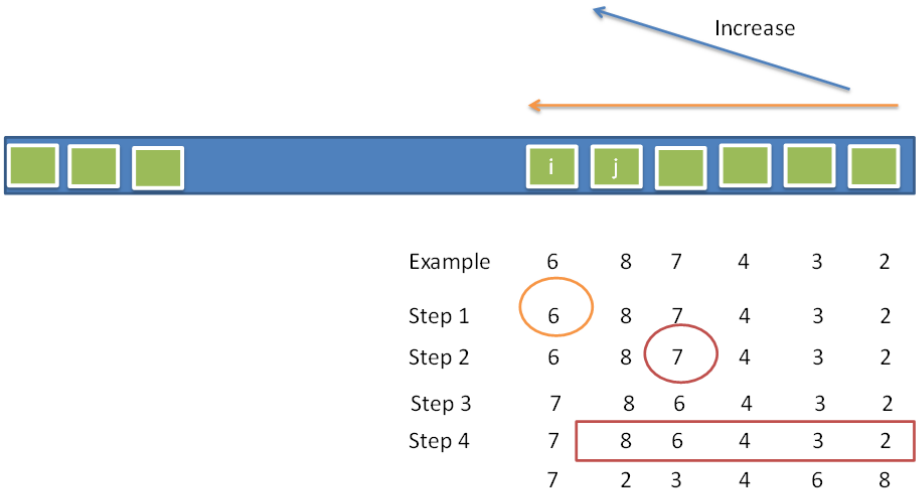
The replacement must be in-place, do not allocate extra memory.

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

算法过程如图 2-1所示（来自 http://fisherlei.blogspot.com/2012/12/lintcode-next-permutation.html）。



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.

图 2-1 下一个排列算法流程

代码

```
// LintCode, Next Permutation
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    void nextPermutation(vector<int> &num) {
        next_permutation(num.begin(), num.end());
    }

    template<typename BidiIt>
    bool next_permutation(BidiIt first, BidiIt last) {
        // Get a reversed range to simplify reversed traversal.
        const auto rfirst = reverse_iterator<BidiIt>(last);
        const auto rlast = reverse_iterator<BidiIt>(first);

        // Begin from the second last element to the first element.
        auto pivot = next(rfirst);
```

```
        // Find `pivot`, which is the first element that is no less than its
        // successor.  `Prev` is used since `pivort` is a `reversed_iterator`.
        while (pivot != rlast && *pivot >= *prev(pivot))
        ++pivot;

        // No such elemenet found, current sequence is already the largest
        // permutation, then rearrange to the first permutation and return false.
        if (pivot == rlast) {
            reverse(rfirst, rlast);
            return false;
        }

        // Scan from right to left, find the first element that is greater than
        // `pivot`.
        auto change = find_if(rfirst, pivot, bind1st(less<int>(), *pivot));

        swap(*change, *pivot);
        reverse(rfirst, pivot);

        return true;
    }
};
```

相关题目

- Permutation Sequence, 见 §2.1.13
- Permutations, 见 §8.3
- Permutations II, 见 §8.4
- Combinations, 见 §8.5

2.1.13 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

简单的，可以用暴力枚举法，调用 $k-1$ 次 `next_permutation()`。

暴力枚举法把前 k 个排列都求出来了，比较浪费，而我们只需要第 k 个排列。

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：

$$
\begin{aligned}
k_2 &= k\%(n-1)! \\
a_2 &= k_2/(n-2)! \\
&\cdots \\
k_{n-1} &= k_{n-2}\%2! \\
a_{n-1} &= k_{n-1}/1! \\
a_n &= 0
\end{aligned}
$$

使用 next_permutation()

```
// LintCode, Permutation Sequence
// 使用 next_permutation(), TLE
class Solution {
public:
    string getPermutation(int n, int k) {
        string s(n, '0');
        for (int i = 0; i < n; ++i)
            s[i] += i+1;
        for (int i = 0; i < k-1; ++i)
            next_permutation(s.begin(), s.end());
        return s;
    }

    template<typename BidiIt>
    bool next_permutation(BidiIt first, BidiIt last) {
        // 代码见上一题 Next Permutation
    }
};
```

康托编码

```
// LintCode, Permutation Sequence
// 康托编码, 时间复杂度 O(n), 空间复杂度 O(1)
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; ++i)
            result *= i;
        return result;
    }

    // seq 已排好序, 是第一个排列
    template<typename Sequence>
    Sequence kth_permutation(const Sequence &seq, int k) {
        const int n = seq.size();
        Sequence S(seq);
        Sequence 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, 见 §2.1.12
- Permutations, 见 §8.3
- Permutations II, 见 §8.4
- Combinations, 见 §8.5

2.1.14 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 '.'.

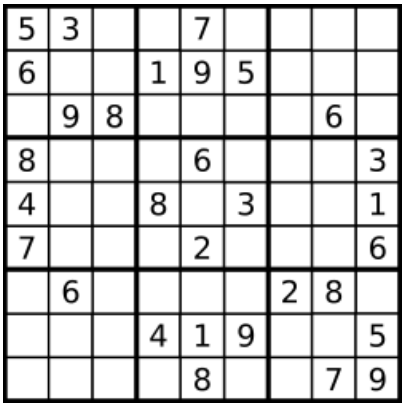


图 2-2 A partially filled sudoku which is valid

分析

细节实现题。

代码

```
// LintCode, Valid Sudoku
// 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    bool isValidSudoku(const vector<vector<char>>& board) {
        bool used[9];

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

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

            fill(used, used + 9, 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) {
                fill(used, used + 9, 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;
        }

        bool check(char ch, bool used[9]) {
            if (ch == '.') return true;

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

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

相关题目

- Sudoku Solver, 见 §10.10

2.1.15 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.

图 2-3 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

```
// LintCode, Trapping Rain Water
// 思路 1, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
    public:
    int trapRainWater(const vector<int> &heights) {
        const int n = heights.size();
        auto max_left = vector<int>(n, 0);
        auto max_right = vector<int>(n, 0);

        for (int i = 1; i < n; i++) {
            max_left[i] = max(max_left[i - 1], heights[i - 1]);
            max_right[n - 1 - i] = max(max_right[n - i], heights[n - i]);
        }
        int sum = 0;
        for (int i = 0; i < n; i++) {
            int height = min(max_left[i], max_right[i]);
            if (height > heights[i]) {
                sum += height - heights[i];
            }
        }

        return sum;
    }
};
```

代码 2

```
// LintCode, Trapping Rain Water
// 思路 2, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    int trapRainWater(const vector<int> &heights) {
        const int n = heights.size();
        int max = 0; // 最高的柱子, 将数组分为两半
        for (int i = 0; i < n; i++)
        if (heights[i] > heights[max]) max = i;
```

```
        int water = 0;
        for (int i = 0, peak = 0; i < max; i++) {
            if (heights[i] > peak) peak = heights[i];
            else water += peak - heights[i];
        }
        for (int i = n - 1, top = 0; i > max; i--) {
            if (heights[i] > top) top = heights[i];
            else water += top - heights[i];
        }
        return water;
    }
};
```

代码 3

第三种解法，用一个栈辅助，小于栈顶的元素压入，大于等于栈顶就把栈里所有小于或等于当前值的元素全部出栈处理掉。

```
// LintCode, Trapping Rain Water
// 用一个栈辅助，小于栈顶的元素压入，大于等于栈顶就把栈里所有小于或
// 等于当前值的元素全部出栈处理掉，计算面积，最后把当前元素入栈
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    int trapRainWater(const vector<int> &heights) {
        const int n = heights.size();
        stack<pair<int, int>> s;
        int water = 0;

        for (int i = 0; i < n; ++i) {
            int height = 0;

            while (!s.empty()) { // 将栈里比当前元素矮或等高的元素全部处理掉
                int bar = s.top().first;
                int pos = s.top().second;
                // bar, height, a[i] 三者夹成的凹陷
                water += (min(bar, heights[i]) - height) * (i - pos - 1);
                height = bar;

                if (heights[i] < bar) // 碰到了比当前元素高的，跳出循环
                    break;
                else
                    s.pop(); // 弹出栈顶，因为该元素处理完了，不再需要了
            }

            s.push(make_pair(heights[i], i));
        }

        return water;
    }
};
```

相关题目

- Container With Most Water, 见 §12.6
- Largest Rectangle in Histogram, 见 §4.1.3

2.1.16 Rotate Image

描述

You are given an $n \times n$ 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。

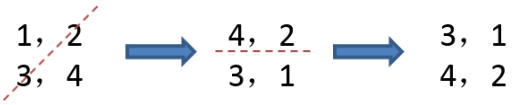


图 2-4 Rotate Image

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

```
// LintCode, Rotate Image
// 思路 1, 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        const int n = matrix.size();

        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], matrix[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], matrix[n - 1 - i][j]);
    }
};
```

代码 2

```
// LintCode, Rotate Image
// 思路 2, 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        const int n = matrix.size();

        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], matrix[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], matrix[j][i]);
    }
};
```

相关题目

- 无

2.1.17 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 1

```
// LintCode, Plus One
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
        return digits;
    }
private:
    // 0 <= digit <= 9
    void add(vector<int> &digits, int digit) {
        int c = digit;  // carry, 进位

        for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
            *it += c;
```

```
            c = *it / 10;
            *it %= 10;
        }

        if (c > 0) digits.insert(digits.begin(), 1);
    }
};
```

代码 2

```
// LintCode, Plus One
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
        return digits;
    }
private:
    // 0 <= digit <= 9
    void add(vector<int> &digits, int digit) {
        int c = digit;  // carry, 进位

        for_each(digits.rbegin(), digits.rend(), [&c](int &d){
            d += c;
            c = d / 10;
            d %= 10;
        });

        if (c > 0) digits.insert(digits.begin(), 1);
    }
};
```

相关题目

- 无

2.1.18 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

```
// LintCode, 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

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

相关题目

- Decode Ways, 见 §13.10

2.1.19 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。如图 §2-5所示。

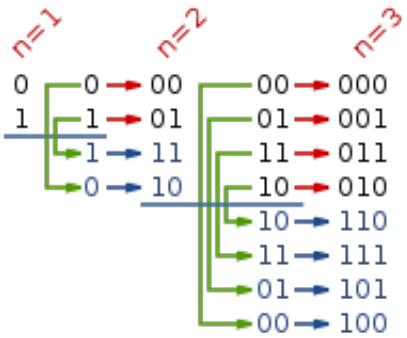


图 2-5 The first few steps of the reflect-and-prefix method.

数学公式

```
// LintCode, Gray Code
// 数学公式, 时间复杂度 O(2^n), 空间复杂度 O(1)
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> result;
        const size_t size = 1 << n;  // 2^n
        result.reserve(size);
        for (size_t i = 0; i < size; ++i)
            result.push_back(binary_to_gray(i));
        return result;
    }
private:
    static unsigned int binary_to_gray(unsigned int n) {
        return n ^ (n >> 1);
    }
};
```

Reflect-and-prefix method

```
// LintCode, Gray Code
// reflect-and-prefix method
// 时间复杂度 O(2^n), 空间复杂度 O(1)
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> result;
        result.reserve(1<<n);
        result.push_back(0);
        for (int i = 0; i < n; i++) {
            const int highest_bit = 1 << i;
            for (int j = result.size() - 1; j >= 0; j--) // 要反着遍历, 才能对称
                result.push_back(highest_bit | result[j]);
        }
        return result;
```

```
    }
};
```

相关题目

- 无

2.1.20 Set Matrix Zeroes

描述

Given a $m \times n$ 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

```
// LintCode, Set Matrix Zeroes
// 时间复杂度 O(m*n), 空间复杂度 O(m+n)
class Solution {
    public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        if (m == 0) return;
        const size_t n = matrix[0].size();

        vector<bool> row(m, false); // 标记该行是否存在 0
        vector<bool> col(n, false); // 标记该列是否存在 0

        for (size_t i = 0; i < m; ++i) {
            for (size_t j = 0; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    row[i] = col[j] = true;
                }
            }
        }
        for (size_t i = 0; i < m; ++i) {
            if (row[i])
            fill(&matrix[i][0], &matrix[i][0] + n, 0);
        }
```

```
        for (size_t j = 0; j < n; ++j) {
            if (col[j]) {
                for (size_t i = 0; i < m; ++i) {
                    matrix[i][j] = 0;
                }
            }
        }
    }
};
```

代码 2

```
// LintCode, Set Matrix Zeroes
// 时间复杂度 O(m*n), 空间复杂度 O(1)
class Solution {
    public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        if (m == 0) return;
        const size_t n = matrix[0].size();
        bool row_has_zero = false; // 第一行是否存在 0
        bool col_has_zero = false; // 第一列是否存在 0

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

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

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

相关题目

- 无

2.1.21 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

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

相关题目

- 无

2.1.22 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

```
// LintCode, Candy
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    int candy(vector<int> &ratings) {
        const int n = ratings.size();
        vector<int> increment(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] = 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] = max(inc++, increment[i]);
            else
                inc = 1;
        }
        // 初始值为 n, 因为每个小朋友至少一颗糖
        return accumulate(&increment[0], &increment[0]+n, n);
    }
};
```

递归版

```
// LintCode, Candy
// 备忘录法, 时间复杂度 O(n), 空间复杂度 O(n)
// @author fancymouse (http://weibo.com/u/1928162822)
```

```
class Solution {
public:
    int candy(const vector<int>& ratings) {
        vector<int> f(ratings.size());
        int sum = 0;
        for (int i = 0; i < ratings.size(); ++i)
            sum += solve(ratings, f, i);
        return sum;
    }
    int solve(const vector<int>& ratings, vector<int>& f, int i) {
        if (f[i] == 0) {
            f[i] = 1;
            if (i > 0 && ratings[i] > ratings[i - 1])
                f[i] = max(f[i], solve(ratings, f, i - 1) + 1);
            if (i < ratings.size() - 1 && ratings[i] > ratings[i + 1])
                f[i] = max(f[i], solve(ratings, f, i + 1) + 1);
        }
        return f[i];
    }
};
```

相关题目

- 无

2.1.23 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 1

```
// LintCode, Single Number
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    int singleNumber(const vector<int> &A) {
        int x = 0;
        for (auto i : A) {
            x ^= i;
        }
```

```
        return x;
    }
};
```

代码 2

```
// LintCode, Single Number
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    int singleNumber(const vector<int> &A) {
        return accumulate(A.begin(), A.end(), 0, bit_xor<int>());
    }
};
```

相关题目

- Single Number II, 见 §2.1.24

2.1.24 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

```
// LintCode, Single Number II
// 方法 1, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    int singleNumberII(const vector<int> &A) {
        const int W = sizeof(int) * 8; // 一个整数的 bit 数, 即整数字长
```

```
        int count[W];  // count[i] 表示在在 i 位出现的 1 的次数
        fill_n(&count[0], W, 0);
        for (int i = 0; i < A.size(); i++) {
            for (int j = 0; j < W; j++) {
                count[j] += (A[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

```
// LintCode, Single Number II
// 方法 2, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    int singleNumberII(const vector<int> &A) {
        int one = 0, two = 0, three = 0;
        for (int i = 0; i < A.size(); ++i) {
            two |= (one & A[i]);
            one ^= A[i];
            three = ~(one & two);
            one &= three;
            two &= three;
        }

        return one;
    }
};
```

相关题目

- Single Number, 见 §2.1.23

2.2 单链表

单链表节点的定义如下:

```
// 单链表节点
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) { }
};
```

2.2.1 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（见 §3.4）很类似

代码

```
// LintCode, Add Two Numbers
// 跟 Add Binary 很类似
// 时间复杂度 O(m+n), 空间复杂度 O(1)
class Solution {
public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        ListNode dummy(-1); // 头节点
        int carry = 0;
        ListNode *prev = &dummy;
        for (ListNode *pa = l1, *pb = l2;
             pa != nullptr || pb != nullptr;
             pa = pa == nullptr ? nullptr : pa->next,
             pb = pb == nullptr ? nullptr : pb->next,
             prev = prev->next) {
            const int ai = pa == nullptr ? 0 : pa->val;
            const int bi = pb == nullptr ? 0 : pb->val;
            const 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, 见 §3.4

2.2.2 Reverse Linked List

描述

Reverse a single linked list.

Example: Given a linked list For linked list `1->2->3->nullptr`, return `3->2->1->nullptr`.

分析

这是一道很基本的单链表题目，考察你对单链表的理解程度。

递归版

```
class Solution {
    public:
    ListNode* reverse(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return head;
        ListNode* tail = head->next;
        ListNode *head_of_reversed_tail = reverse(tail);
        tail->next = head;
        head->next = nullptr;
        return head_of_reversed_tail;
    }
};
```

迭代版

```
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        ListNode* prev = nullptr;
        while (head) {
            ListNode* next = head->next;
            head->next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }
};
```

相关题目

- Reverse Linked List II, 见 §2.2.3

2.2.3 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

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

        ListNode *prev = &dummy;
        for (int i = 0; i < m-1; ++i)
            prev = prev->next;
        ListNode* const 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;
    }
};
```

相关题目

- Reverse Linked List, 见 §2.2.2

2.2.4 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

```
// LintCode, 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(-1); // 头结点
        ListNode right_dummy(-1); // 头结点

        auto left_cur = &left_dummy;
        auto right_cur = &right_dummy;

        for (ListNode *cur = head; cur; 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 = nullptr;

        return left_dummy.next;
    }
};
```

相关题目

- 无

2.2.5 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

```
// LintCode, 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) return head;
        ListNode dummy(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 == nullptr) return;

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

迭代版

```
// LintCode, 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 == nullptr) return nullptr;
```

```
        for (ListNode *prev = head, *cur = head->next; cur; cur = cur->next) {
            if (prev->val == cur->val) {
                prev->next = cur->next;
                delete cur;
            } else {
                prev = cur;
            }
        }
        return head;
    }
};
```

相关题目

- Remove Duplicates from Sorted List II，见 §2.2.6

2.2.6 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

```
// LintCode, 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 || !head->next) return head;

        ListNode *p = head->next;
        if (head->val == p->val) {
            while (p && head->val == p->val) {
                ListNode *tmp = p;
                p = p->next;
                delete tmp;
            }
            delete head;
            return deleteDuplicates(p);
```

```
        } else {
            head->next = deleteDuplicates(head->next);
            return head;
        }
    }
};
```

迭代版

```
// LintCode, Remove Duplicates from Sorted List II
// 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 == nullptr) return head;

        ListNode dummy(INT_MIN); // 头结点
        dummy.next = head;
        ListNode *prev = &dummy, *cur = head;
        while (cur != nullptr) {
            bool duplicated = false;
            while (cur->next != nullptr && cur->val == cur->next->val) {
                duplicated = true;
                ListNode *temp = cur;
                cur = cur->next;
                delete temp;
            }
            if (duplicated) { // 删除重复的最后一个元素
                ListNode *temp = cur;
                cur = cur->next;
                delete temp;
                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，见 §2.2.5

2.2.7 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

```
// LintCode, 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 == nullptr || k == 0) return head;

        int len = 1;
        ListNode* p = head;
        while (p->next) { // 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 = nullptr; // 断开环
        return head;
    }
};
```

相关题目

- 无

2.2.8 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

```
// LintCode, 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{-1, head};
        ListNode *p = &dummy, *q = &dummy;

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

        while(q->next) { // 一起走
            p = p->next;
            q = q->next;
        }
        ListNode *tmp = p->next;
        p->next = p->next->next;
        delete tmp;
        return dummy.next;
    }
};
```

相关题目

- 无

2.2.9 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

```
// LintCode, 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 == nullptr || head->next == nullptr) return head;
        ListNode dummy(-1);
        dummy.next = head;

        for(ListNode *prev = &dummy, *cur = prev->next, *next = cur->next;
                next;
                prev = cur, cur = cur->next, next = cur ? cur->next: nullptr) {
            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。

```
// LintCode, 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 && p->next) {
            swap(p->val, p->next->val);
            p = p->next->next;
        }

        return head;
    }
};
```

相关题目

- Reverse Nodes in k-Group, 见 §2.2.10

2.2.10 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

```
// LintCode, 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 == nullptr || head->next == nullptr || k < 2)
            return head;

        ListNode *next_group = head;
        for (int i = 0; i < k; ++i) {
            if (next_group)
                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 ? prev : new_next_group;
            prev = cur;
            cur = next;
        }
        return prev; // prev will be the new head of this group
    }
};
```

迭代版

```
// LintCode, 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 == nullptr || head->next == nullptr || k < 2) return head;
        ListNode dummy(-1);
        dummy.next = head;
```

```
        for(ListNode *prev = &dummy, *end = head; end; end = prev->next) {
            for (int i = 1; i < k && end; i++)
                end = end->next;
            if (end  == nullptr) 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 ? next->next : nullptr) {
            cur->next = p;
        }
        begin->next = end_next;
        prev->next = end;
        return begin;
    }
};
```

相关题目

- Swap Nodes in Pairs, 见 §2.2.9

2.2.11 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

```
// LintCode, 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 != nullptr; ) {
            RandomListNode* node = new RandomListNode(cur->label);
            node->next = cur->next;
            cur->next = node;
            cur = node->next;
        }

        for (RandomListNode* cur = head; cur != nullptr; ) {
            if (cur->random != NULL)
                cur->next->random = cur->random->next;
            cur = cur->next->next;
        }

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

相关题目

- 无

2.2.12 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 http://lintcode.com/2010/09/detecting-loop-in-singly-linked-list.html

代码

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

相关题目

- Linked List Cycle II, 见 §2.2.13

2.2.13 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：

$$\begin{aligned} 2s &= s + nr \\ s &= nr \end{aligned}$$

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

$$\begin{aligned} x + a &= nr = (n–1)r + r = (n-1)r + L - x \\ x &= (n-1)r + (L–x–a) \end{aligned}$$

$L–x–a$ 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

```
//LintCode, 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 && fast->next) {
            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 nullptr;
    }
};
```

相关题目

- Linked List Cycle, 见 §2.2.12

2.2.14 Reorder List

描述

Given a singly linked list $L: L_0 \to L_1 \to \cdots \to L_{n-1} \to L_n$, reorder it to: $L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to \cdots$

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

```
// LintCode, 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 == nullptr || head->next == nullptr) return;

        ListNode *slow = head, *fast = head, *prev = nullptr;
        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = nullptr; // cut at middle

        slow = reverse(slow);

        // merge two lists
        ListNode *curr = head;
        while (curr->next) {
            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 == nullptr || head->next == nullptr) return head;

        ListNode *prev = head;
        for (ListNode *curr = head->next, *next = curr->next; curr;
            prev = curr, curr = next, next = next ? next->next : nullptr) {
                curr->next = prev;
        }
        head->next = nullptr;
        return prev;
    }
};
```

相关题目

- 无

2.2.15 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，我们使用一个双向链表 (`std::list`) 和一个哈希表 (`std::unordered_map`)，因为：

- 哈希表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
- 双向链表插入和删除效率高，单向链表插入和删除时，还要查找节点的前驱节点

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

代码

```
// LintCode, LRU Cache
// 时间复杂度 O(logn), 空间复杂度 O(n)
class LRUCache{
private:
    struct CacheNode {
        int key;
        int value;
        CacheNode(int k, int v) :key(k), value(v){}
    };
public:
    LRUCache(int capacity) {
        this->capacity = capacity;
    }

    int get(int key) {
        if (cacheMap.find(key) == cacheMap.end()) return -1;

        // 把当前访问的节点移到链表头部, 并且更新 map 中该节点的地址
        cacheList.splice(cacheList.begin(), cacheList, cacheMap[key]);
        cacheMap[key] = cacheList.begin();
        return cacheMap[key]->value;
    }

    void set(int key, int value) {
        if (cacheMap.find(key) == cacheMap.end()) {
            if (cacheList.size() == capacity) { //删除链表尾部节点（最少访问的节点）
```

```
                cacheMap.erase(cacheList.back().key);
                cacheList.pop_back();
            }
            // 插入新节点到链表头部，并且在 map 中增加该节点
            cacheList.push_front(CacheNode(key, value));
            cacheMap[key] = cacheList.begin();
        } else {
            //更新节点的值，把当前访问的节点移到链表头部，并且更新 map 中该节点的地址
            cacheMap[key]->value = value;
            cacheList.splice(cacheList.begin(), cacheList, cacheMap[key]);
            cacheMap[key] = cacheList.begin();
        }
    }
private:
    list<CacheNode> cacheList;
    unordered_map<int, list<CacheNode>::iterator> cacheMap;
    int capacity;
};
```

相关题目

- 无

第 3 章
字符串

3.1 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

```
// Leet Code, Valid Palindrome
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    bool isPalindrome(string s) {
        transform(s.begin(), s.end(), s.begin(), ::tolower);
        auto left = s.begin(), right = prev(s.end());
        while (left < right) {
            if (!::isalnum(*left))  ++left;
            else if (!::isalnum(*right)) --right;
            else if (*left != *right) return false;
            else{ left++, right--; }
        }
        return true;
    }
};
```

相关题目

- Palindrome Number, 见 §15.2

3.2 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

```
// LintCode, 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(const char *haystack, const char *needle) {
        if (haystack == nullptr || needle == nullptr) return -1;
        // if needle is empty return the full string
        if (!*needle) return 0;

        const char *p1;
        const char *p2;
        const char *p1_advance = haystack;
        for (p2 = &needle[1]; *p2; ++p2) {
            p1_advance++;   // advance p1_advance M-1 times
        }

        for (p1 = haystack; *p1_advance; p1_advance++) {
            char *p1_old = (char*) p1;
            p2 = needle;
            while (*p1 && *p2 && *p1 == *p2) {
                p1++;
                p2++;
            }
            if (!*p2) return p1_old - haystack;

            p1 = p1_old + 1;
        }
        return -1;
    }
};
```

KMP

```
// LintCode, Implement strStr()
// KMP, 时间复杂度 O(N+M), 空间复杂度 O(M)
class Solution {
public:
    int strStr(const char *haystack, const char *needle) {
        if (haystack == nullptr || needle == nullptr) return -1;
        return kmp(haystack, needle);
    }
    private:
    /*
    * @brief 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
    */
    static void compute_prefix(const char *pattern, int next[]) {
        int i;
        int j = -1;
        const int m = strlen(pattern);

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

            if (pattern[i] == pattern[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
    */
    static int kmp(const char *text, const char *pattern) {
        int i;
        int j = -1;
        const int n = strlen(text);
        const int m = strlen(pattern);
        if (n == 0 && m == 0) return 0; /* "","" */
        if (m == 0) return 0;  /* "a","" */
        int *next = (int*)malloc(sizeof(int) * m);

        compute_prefix(pattern, next);

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

```
            if (text[i] == pattern[j + 1]) j++;
            if (j == m - 1) {
                free(next);
                return i-j;
            }
        }

        free(next);
        return -1;
    }
};
```

相关题目

- String to Integer (atoi)，见 §3.3

3.3 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

```
// LintCode, String to Integer (atoi)
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int atoi(const string &str) {
        int num = 0;
        int sign = 1;
        const int n = str.size();
        int i = 0;

        while (str[i] == ' ' && i < n) i++;

        if (str[i] == '+') {
            i++;
        } else if (str[i] == '-') {
            sign = -1;
            i++;
        }

        for (; i < n; i++) {
            if (str[i] < '0' || str[i] > '9')
                break;
            if (num > INT_MAX / 10 ||
                (num == INT_MAX / 10 &&
                 (str[i] - '0') > INT_MAX % 10)) {
                return sign == -1 ? INT_MIN : INT_MAX;
            }
            num = num * 10 + str[i] - '0';
        }
        return num * sign;
    }
};
```

相关题目

- Implement strStr()，见 §3.2

3.4 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

```
//LintCode, 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) {
        string result;
        const size_t n = a.size() > b.size() ? a.size() : b.size();
        reverse(a.begin(), a.end());
        reverse(b.begin(), b.end());
        int carry = 0;
        for (size_t i = 0; i < n; i++) {
            const int ai = i < a.size() ? a[i] - '0' : 0;
            const int bi = i < b.size() ? b[i] - '0' : 0;
            const int val = (ai + bi + carry) % 2;
            carry = (ai + bi + carry) / 2;
            result.insert(result.begin(), val + '0');
        }
        if (carry == 1) {
            result.insert(result.begin(), '1');
        }
        return result;
    }
};
```

相关题目

- Add Two Numbers, 见 §2.2.1

3.5 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

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

思路三：Manacher's Algorithm, 复杂度 $O(n)$。详细解释见 http://lintcode.com/2011/11/longest-palindromic-substring-part-ii.html 。

备忘录法

```
// LintCode, 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)
typedef string::const_iterator Iterator;

namespace std {
template<>
struct hash<pair<Iterator, Iterator>> {
    size_t operator()(pair<Iterator, Iterator> const& p) const {
        return ((size_t) &(*p.first)) ^ ((size_t) &(*p.second));
    }
};
}

class Solution {
public:
    string longestPalindrome(string const& s) {
        cache.clear();
        return cachedLongestPalindrome(s.begin(), s.end());
    }

private:
    unordered_map<pair<Iterator, Iterator>, string> cache;

    string longestPalindrome(Iterator first, Iterator last) {
        size_t length = distance(first, last);

        if (length < 2) return string(first, last);
```

```
        auto s = cachedLongestPalindrome(next(first), prev(last));

        if (s.length() == length - 2 && *first == *prev(last))
            return string(first, last);

        auto s1 = cachedLongestPalindrome(next(first), last);
        auto s2 = cachedLongestPalindrome(first, prev(last));

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

    string cachedLongestPalindrome(Iterator first, Iterator last) {
        auto key = make_pair(first, last);
        auto pos = cache.find(key);

        if (pos != cache.end()) return pos->second;
        else return cache[key] = longestPalindrome(first, last);
    }
};
```

动规

```
// LintCode, 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(string s) {
        const int n = s.size();
        bool f[n][n];
        fill_n(&f[0][0], n * n, false);
        // 用 vector 会超时
        //vector<vector<bool> > f(n, vector<bool>(n, false));
        size_t max_len = 1, start = 0;  // 最长回文子串的长度, 起点

        for (size_t i = 0; i < s.size(); i++) {
            f[i][i] = true;
            for (size_t j = 0; j < i; j++) {  // [j, i]
                f[j][i] = (s[j] == s[i] && (i - j < 2 || f[j + 1][i - 1]));
                if (f[j][i] && max_len < (i - j + 1)) {
                    max_len = i - j + 1;
                    start = j;
                }
            }
        }
        return s.substr(start, max_len);
    }
};
```

Manacher's Algorithm

```
// LintCode, Longest Palindromic Substring
// Manacher's Algorithm
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    // 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
    string preProcess(string s) {
        int n = s.length();
        if (n == 0) return "^$";

        string ret = "^";
        for (int i = 0; i < n; i++) ret += "#" + s.substr(i, 1);

        ret += "#$";
        return ret;
    }

    string longestPalindrome(string s) {
        string T = preProcess(s);
        const 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[n];
        int C = 0, R = 0;

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

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

            // Attempt to expand palindrome centered at i
            while (T[i + 1 + P[i]] == T[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 max_len = 0;
        int center_index = 0;
        for (int i = 1; i < n - 1; i++) {
            if (P[i] > max_len) {
                max_len = P[i];
                center_index = i;
```

```
            }
        }

        return s.substr((center_index - 1 - max_len) / 2, max_len);
    }
};
```

相关题目

- 无

3.6 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

```
// LintCode, Regular Expression Matching
// 递归版, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    bool isMatch(const char *s, const char *p) {
        if (*p == '\0') return *s == '\0';

        // next char is not '*', then must match current character
        if (*(p + 1) != '*') {
            if (*p == *s || (*p == '.' && *s != '\0'))
                return isMatch(s + 1, p + 1);
```

```
            else
                return false;
        } else { // next char is '*'
            while (*p == *s || (*p == '.' && *s != '\0')) {
                if (isMatch(s, p + 2))
                    return true;
                s++;
            }
            return isMatch(s, p + 2);
        }
    }
};
```

迭代版

相关题目

- Wildcard Matching, 见 §3.7

3.7 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

```
// LintCode, 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:
    bool isMatch(const char *s, const char *p) {
        if (*p == '*') {
            while (*p == '*') ++p;  //skip continuous '*'
            if (*p == '\0') return true;
            while (*s != '\0' && !isMatch(s, p)) ++s;

            return *s != '\0';
        }
        else if (*p == '\0' || *s == '\0') return *p == *s;
        else if (*p == *s || *p == '?') return isMatch(++s, ++p);
        else return false;
    }
};
```

迭代版

```
// LintCode, Wildcard Matching
// 迭代版, 时间复杂度 O(n*m), 空间复杂度 O(1)
class Solution {
public:
    bool isMatch(const char *s, const char *p) {
        bool star = false;
        const char *str, *ptr;
        for (str = s, ptr = p; *str != '\0'; str++, ptr++) {
            switch (*ptr) {
            case '?':
                break;
            case '*':
                star = true;
                s = str, p = ptr;
                while (*p == '*') p++;  //skip continuous '*'
                if (*p == '\0') return true;
                str = s - 1;
                ptr = p - 1;
                break;
            default:
                if (*str != *ptr) {
                    // 如果前面没有'*', 则匹配不成功
                    if (!star) return false;
                    s++;
                    str = s - 1;
                    ptr = p - 1;
                }
            }
        }
        while (*ptr == '*') ptr++;
```

```
        return (*ptr == '\0');
    }
};
```

相关题目

- Regular Expression Matching, 见 §3.6

3.8 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

```
// LintCode, 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+...)
// @author 周倩 (http://weibo.com/zhouditty)
class Solution {
public:
    string longestCommonPrefix(vector<string> &strs) {
        if (strs.empty()) return "";

        for (int idx = 0; idx < strs[0].size(); ++idx) { // 纵向扫描
            for (int i = 1; i < strs.size(); ++i) {
                if (strs[i][idx] != strs[0][idx]) return strs[0].substr(0,idx);
            }
        }
        return strs[0];
    }
};
```

横向扫描

```
// LintCode, 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(vector<string> &strs) {
        if (strs.empty()) return "";
```

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

        return strs[0].substr(0, right_most + 1);
    }
};
```

相关题目

- 无

3.9 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

```
// LintCode, Valid Number
// @author 龚陆安 (http://weibo.com/luangong)
// finite automata, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    bool isNumber(const char *s) {
        enum InputType {
            INVALID,    // 0
            SPACE,      // 1
            SIGN,       // 2
```

```
            DIGIT,      // 3
            DOT,        // 4
            EXPONENT,   // 5
            NUM_INPUTS  // 6
        };
        const int transitionTable[][NUM_INPUTS] = {
                -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 (; *s != '\0'; ++s) {
            InputType inputType = INVALID;
            if (isspace(*s))
                inputType = SPACE;
            else if (*s == '+' || *s == '-')
                inputType = SIGN;
            else if (isdigit(*s))
                inputType = DIGIT;
            else if (*s == '.')
                inputType = DOT;
            else if (*s == 'e' || *s == 'E')
                inputType = EXPONENT;

            // Get next state from current state and input symbol
            state = transitionTable[state][inputType];

            // 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;

    }
};
```

使用 strtod()

```
// LintCode, Valid Number
// @author 连城 (http://weibo.com/lianchengzju)
// 偷懒, 直接用 strtod(), 时间复杂度 O(n)
class Solution {
public:
    bool isNumber (char const* s) {
        char* endptr;
```

```
        strtod (s, &endptr);

        if (endptr == s) return false;

        for (; *endptr; ++endptr)
            if (!isspace (*endptr)) return false;

        return true;
    }
};
```

相关题目

- 无

3.10 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

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

        string roman;
        for (size_t i = 0; num > 0; ++i) {
            int count = num / radix[i];
            num %= radix[i];
            for (; count > 0; --count) roman += symbol[i];
        }
        return roman;
    }
};
```

相关题目

- Roman to Integer, 见 §3.11

3.11 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

```
// LintCode, Roman to Integer
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    inline int map(const 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;
        }
    }

    int romanToInt(string s) {
        int result = 0;
        for (size_t i = 0; i < s.size(); i++) {
            if (i > 0 && map(s[i]) > map(s[i - 1])) {
                result += (map(s[i]) - 2 * map(s[i - 1]));
            } else {
                result += map(s[i]);
            }
        }
        return result;
    }
};
```

相关题目

- Integer to Roman, 见 §3.10

3.12 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

```
// LintCode, 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)
            s = getNext(s);

        return s;
    }

    string getNext(const string &s) {
        stringstream ss;

        for (auto i = s.begin(); i != s.end(); ) {
            auto j = find_if(i, s.end(), bind1st(not_equal_to<char>(), *i));
            ss << distance(i, j) << *i;
            i = j;
        }

        return ss.str();
```

```
    }
};
```

相关题目

- 无

3.13 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

```
// LintCode, Anagrams
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    vector<string> anagrams(vector<string> &strs) {
        unordered_map<string, vector<string> > group;
        for (const auto &s : strs) {
            string key = s;
            sort(key.begin(), key.end());
            group[key].push_back(s);
        }

        vector<string> result;
        for (auto it = group.cbegin(); it != group.cend(); it++) {
            if (it->second.size() > 1)
                result.insert(result.end(), it->second.begin(), it->second.end());
        }
        return result;
    }
};
```

相关题目

- 无

3.14 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

```
// LintCode, Simplify Path
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    string simplifyPath(string const& path) {
        vector<string> dirs; // 当做栈

        for (auto i = path.begin(); i != path.end();) {
            ++i;

            auto j = find(i, path.end(), '/');
            auto dir = string(i, j);

            if (!dir.empty() && dir != ".") {// 当有连续 '///' 时, dir 为空
                if (dir == "..") {
                    if (!dirs.empty())
                        dirs.pop_back();
                } else
                    dirs.push_back(dir);
            }

            i = j;
        }

        stringstream out;
        if (dirs.empty()) {
            out << "/";
```

```
        } else {
            for (auto dir : dirs)
                out << '/' << dir;
        }

        return out.str();
    }
};
```

相关题目

- 无

3.15 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

细节实现题。

用 STL

```
// LintCode, Length of Last Word
// 偷懒, 用 STL
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int lengthOfLastWord(const char *s) {
        const string str(s);
        auto first = find_if(str.rbegin(), str.rend(), ::isalpha);
        auto last = find_if_not(first, str.rend(), ::isalpha);
        return distance(first, last);
    }
};
```

顺序扫描

```
// LintCode, Length of Last Word
// 顺序扫描, 记录每个 word 的长度
```

```
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int lengthOfLastWord(const char *s) {
        int len = 0;
        while (*s) {
            if (*s++ != ' ')
                ++len;
            else if (*s && *s != ' ')
                len = 0;
        }
        return len;
    }
};
```

相关题目

- 无

第 4 章
栈和队列

4.1 栈

4.1.1 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

```
// LintCode, Valid Parentheses
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    bool isValid (string const& s) {
        string left = "([{";
        string right = ")]}";
        stack<char> stk;

        for (auto c : s) {
            if (left.find(c) != string::npos) {
                stk.push (c);
            } else {
                if (stk.empty () || stk.top () != left[right.find (c)])
                    return false;
                else
                    stk.pop ();
            }
        }
```

```
        return stk.empty();
    }
};
```

相关题目

- Generate Parentheses, 见 §10.9
- Longest Valid Parentheses, 见 §4.1.2

4.1.2 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

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

        for (int i = 0; i < s.size(); ++i) {
            if (s[i] =='(') {
                lefts.push(i);
            } else {
                if (lefts.empty()) {
                    // no matching left
                    last = i;
                } else {
                    // find a matching pair
                    lefts.pop();
                    if (lefts.empty()) {
                        max_len = max(max_len, i-last);
                    } else {
                        max_len = max(max_len, i-lefts.top());
```

```
                    }
                }
            }
        }
        return max_len;
    }
};
```

Dynamic Programming, One Pass

```
// LintCode, Longest Valid Parenthese
// 时间复杂度 O(n), 空间复杂度 O(n)
// @author 一只杰森 (http://weibo.com/wjson)
class Solution {
public:
    int longestValidParentheses(string s) {
        vector<int> f(s.size(), 0);
        int ret = 0;
        for (int i = s.size() - 2; i >= 0; --i) {
            int match = i + f[i + 1] + 1;
            // case: "((...))"
            if (s[i] == '(' && match < s.size() && s[match] == ')') {
                f[i] = f[i + 1] + 2;
                // if a valid sequence exist afterwards "((...))()"
                if (match + 1 < s.size()) f[i] += f[match + 1];
            }
            ret = max(ret, f[i]);
        }
        return ret;
    }
};
```

两遍扫描

```
// LintCode, 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(string s) {
        int answer = 0, depth = 0, start = -1;
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    answer = max(answer, i - start);
                }
```

```
            }
        }

        depth = 0;
        start = s.size();
        for (int i = s.size() - 1; i >= 0; --i) {
            if (s[i] == ')') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    answer = max(answer, start - i);
                }
            }
        }
        return answer;
    }
};
```

相关题目

- Valid Parentheses, 见 §4.1.1
- Generate Parentheses, 见 §10.9

4.1.3 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.

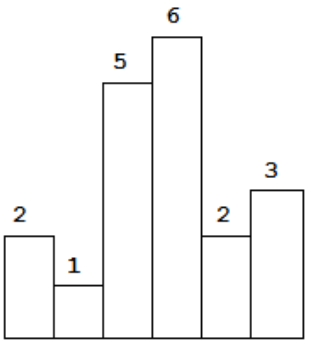


图 4-1 Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.

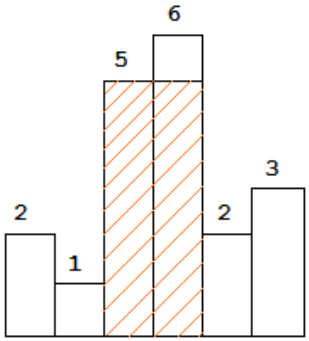


图 4-2 The largest rectangle is shown in the shaded area, which has area = 10 unit.

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

分析

简单的，类似于 Container With Most Water(§12.6)，对每个柱子，左右扩展，直到碰到比自己矮的，计算这个矩形的面积，用一个变量记录最大的面积，复杂度 $O(n^2)$，会超时。

如图 §4-2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

```
// LintCode, Largest Rectangle in Histogram
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    int largestRectangleArea(vector<int> &height) {
        stack<int> s;
        height.push_back(0);
        int result = 0;
        for (int i = 0; i < height.size(); ) {
            if (s.empty() || height[i] > height[s.top()])
                s.push(i++);
            else {
                int tmp = s.top();
                s.pop();
                result = max(result,
                        height[tmp] * (s.empty() ? i : i - s.top() - 1));
            }
        }
```

```
        return result;
    }
};
```

相关题目

- Trapping Rain Water, 见 §2.1.15
- Container With Most Water, 见 §12.6

4.1.4 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

无

递归版

```
// LintCode, Evaluate Reverse Polish Notation
// 递归, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    int evalRPN(vector<string> &tokens) {
        int x, y;
        auto token = tokens.back();  tokens.pop_back();
        if (is_operator(token))  {
            y = evalRPN(tokens);
            x = evalRPN(tokens);
            if (token[0] == '+')       x += y;
            else if (token[0] == '-')  x -= y;
            else if (token[0] == '*')  x *= y;
            else                       x /= y;
        } else  {
            size_t i;
            x = stoi(token, &i);
        }
        return x;
    }
private:
    bool is_operator(const string &op) {
        return op.size() == 1 && string("+-*/").find(op) != string::npos;
```

```
    }
};
```

迭代版

```
// LintCode, 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(vector<string> &tokens) {
        stack<string> s;
        for (auto token : tokens) {
            if (!is_operator(token)) {
                s.push(token);
            } else {
                int y = stoi(s.top());
                s.pop();
                int x = stoi(s.top());
                s.pop();
                if (token[0] == '+')       x += y;
                else if (token[0] == '-')  x -= y;
                else if (token[0] == '*')  x *= y;
                else                       x /= y;
                s.push(to_string(x));
            }
        }
        return stoi(s.top());
    }
private:
    bool is_operator(const string &op) {
        return op.size() == 1 && string("+-*/").find(op) != string::npos;
    }
};
```

相关题目

- 无

4.2 队列

第 5 章
树

LintCode 上二叉树的节点定义如下：

```
// 树的节点
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) { }
};
```

5.1 二叉树的遍历

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

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

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

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

5.1.1 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

```
// LintCode, Binary Tree Preorder Traversal
// 使用栈, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        const TreeNode *p;
        stack<const TreeNode *> s;

        p = root;
        if (p != nullptr) s.push(p);

        while (!s.empty()) {
            p = s.top();
            s.pop();
            result.push_back(p->val);

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

Morris 先序遍历

```
// LintCode, 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:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode *cur, *prev;

        cur = root;
        while (cur != nullptr) {
            if (cur->left == nullptr) {
                result.push_back(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 != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) { /* 还没线索化，则建立线索 */
                    result.push_back(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 = nullptr;
                    /* 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，见 §5.1.2
- Binary Tree Postorder Traversal，见 §5.1.3
- Recover Binary Search Tree，见 §5.1.7

5.1.2 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

```
// LintCode, Binary Tree Inorder Traversal
// 使用栈, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        const TreeNode *p = root;
        stack<const TreeNode *> s;

        while (!s.empty() || p != nullptr) {
            if (p != nullptr) {
                s.push(p);
                p = p->left;
            } else {
                p = s.top();
                s.pop();
                result.push_back(p->val);
                p = p->right;
            }
        }
        return result;
    }
};
```

Morris 中序遍历

```
// LintCode, 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:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode *cur, *prev;

        cur = root;
        while (cur != nullptr) {
            if (cur->left == nullptr) {
                result.push_back(cur->val);
                prev = cur;
                cur = cur->right;
            } else {
                /* 查找前驱 */
                TreeNode *node = cur->left;
                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) { /* 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.push_back(cur->val);
                    node->right = nullptr;
                    prev = cur;
                    cur = cur->right;
                }
            }
        }
        return result;
    }
};
```

相关题目

- Binary Tree Preorder Traversal，见 §5.1.1
- Binary Tree Postorder Traversal，见 §5.1.3
- Recover Binary Search Tree，见 §5.1.7

5.1.3 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

```
// LintCode, Binary Tree Postorder Traversal
// 使用栈，时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        /* p，正在访问的结点，q，刚刚访问过的结点 */
        const TreeNode *p, *q;
        stack<const TreeNode *> s;
```

```
        p = root;

        do {
            while (p != nullptr) { /* 往左下走 */
                s.push(p);
                p = p->left;
            }
            q = nullptr;
            while (!s.empty()) {
                p = s.top();
                s.pop();
                /* 右孩子不存在或已被访问, 访问之 */
                if (p->right == q) {
                    result.push_back(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

```
// LintCode, 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:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode dummy(-1);
        TreeNode *cur, *prev = nullptr;
        std::function < void(const TreeNode*)> visit =
            [&result](const TreeNode *node){
            result.push_back(node->val);
        };

        dummy.left = root;
        cur = &dummy;
        while (cur != nullptr) {
            if (cur->left == nullptr) {
                prev = cur; /* 必须要有 */
                cur = cur->right;
            } else {
                TreeNode *node = cur->left;
```

```
                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) { /* 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, visit);
                    prev->right = nullptr;
                    prev = cur; /* 必须要有 */
                    cur = cur->right;
                }
            }
        }
        return result;
    }
private:
    // 逆转路径
    static void reverse(TreeNode *from, TreeNode *to) {
        TreeNode *x = from, *y = from->right, *z;
        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
    static void visit_reverse(TreeNode* from, TreeNode *to,
                    std::function< void(const TreeNode*) >& visit) {
        TreeNode *p = to;
        reverse(from, to);

        while (true) {
            visit(p);
            if (p == from)
                break;
            p = p->right;
        }

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

相关题目

- Binary Tree Preorder Traversal，见 §5.1.1
- Binary Tree Inorder Traversal，见 §5.1.2

- Recover Binary Search Tree，见 §5.1.7

5.1.4 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

```
// LintCode, Binary Tree Level Order Traversal
// 递归版, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int>> result;
        traverse(root, 1, result);
        return result;
    }

    void traverse(TreeNode *root, size_t level, vector<vector<int>> &result) {
        if (!root) return;

        if (level > result.size())
            result.push_back(vector<int>());

        result[level-1].push_back(root->val);
        traverse(root->left, level+1, result);
        traverse(root->right, level+1, result);
    }
};
```

迭代版

```
// LintCode, Binary Tree Level Order Traversal
// 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int> > result;
        if(root == nullptr) return result;

        queue<TreeNode*> current, next;
        vector<int> level; // elments in level level

        current.push(root);
        while (!current.empty()) {
            while (!current.empty()) {
                TreeNode* node = current.front();
                current.pop();
                level.push_back(node->val);
                if (node->left != nullptr) next.push(node->left);
                if (node->right != nullptr) next.push(node->right);
            }
            result.push_back(level);
            level.clear();
            swap(next, current);
        }
        return result;
    }
};
```

相关题目

- Binary Tree Level Order Traversal II，见 §5.1.5
- Binary Tree Zigzag Level Order Traversal，见 §5.1.6

5.1.5 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（见§5.1.4）的基础上，`reverse()`一下即可。

递归版

```
// LintCode, Binary Tree Level Order Traversal II
// 递归版, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > levelOrderBottom(TreeNode *root) {
        vector<vector<int>> result;
        traverse(root, 1, result);
        std::reverse(result.begin(), result.end()); // 比上一题多此一行
        return result;
    }

    void traverse(TreeNode *root, size_t level, vector<vector<int>> &result) {
        if (!root) return;

        if (level > result.size())
            result.push_back(vector<int>());

        result[level-1].push_back(root->val);
        traverse(root->left, level+1, result);
        traverse(root->right, level+1, result);
    }
};
```

迭代版

```
// LintCode, Binary Tree Level Order Traversal II
// 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > levelOrderBottom(TreeNode *root) {
        vector<vector<int> > result;
        if(root == nullptr) return result;

        queue<TreeNode*> current, next;
        vector<int> level; // elments in level level

        current.push(root);
        while (!current.empty()) {
            while (!current.empty()) {
```

```
                TreeNode* node = current.front();
                current.pop();
                level.push_back(node->val);
                if (node->left != nullptr) next.push(node->left);
                if (node->right != nullptr) next.push(node->right);
            }
            result.push_back(level);
            level.clear();
            swap(next, current);
        }
        reverse(result.begin(), result.end()); // 比上一题多此一行
        return result;
    }
};
```

相关题目

- Binary Tree Level Order Traversal，见 §5.1.4
- Binary Tree Zigzag Level Order Traversal，见 §5.1.6

5.1.6 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

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

    void traverse(TreeNode *root, size_t level, vector<vector<int>> &result,
            bool left_to_right) {
        if (!root) return;

        if (level > result.size())
            result.push_back(vector<int>());

        if (left_to_right)
            result[level-1].push_back(root->val);
        else
            result[level-1].insert(result[level-1].begin(), root->val);

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

迭代版

```
//LintCode, 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)
class Solution {
public:
    vector<vector<int> > zigzagLevelOrder(TreeNode *root) {
        vector<vector<int> > result;
        if (nullptr == root) return result;

        queue<TreeNode*> q;
        bool left_to_right = true;  //left to right
        vector<int> level;  // one level's elements

        q.push(root);
        q.push(nullptr);  // level separator
        while (!q.empty()) {
            TreeNode *cur = q.front();
            q.pop();
            if (cur) {
                level.push_back(cur->val);
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
```

```
            } else {
                if (left_to_right) {
                    result.push_back(level);
                } else {
                    reverse(level.begin(), level.end());
                    result.push_back(level);
                }
                level.clear();
                left_to_right = !left_to_right;

                if (q.size() > 0) q.push(nullptr);
            }
        }

        return result;
    }
};
```

相关题目

- Binary Tree Level Order Traversal，见 §5.1.4
- Binary Tree Level Order Traversal II，见 §5.1.5

5.1.7 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

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

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

代码

```
// LintCode, Recover Binary Search Tree
// Morris 中序遍历, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    void recoverTree(TreeNode* root) {
        pair<TreeNode*, TreeNode*> broken;
        TreeNode* prev = nullptr;
```

```
        TreeNode* cur = root;

        while (cur != nullptr) {
            if (cur->left == nullptr) {
                detect(broken, prev, cur);
                prev = cur;
                cur = cur->right;
            } else {
                auto node = cur->left;

                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) {
                    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 = nullptr;
                    prev = cur;
                    cur = cur->right;
                }
            }
        }

        swap(broken.first->val, broken.second->val);
    }

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

相关题目

- Binary Tree Inorder Traversal，见 §5.1.2

5.1.8 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

递归版

```
// LintCode, Same Tree
// 递归版, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        if (!p && !q) return true;   // 终止条件
        if (!p || !q) 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

```
// LintCode, Same Tree
// 迭代版, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        stack<TreeNode*> s;
        s.push(p);
        s.push(q);

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

            if (!p && !q) continue;
            if (!p || !q) 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，见 §5.1.9

5.1.9 Symmetric 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

```
// LintCode, Symmetric Tree
// 递归版, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    bool isSymmetric(TreeNode *root) {
        return root ? isSymmetric(root->left, root->right) : true;
    }
    bool isSymmetric(TreeNode *left, TreeNode *right) {
        if (!left && !right) return true;   // 终止条件
        if (!left || !right) return false;  // 终止条件
        return left->val == right->val      // 三方合并
                && isSymmetric(left->left, right->right)
                && isSymmetric(left->right, right->left);
    }
};
```

迭代版

```
// LintCode, Symmetric Tree
// 迭代版, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    bool isSymmetric (TreeNode* root) {
        if (!root) return true;

        stack<TreeNode*> s;
        s.push(root->left);
        s.push(root->right);

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

```
            if (!p && !q) continue;
            if (!p || !q) 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，见 §5.1.8

5.1.10 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

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

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

```
        int left = balancedHeight (root->left);
        int right = balancedHeight (root->right);

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

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

相关题目

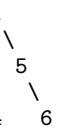
- 无

5.1.11 Flatten Binary Tree to Linked List

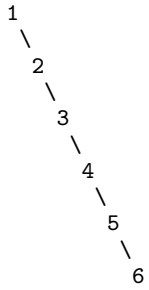
描述

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

For example, Given



The flattened tree should look like:



分析

无

递归版 1

```
// LintCode, Flatten Binary Tree to Linked List
// 递归版 1, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
```

```
public:
    void flatten(TreeNode *root) {
        if (root == nullptr) return;  // 终止条件

        flatten(root->left);
        flatten(root->right);

        if (nullptr == root->left) 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) p = p->right; //寻找左链表最后一个节点
        p->right = root->right;
        root->right = root->left;
        root->left = nullptr;
    }
};
```

递归版 2

```
// LintCode, 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)
class Solution {
public:
    void flatten(TreeNode *root) {
        flatten(root, NULL);
    }
private:
    // 把 root 所代表树变成链表后，tail 跟在该链表后面
    TreeNode *flatten(TreeNode *root, TreeNode *tail) {
        if (NULL == root) return tail;

        root->right = flatten(root->left, flatten(root->right, tail));
        root->left = NULL;
        return root;
    }
};
```

迭代版

```
// LintCode, Flatten Binary Tree to Linked List
// 迭代版，时间复杂度 O(n)，空间复杂度 O(logn)
class Solution {
public:
    void flatten(TreeNode* root) {
        if (root == nullptr) return;

        stack<TreeNode*> s;
        s.push(root);
```

```
        while (!s.empty()) {
            auto p = s.top();
            s.pop();

            if (p->right)
                s.push(p->right);
            if (p->left)
                s.push(p->left);

            p->left = nullptr;
            if (!s.empty())
                p->right = s.top();
        }
    }
};
```

相关题目

- 无

5.1.12 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

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

        TreeLinkNode dummy(-1);
        for (TreeLinkNode *curr = root, *prev = &dummy;
                curr; curr = curr->next) {
            if (curr->left != nullptr){
                prev->next = curr->left;
                prev = prev->next;
            }
            if (curr->right != nullptr){
                prev->next = curr->right;
                prev = prev->next;
            }
        }
        connect(dummy.next);
    }
};
```

迭代版

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

                if (root->left) {
                    if (prev) prev->next = root->left;
                    prev = root->left;
                }
                if (root->right) {
                    if (prev) prev->next = root->right;
                    prev = root->right;
                }
            }
            root = next; // turn to next level
        }
    }
};
```

相关题目

- Populating Next Right Pointers in Each Node，见 §5.4.6

5.2 二叉树的构建

5.2.1 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

```
// LintCode, Construct Binary Tree from Preorder and Inorder Traversal
// 递归，时间复杂度 O(n)，空间复杂度 O(\logn)
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        return buildTree(begin(preorder), end(preorder),
                begin(inorder), end(inorder));
    }

    template<typename InputIterator>
    TreeNode* buildTree(InputIterator pre_first, InputIterator pre_last,
            InputIterator in_first, InputIterator in_last) {
        if (pre_first == pre_last) return nullptr;
        if (in_first == in_last) return nullptr;

        auto root = new TreeNode(*pre_first);

        auto inRootPos = find(in_first, in_last, *pre_first);
        auto leftSize = distance(in_first, inRootPos);

        root->left = buildTree(next(pre_first), next(pre_first,
                leftSize + 1), in_first, next(in_first, leftSize));
        root->right = buildTree(next(pre_first, leftSize + 1), pre_last,
                next(inRootPos), in_last);

        return root;
    }
};
```

相关题目

- Construct Binary Tree from Inorder and Postorder Traversal，见 §5.2.2

5.2.2 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

```
// LintCode, Construct Binary Tree from Inorder and Postorder Traversal
// 递归, 时间复杂度 O(n), 空间复杂度 O(\logn)
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        return buildTree(begin(inorder), end(inorder),
                begin(postorder), end(postorder));
    }

    template<typename BidiIt>
    TreeNode* buildTree(BidiIt in_first, BidiIt in_last,
            BidiIt post_first, BidiIt post_last) {
        if (in_first ==in_last) return nullptr;
        if (post_first == post_last) return nullptr;

        const auto val = *prev(post_last);
        TreeNode* root = new TreeNode(val);

        auto in_root_pos = find(in_first, in_last, val);
        auto left_size = distance(in_first, in_root_pos);
        auto post_left_last = next(post_first, left_size);

        root->left = buildTree(in_first, in_root_pos, post_first, post_left_last);
        root->right = buildTree(next(in_root_pos), in_last, post_left_last,
                prev(post_last));

        return root;
    }
};
```

相关题目

- Construct Binary Tree from Preorder and Inorder Traversal，见 §5.2.1

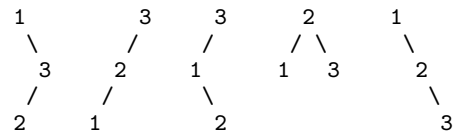
5.3 二叉查找树

5.3.1 Unique Binary Search Trees

描述

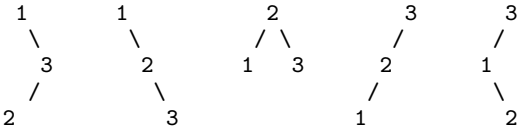
Given n, how many structurally unique BST's (binary search trees) that store values $1...n$?

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



分析

如果把上例的顺序改一下，就可以看出规律了。



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：**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

$$
\begin{aligned}
f(2) &= f(0) * f(1) \text{，1 为根的情况} \\
&+ f(1) * f(0) \text{，2 为根的情况}
\end{aligned}
$$

再看一看 3 个元素的数组，可以发现 BST 的取值方式如下：

$$\begin{aligned} f(3) &= f(0) * f(2) \text{，1 为根的情况} \\ &+ f(1) * f(1) \text{，2 为根的情况} \\ &+ f(2) * f(0) \text{，3 为根的情况} \end{aligned}$$

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

```
// LintCode, Unique Binary Search Trees
// 时间复杂度 O(n^2), 空间复杂度 O(n)
class Solution {
public:
    int numTrees(int n) {
        vector<int> f(n + 1, 0);

        f[0] = 1;
        f[1] = 1;
        for (int i = 2; i <= n; ++i) {
            for (int k = 1; k <= i; ++k)
                f[i] += f[k-1] * f[i - k];
        }

        return f[n];
    }
};
```

相关题目

- Unique Binary Search Trees II，见 §5.3.2

5.3.2 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.

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

分析

见前面一题。

代码

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

相关题目

- Unique Binary Search Trees，见 §5.3.1

5.3.3 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

代码

```
// LintCode, Validate Binary Search Tree
// 时间复杂度 O(n), 空间复杂度 O(\logn)
class Solution {
public:
    bool 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，见 §5.3.3

5.3.4 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

```
// LintCode, Convert Sorted Array to Binary Search Tree
// 分治法, 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    TreeNode* sortedArrayToBST (vector<int>& num) {
        return sortedArrayToBST(num.begin(), num.end());
    }

    template<typename RandomAccessIterator>
    TreeNode* sortedArrayToBST (RandomAccessIterator first,
```

```
            RandomAccessIterator last) {
        const auto length = distance(first, last);

        if (length <= 0) return nullptr;  // 终止条件

        // 三方合并
        auto mid = first + length / 2;
        TreeNode* root = new TreeNode (*mid);
        root->left = sortedArrayToBST(first, mid);
        root->right = sortedArrayToBST(mid + 1, last);

        return root;
    }
};
```

相关题目

- Convert Sorted List to Binary Search Tree，见 §5.3.5

5.3.5 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://lintcode.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 $O(n \log n)$。

```
// LintCode, Convert Sorted List to Binary Search Tree
// 分治法, 类似于 Convert Sorted Array to Binary Search Tree,
// 自顶向下, 时间复杂度 O(n^2), 空间复杂度 O(logn)
class Solution {
public:
    TreeNode* sortedListToBST (ListNode* head) {
        return sortedListToBST (head, listLength (head));
    }

    TreeNode* sortedListToBST (ListNode* head, int len) {
        if (len == 0) return nullptr;
```

```
        if (len == 1) return new TreeNode (head->val);

        TreeNode* root = new TreeNode (nth_node (head, len / 2 + 1)->val);
        root->left = sortedListToBST (head, len / 2);
        root->right = sortedListToBST (nth_node (head, len / 2 + 2),
                (len - 1) / 2);

        return root;
    }

    int listLength (ListNode* node) {
        int n = 0;

        while(node) {
            ++n;
            node = node->next;
        }

        return n;
    }

    ListNode* nth_node (ListNode* node, int n) {
        while (--n)
            node = node->next;

        return node;
    }
};
```

自底向上

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

        int mid = start + (end - start) / 2;
        TreeNode *leftChild = sortedListToBST(list, start, mid - 1);
        TreeNode *parent = new TreeNode(list->val);
        parent->left = leftChild;
        list = list->next;
```

```
        parent->right = sortedListToBST(list, mid + 1, end);
        return parent;
    }
};
```

相关题目

- Convert Sorted Array to Binary Search Tree，见 §5.3.4

5.4 二叉树的递归

二叉树是一个递归的数据结构，因此是一个用来考察递归思维能力的绝佳数据结构。

递归一定是深搜（见 §10.12.5节“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`。

5.4.1 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

```
// LintCode, Minimum Depth of Binary Tree
// 递归版，时间复杂度 O(n)，空间复杂度 O(logn)
class Solution {
public:
    int minDepth(const TreeNode *root) {
        return minDepth(root, false);
    }
private:
    static int minDepth(const TreeNode *root, bool hasbrother) {
        if (!root) return hasbrother ? INT_MAX : 0;

        return 1 + min(minDepth(root->left, root->right != NULL),
                minDepth(root->right, root->left != NULL));
```

```
    }
};
```

迭代版

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

        int result = INT_MAX;

        stack<pair<TreeNode*, int>> s;
        s.push(make_pair(root, 1));

        while (!s.empty()) {
            auto node = s.top().first;
            auto depth = s.top().second;
            s.pop();

            if (node->left == nullptr && node->right == nullptr)
                result = min(result, depth);

            if (node->left && result > depth) // 深度控制, 剪枝
                s.push(make_pair(node->left, depth + 1));

            if (node->right && result > depth) // 深度控制, 剪枝
                s.push(make_pair(node->right, depth + 1));
        }

        return result;
    }
};
```

相关题目

- Maximum Depth of Binary Tree，见 §5.4.2

5.4.2 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

```
// LintCode, 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 == nullptr) return 0;

        return max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};
```

相关题目

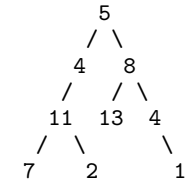
- Minimum Depth of Binary Tree，见 §5.4.1

5.4.3 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`,



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

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

        if (root->left == nullptr && root->right == nullptr) // 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，见 §5.4.4

5.4.4 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

```
// LintCode, Path Sum II
// 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    vector<vector<int> > pathSum(TreeNode *root, int sum) {
        vector<vector<int> > result;
        vector<int> cur; // 中间结果
        pathSum(root, sum, cur, result);
        return result;
    }
private:
    void pathSum(TreeNode *root, int gap, vector<int> &cur,
            vector<vector<int> > &result) {
        if (root == nullptr) return;

        cur.push_back(root->val);

        if (root->left == nullptr && root->right == nullptr) { // leaf
            if (gap == root->val)
                result.push_back(cur);
        }
        pathSum(root->left, gap - root->val, cur, result);
        pathSum(root->right, gap - root->val, cur, result);

        cur.pop_back();
    }
};
```

相关题目

- Path Sum，见 §5.4.3

5.4.5 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，见第 §13.2节。如果说 Array 只有一个方向的话，那么 Binary Tree 其实只是左、右两个方向而已，我们需要比较两个方向上的值。

不过，Array 可以从头到尾遍历，那么 Binary Tree 怎么办呢，我们可以采用 Binary Tree 最常用的 dfs 来进行遍历。先算出左右子树的结果 L 和 R，如果 L 大于 0，那么对后续结果是有利的，我们加上 L，如果 R 大于 0，对后续结果也是有利的，继续加上 R。

代码

```
// LintCode, Binary Tree Maximum Path Sum
// 时间复杂度 O(n)，空间复杂度 O(logn)
class Solution {
public:
    int maxPathSum(TreeNode *root) {
        max_sum = INT_MIN;
        dfs(root);
        return max_sum;
    }
private:
    int max_sum;
    int dfs(const TreeNode *root) {
        if (root == nullptr) 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 = max(max_sum, sum);
        return max(r, l) > 0 ? 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，见 §13.2

5.4.6 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, Given the following perfect binary tree,

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

After calling your function, the tree should look like:

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

分析

无

代码

```
// LintCode, Populating Next Right Pointers in Each Node
// 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    void connect(TreeLinkNode *root) {
        connect(root, NULL);
    }
private:
    void connect(TreeLinkNode *root, TreeLinkNode *sibling) {
        if (root == nullptr)
            return;
        else
            root->next = sibling;

        connect(root->left, root->right);
        if (sibling)
            connect(root->right, sibling->left);
        else
            connect(root->right, nullptr);
    }
};
```

相关题目

- Populating Next Right Pointers in Each Node II，见 §5.1.12

5.4.7 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

```
// LintCode, Decode Ways
// 时间复杂度 O(n), 空间复杂度 O(logn)
class Solution {
public:
    int sumNumbers(TreeNode *root) {
        return dfs(root, 0);
    }
private:
    int dfs(TreeNode *root, int sum) {
        if (root == nullptr) return 0;
        if (root->left == nullptr && root->right == nullptr)
            return sum * 10 + root->val;

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

相关题目

- 无

第 6 章
排序

6.1 Merge Sorted Array

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

```
//LintCode, Merge Sorted Array
// 时间复杂度 O(m+n), 空间复杂度 O(1)
class Solution {
public:
    vector<int> mergeSortedArray(const vector<int> &A, const vector<int> &B) {
        const int m = A.size();
        const int n = B.size();
        vector<int> result(m + n, 0);
        int ia = m - 1, ib = n - 1, icur = m + n - 1;
        while(ia >= 0 && ib >= 0) {
            result[icur--] = A[ia] >= B[ib] ? A[ia--] : B[ib--];
        }

        while(ia >= 0) {
            result[icur--] = A[ia--];
        }
        while(ib >= 0) {
            result[icur--] = B[ib--];
        }

        return result;
    }
};
```

相关题目

- Merge Two Sorted Lists，见 §6.2
- Merge k Sorted Lists，见 §6.3

6.2 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

```
//LintCode, Merge Two Sorted Lists
// 时间复杂度 O(min(m,n)), 空间复杂度 O(1)
class Solution {
public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        if (l1 == nullptr) return l2;
        if (l2 == nullptr) return l1;
        ListNode dummy(-1);
        ListNode *p = &dummy;
        for (; l1 != nullptr && l2 != nullptr; p = p->next) {
            if (l1->val > l2->val) { p->next = l2; l2 = l2->next; }
            else { p->next = l1; l1 = l1->next; }
        }
        p->next = l1 != nullptr ? l1 : l2;
        return dummy.next;
    }
};
```

相关题目

- Merge Sorted Array §6.1
- Merge k Sorted Lists，见 §6.3

6.3 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（见 §6.2）的函数

代码

```
//LintCode, Merge k Sorted Lists
// 时间复杂度 O(n1+n2+...), 空间复杂度 O(1)
class Solution {
public:
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if (lists.size() == 0) return nullptr;

        ListNode *p = lists[0];
        for (int i = 1; i < lists.size(); i++) {
            p = mergeTwoLists(p, lists[i]);
        }
        return p;
    }

    // Merge Two Sorted Lists
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode head(-1);
        for (ListNode* p = &head; l1 != nullptr || l2 != nullptr; p = p->next) {
            int val1 = l1 == nullptr ? INT_MAX : l1->val;
            int val2 = l2 == nullptr ? INT_MAX : l2->val;
            if (val1 <= val2) {
                p->next = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                l2 = l2->next;
            }
        }
        return head.next;
    }
};
```

相关题目

- Merge Sorted Array §6.1
- Merge Two Sorted Lists，见 §6.2

6.4 Insertion Sort List

描述

Sort a linked list using insertion sort.

分析

无

代码

```
// LintCode, Insertion Sort List
// 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    ListNode *insertionSortList(ListNode *head) {
        ListNode dummy(INT_MIN);
        //dummy.next = head;

        for (ListNode *cur = head; cur != nullptr;) {
            auto 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 = nullptr;
        for (ListNode *cur = head; cur != nullptr && cur->val <= x;
            pre = cur, cur = cur->next)
            ;
        return pre;
    }
};
```

相关题目

- Sort List, 见 §6.5

6.5 Sort List

描述

Sort a linked list in $O(nlogn)$ 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

```
// LintCode, Sort List
// 归并排序, 时间复杂度 O(nlogn), 空间复杂度 O(1)
class Solution {
public:
    ListNode *sortList(ListNode *head) {
        if (head == NULL || head->next == NULL)return head;

        // 快慢指针找到中间节点
        ListNode *fast = head, *slow = head;
        while (fast->next != NULL && fast->next->next != NULL) {
            fast = fast->next->next;
            slow = slow->next;
        }
        // 断开
        fast = slow;
        slow = slow->next;
        fast->next = NULL;

        ListNode *l1 = sortList(head);  // 前半段排序
        ListNode *l2 = sortList(slow);  // 后半段排序
        return mergeTwoLists(l1, l2);
    }

    // Merge Two Sorted Lists
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode dummy(-1);
        for (ListNode* p = &dummy; l1 != nullptr || l2 != nullptr; p = p->next) {
            int val1 = l1 == nullptr ? INT_MAX : l1->val;
            int val2 = l2 == nullptr ? INT_MAX : l2->val;
            if (val1 <= val2) {
                p->next = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                l2 = l2->next;
            }
        }
        return dummy.next;
    }
};
```

相关题目

- Insertion Sort List，见 §6.4

6.6 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

```
// LintCode, First Missing Positive
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int firstMissingPositive(vector<int> &A) {
        bucket_sort(A);

        const int n = A.size();
        for (int i = 0; i < n; ++i)
            if (A[i] != (i + 1))
                return i + 1;
        return n + 1;
    }
private:
    static void bucket_sort(vector<int> &A) {
        const int n = A.size();
        for (int i = 0; i < n; i++) {
            while (A[i] != i + 1) {
                if (A[i] <= 0 || A[i] > n || A[i] == A[A[i] - 1])
                    break;
                swap(A[i], A[A[i] - 1]);
            }
        }
    }
};
```

相关题目

- Sort Colors, 见 §6.7

6.7 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

```
// LintCode, Sort Colors
// Counting Sort
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    void sortColors(int A[], int n) {
        int counts[3] = { 0 }; // 记录每个颜色出现的次数

        for (int i = 0; i < n; i++)
            counts[A[i]]++;

        for (int i = 0, index = 0; i < 3; i++)
            for (int j = 0; j < counts[i]; j++)
                A[index++] = i;

    }
};
```

代码 2

```
// LintCode, Sort Colors
// 双指针, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    void sortColors(int A[], int n) {
        // 一个是 red 的 index, 一个是 blue 的 index, 两边往中间走
        int red = 0, blue = n - 1;

        for (int i = 0; i < blue + 1;) {
            if (A[i] == 0)
                swap(A[i++], A[red++]);
            else if (A[i] == 2)
                swap(A[i], A[blue--]);
            else
                i++;
        }
    }
};
```

代码 3

```
// LintCode, Sort Colors
// use partition()
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    void sortColors(int A[], int n) {
        partition(partition(A, A + n, bind1st(equal_to<int>(), 0)), A + n,
                bind1st(equal_to<int>(), 1));
    }
};
```

代码 4

```
// LintCode, Sort Colors
// 重新实现 partition()
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    void sortColors(int A[], int n) {
        partition(partition(A, A + n, bind1st(equal_to<int>(), 0)), A + n,
                bind1st(equal_to<int>(), 1));
    }
private:
    template<typename ForwardIterator, typename UnaryPredicate>
    ForwardIterator partition(ForwardIterator first, ForwardIterator last,
            UnaryPredicate pred) {
        auto pos = first;

        for (; first != last; ++first)
```

```
            if (pred(*first))
                swap(*first, *pos++);

        return pos;
    }
};
```

相关题目

- First Missing Positive, 见 §6.6

第 7 章
查找

7.1 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。

使用 STL

```
// LintCode, Search for a Range
// 偷懒的做法, 使用 STL
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    vector<int> searchRange(int A[], int n, int target) {
        const int l = distance(A, lower_bound(A, A + n, target));
        const int u = distance(A, prev(upper_bound(A, A + n, target)));
        if (A[l] != target) // not found
            return vector<int> { -1, -1 };
        else
            return vector<int> { l, u };
    }
};
```

重新实现 lower_bound 和 upper_bound

```
// LintCode, 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)
class Solution {
```

```
public:
    vector<int> searchRange (int A[], int n, int target) {
        auto lower = lower_bound(A, A + n, target);
        auto uppper = upper_bound(lower, A + n, target);

        if (lower == A + n || *lower != target)
            return vector<int> { -1, -1 };
        else
            return vector<int> {distance(A, lower), distance(A, prev(uppper))};
    }

    template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound (ForwardIterator first,
            ForwardIterator last, T value) {
        while (first != last) {
            auto mid = next(first, distance(first, last) / 2);

            if (value > *mid)   first = ++mid;
            else                last = mid;
        }

        return first;
    }

    template<typename ForwardIterator, typename T>
    ForwardIterator upper_bound (ForwardIterator first,
            ForwardIterator last, T value) {
        while (first != last) {
            auto mid = next(first, distance (first, last) / 2);

            if (value >= *mid)   first = ++mid;  // 与 lower_bound 仅此不同
            else                 last = mid;
        }

        return first;
    }
};
```

相关题目

- Search Insert Position, 见 §7.2

7.2 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

```
// LintCode, Search Insert Position
// 重新实现 lower_bound
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    int searchInsert(int A[], int n, int target) {
        return lower_bound(A, A + n, target) - A;
    }

    template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound (ForwardIterator first,
            ForwardIterator last, T value) {
        while (first != last) {
            auto mid = next(first, distance(first, last) / 2);

            if (value > *mid)   first = ++mid;
            else                last = mid;
        }

        return first;
    }
};
```

相关题目

- Search for a Range, 见 §7.1

7.3 Search a 2D Matrix

描述

Write an efficient algorithm that searches for a value in an $m \times n$ 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

```
// LintCode, Search a 2D Matrix
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    bool searchMatrix(const vector<vector<int>>& matrix, int target) {
        if (matrix.empty()) return false;
        const size_t  m = matrix.size();
        const size_t n = matrix.front().size();

        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

- 无

第 8 章
暴力枚举法

8.1 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],
  []
]
```

8.1.1 递归

增量构造法

每个元素，都有两种选择，选或者不选。

```
// LintCode, Subsets
// 增量构造法，深搜，时间复杂度 O(2^n)，空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end());  // 输出要求有序
        vector<vector<int> > result;
        vector<int> path;
        subsets(S, path, 0, result);
        return result;
```

```
    }

private:
    static void subsets(const vector<int> &S, vector<int> &path, int step,
            vector<vector<int> > &result) {
        if (step == S.size()) {
            result.push_back(path);
            return;
        }
        // 不选 S[step]
        subsets(S, path, step + 1, result);
        // 选 S[step]
        path.push_back(S[step]);
        subsets(S, path, step + 1, result);
        path.pop_back();
    }
};
```

位向量法

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

```
// LintCode, Subsets
// 位向量法, 深搜, 时间复杂度 O(2^n), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end());  // 输出要求有序

        vector<vector<int> > result;
        vector<bool> selected(S.size(), false);
        subsets(S, selected, 0, result);
        return result;
    }

private:
    static void subsets(const vector<int> &S, vector<bool> &selected, int step,
            vector<vector<int> > &result) {
        if (step == S.size()) {
            vector<int> subset;
            for (int i = 0; i < S.size(); i++) {
                if (selected[i]) subset.push_back(S[i]);
            }
            result.push_back(subset);
            return;
        }
        // 不选 S[step]
        selected[step] = false;
        subsets(S, selected, step + 1, result);
        // 选 S[step]
        selected[step] = true;
        subsets(S, selected, step + 1, result);
```

```
    }
};
```

8.1.2 迭代

增量构造法

```
// LintCode, Subsets
// 迭代版, 时间复杂度 O(2^n), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end()); // 输出要求有序
        vector<vector<int> > result(1);
        for (auto elem : S) {
            result.reserve(result.size() * 2);
            auto half = result.begin() + result.size();
            copy(result.begin(), half, back_inserter(result));
            for_each(half, result.end(), [&elem](decltype(result[0]) &e){
                e.push_back(elem);
            });
        }
        return result;
    }
};
```

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。

```
// LintCode, Subsets
// 二进制法, 时间复杂度 O(2^n), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end()); // 输出要求有序
        vector<vector<int> > result;
        const size_t n = S.size();
        vector<int> v;

        for (size_t i = 0; i < 1 << n; i++) {
            for (size_t j = 0; j < n; j++) {
                if (i & 1 << j) v.push_back(S[j]);
            }
            result.push_back(v);
            v.clear();
```

```
        }
        return result;
    }
};
```

相关题目

- Subsets II，见 §8.2

8.2 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。

8.2.1 递归

增量构造法

```
// LintCode, 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)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        sort(S.begin(), S.end());  // 必须排序

        vector<vector<int> > result;
        vector<int> path;

        dfs(S, S.begin(), path, result);
```

```
        return result;
    }

private:
    static void dfs(const vector<int> &S, vector<int>::iterator start,
            vector<int> &path, vector<vector<int> > &result) {
        result.push_back(path);

        for (auto i = start; i < S.end(); i++) {
            if (i != start && *i == *(i-1)) continue;
            path.push_back(*i);
            dfs(S, i + 1, path, result);
            path.pop_back();
        }
    }
};
// LintCode, 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)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > result;
        sort(S.begin(), S.end()); // 必须排序

        unordered_map<int, int> count_map; // 记录每个元素的出现次数
        for_each(S.begin(), S.end(), [&count_map](int e) {
            if (count_map.find(e) != count_map.end())
                count_map[e]++;
            else
                count_map[e] = 1;
        });

        // 将 map 里的 pair 拷贝到一个 vector 里
        vector<pair<int, int> > elems;
        for_each(count_map.begin(), count_map.end(),
                [&elems](const pair<int, int> &e) {
                    elems.push_back(e);
                });
        sort(elems.begin(), elems.end());
        vector<int> path; // 中间结果

        subsets(elems, 0, path, result);
        return result;
    }

private:
    static void subsets(const vector<pair<int, int> > &elems,
            size_t step, vector<int> &path, vector<vector<int> > &result) {
        if (step == elems.size()) {
            result.push_back(path);
            return;
        }
```

```
        for (int i = 0; i <= elems[step].second; i++) {
            for (int j = 0; j < i; ++j) {
                path.push_back(elems[step].first);
            }
            subsets(elems, step + 1, path, result);
            for (int j = 0; j < i; ++j) {
                path.pop_back();
            }
        }
    }
};
```

位向量法

```
// LintCode, Subsets II
// 位向量法, 时间复杂度 O(2^n), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > result; // 必须排序
        sort(S.begin(), S.end());
        vector<int> count(S.back() - S.front() + 1, 0);
        // 计算所有元素的个数
        for (auto i : S) {
            count[i - S[0]]++;
        }

        // 每个元素选择了多少个
        vector<int> selected(S.back() - S.front() + 1, -1);

        subsets(S, count, selected, 0, result);
        return result;
    }

private:
    static void subsets(const vector<int> &S, vector<int> &count,
            vector<int> &selected, size_t step, vector<vector<int> > &result) {
        if (step == count.size()) {
            vector<int> subset;
            for(size_t i = 0; i < selected.size(); i++) {
                for (int j = 0; j < selected[i]; j++) {
                    subset.push_back(i+S[0]);
                }
            }
            result.push_back(subset);
            return;
        }

        for (int i = 0; i <= count[step]; i++) {
            selected[step] = i;
            subsets(S, count, selected, step + 1, result);
        }
    }
```

```
};
```

8.2.2 迭代

增量构造法

```
// LintCode, Subsets II
// 增量构造法
// 时间复杂度 O(2^n), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        sort(S.begin(), S.end()); // 必须排序
        vector<vector<int> > result(1);

        size_t previous_size = 0;
        for (size_t i = 0; i < S.size(); ++i) {
            const size_t size = result.size();
            for (size_t j = 0; j < size; ++j) {
                if (i == 0 || S[i] != S[i-1] || j >= previous_size) {
                    result.push_back(result[j]);
                    result.back().push_back(S[i]);
                }
            }
            previous_size = size;
        }
        return result;
    }
};
```

二进制法

```
// LintCode, Subsets II
// 二进制法, 时间复杂度 O(2^n), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        sort(S.begin(), S.end()); // 必须排序
        // 用 set 去重, 不能用 unordered_set, 因为输出要求有序
        set<vector<int> > result;
        const size_t n = S.size();
        vector<int> v;

        for (size_t i = 0; i < 1U << n; ++i) {
            for (size_t j = 0; j < n; ++j) {
                if (i & 1 << j)
                    v.push_back(S[j]);
            }
            result.insert(v);
            v.clear();
        }
        vector<vector<int> > real_result;
```

```
        copy(result.begin(), result.end(), back_inserter(real_result));
        return real_result;
    }
};
```

相关题目

- Subsets，见 §8.1

8.3 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]`.

8.3.1 next_permutation()

偷懒的做法，可以直接使用 `std::next_permutation()`。如果是在 OJ 网站上，可以用这个 API 偷个懒；如果是在面试中，面试官肯定会让你重新实现。

解法 1

```
// LintCode, Permutations
// 时间复杂度 O(n!), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        vector<vector<int> > result;
        sort(num.begin(), num.end());

        do {
            result.push_back(num);
        } while(next_permutation(num.begin(), num.end()));
        return result;
    }
};
```

8.3.2 解法 2: 重新实现 next_permutation()

见第 §2.1.12 节。

解法 3

```
// LintCode, Permutations
// 重新实现 next_permutation()
// 时间复杂度 O(n!), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<int>> permute(vector<int>& num) {
        sort(num.begin(), num.end());

        vector<vector<int>> permutations;

        do {
            permutations.push_back(num);
        } while (next_permutation(num.begin(), num.end())); // 见第 2.1 节

        return permutations;
    }
};
```

8.3.3 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

```
// LintCode, Permutations
// 深搜, 增量构造法
// 时间复杂度 O(n!), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > permute(vector<int>& num) {
        sort(num.begin(), num.end());

        vector<vector<int>> result;
        vector<int> path;  // 中间结果

        dfs(num, path, result);
        return result;
    }
private:
    void dfs(const vector<int>& num, vector<int> &path,
            vector<vector<int> > &result) {
        if (path.size() == num.size()) {  // 收敛条件
            result.push_back(path);
            return;
        }
```

```
            // 扩展状态
            for (auto i : num) {
                // 查找 i 是否在 path 中出现过
                auto pos = find(path.begin(), path.end(), i);

                if (pos == path.end()) {
                    path.push_back(i);
                    dfs(num, path, result);
                    path.pop_back();
                }
            }
        }
    };
```

相关题目

- Next Permutation, 见 §2.1.12
- Permutation Sequence, 见 §2.1.13
- Permutations II, 见 §8.4
- Combinations, 见 §8.5

8.4 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]`.

8.4.1 next_permutation()

直接使用 `std::next_permutation()`，代码与上一题相同。

8.4.2 重新实现 next_permutation()

重新实现 `std::next_permutation()`，代码与上一题相同。

8.4.3 递归

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

扩展节点，每次从小到大，选一个没有被用光的元素，直到所有元素被用光。

本题不需要判重，因为状态装换图是一颗有层次的树。

代码

```
// LintCode, Permutations II
// 深搜, 时间复杂度 O(n!), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > permuteUnique(vector<int>& num) {
        sort(num.begin(), num.end());

        unordered_map<int, int> count_map; // 记录每个元素的出现次数
        for_each(num.begin(), num.end(), [&count_map](int e) {
            if (count_map.find(e) != count_map.end())
                count_map[e]++;
            else
                count_map[e] = 1;
        });

        // 将 map 里的 pair 拷贝到一个 vector 里
        vector<pair<int, int> > elems;
        for_each(count_map.begin(), count_map.end(),
                [&elems](const pair<int, int> &e) {
                    elems.push_back(e);
                });

        vector<vector<int>> result; // 最终结果
        vector<int> p;  // 中间结果

        n = num.size();
        permute(elems.begin(), elems.end(), p, result);
        return result;
    }

private:
    size_t n;
    typedef vector<pair<int, int> >::const_iterator Iter;

    void permute(Iter first, Iter last, vector<int> &p,
            vector<vector<int> > &result) {
        if (n == p.size()) {  // 收敛条件
            result.push_back(p);
        }

        // 扩展状态
        for (auto i = first; i != last; i++) {
            int count = 0; // 统计 *i 在 p 中出现过多少次
            for (auto j = p.begin(); j != p.end(); j++) {
                if (i->first == *j) {
                    count ++;
                }
            }
            if (count < i->second) {
                p.push_back(i->first);
                permute(first, last, p, result);
```

```
                p.pop_back(); // 撤销动作，返回上一层
            }
        }
    }
};
```

相关题目

- Next Permutation, 见 §2.1.12
- Permutation Sequence, 见 §2.1.13
- Permutations, 见 §8.3
- Combinations, 见 §8.5

8.5 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],
]
```

8.5.1 递归

```
// LintCode, Combinations
// 深搜，递归
// 时间复杂度 O(n!)，空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > combine(int n, int k) {
        vector<vector<int> > result;
        vector<int> path;
        dfs(n, k, 1, 0, path, result);
        return result;
    }
private:
    // start，开始的数，cur，已经选择的数目
    static void dfs(int n, int k, int start, int cur,
            vector<int> &path, vector<vector<int> > &result) {
        if (cur == k) {
            result.push_back(path);
```

```
        }
        for (int i = start; i <= n; ++i) {
            path.push_back(i);
            dfs(n, k, i + 1, cur + 1, path, result);
            path.pop_back();
        }
    }
};
```

8.5.2 迭代

```
// LintCode, Combinations
// use prev_permutation()
// 时间复杂度 O((n-k)!), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > combine(int n, int k) {
        vector<int> values(n);
        iota(values.begin(), values.end(), 1);

        vector<bool> select(n, false);
        fill_n(select.begin(), k, true);

        vector<vector<int> > result;
        do{
            vector<int> one(k);
            for (int i = 0, index = 0; i < n; ++i)
                if (select[i])
                    one[index++] = values[i];
            result.push_back(one);
        } while(prev_permutation(select.begin(), select.end()));
        return result;
    }
};
```

相关题目

- Next Permutation, 见 §2.1.12
- Permutation Sequence, 见 §2.1.13
- Permutations, 见 §8.3
- Permutations II, 见 §8.4

8.6 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.

图 8-1 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

8.6.1 递归

```
// LintCode, Letter Combinations of a Phone Number
// 时间复杂度 O(3^n), 空间复杂度 O(n)
class Solution {
public:
    const vector<string> keyboard { " ", "", "abc", "def", // '0','1','2',...
            "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    vector<string> letterCombinations (const string &digits) {
        vector<string> result;
        dfs(digits, 0, "", result);
        return result;
    }

    void dfs(const string &digits, size_t cur, string path,
            vector<string> &result) {
        if (cur == digits.size()) {
            result.push_back(path);
            return;
        }
        for (auto c : keyboard[digits[cur] - '0']) {
            dfs(digits, cur + 1, path + c, result);
        }
    }
};
```

8.6.2 迭代

```
// LintCode, Letter Combinations of a Phone Number
// 时间复杂度 O(3^n), 空间复杂度 O(1)
class Solution {
public:
    const vector<string> keyboard { " ", "", "abc", "def", // '0','1','2',...
            "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    vector<string> letterCombinations (const string &digits) {
        vector<string> result(1, "");
        for (auto d : digits) {
            const size_t n = result.size();
            const size_t m = keyboard[d - '0'].size();

            result.resize(n * m);
            for (size_t i = 0; i < m; ++i)
                copy(result.begin(), result.begin() + n, result.begin() + n * i);

            for (size_t i = 0; i < m; ++i) {
                auto begin = result.begin();
                for_each(begin + n * i, begin + n * (i+1), [&](string &s) {
                    s += keyboard[d - '0'][i];
                });
            }
        }
        return result;
    }
};
```

相关题目

- 无

第 9 章
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。

9.1 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

代码

```
//LintCode, Word Ladder
// 时间复杂度 O(n), 空间复杂度 O(n)
```

```
class Solution {
public:
    int ladderLength(const string& start, const string &end,
            const unordered_set<string> &dict) {
        queue<string> current, next;    // 当前层，下一层
        unordered_set<string> visited;  // 判重

        int level = 0;  // 层次
        bool found = false;

        auto state_is_target = [&](const string &s) {return s == end;};
        auto state_extend = [&](const string &s) {
            vector<string> result;

            for (size_t i = 0; i < s.size(); ++i) {
                string new_word(s);
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c == new_word[i]) continue;

                    swap(c, new_word[i]);

                    if ((dict.count(new_word) > 0 || new_word == end) &&
                            !visited.count(new_word)) {
                        result.push_back(new_word);
                        visited.insert(new_word);
                    }
                    swap(c, new_word[i]); // 恢复该单词
                }
            }

            return result;
        };

        current.push(start);
        while (!current.empty() && !found) {
            ++level;
            while (!current.empty() && !found) {
                const string str = current.front();
                current.pop();

                const auto& new_states = state_extend(str);
                for (const auto& state : new_states) {
                    next.push(state);
                    if (state_is_target(state)) {
                        found = true; //找到了
                        break;
                    }
                }
            }
            swap(next, current);
        }
        if (found) return level + 1;
        else return 0;
```

```
    }
};
```

相关题目

- Word Ladder II，见 §9.2

9.2 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。

这题跟普通的广搜有很大的不同，就是要输出所有路径，因此在记录前驱和判重地方与普通广搜略有不同。

代码

```
//LintCode, Word Ladder II
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<string> > findLadders(string start, string end,
            const unordered_set<string> &dict) {
```

```
        unordered_set<string> current, next;  // 当前层，下一层，用集合是为了去重
        unordered_set<string> visited; // 判重
        unordered_map<string, vector<string> > father; // 树

        bool found = false;

        auto state_is_target = [&](const string &s) {return s == end;};
        auto state_extend = [&](const string &s) {
            unordered_set<string> result;

            for (size_t i = 0; i < s.size(); ++i) {
                string new_word(s);
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c == new_word[i]) continue;

                    swap(c, new_word[i]);

                    if ((dict.count(new_word) > 0 || new_word == end) &&
                            !visited.count(new_word)) {
                        result.insert(new_word);
                    }
                    swap(c, new_word[i]); // 恢复该单词
                }
            }

            return result;
        };

        current.insert(start);
        while (!current.empty() && !found) {
            // 先将本层全部置为已访问，防止同层之间互相指向
            for (const auto& word : current)
                visited.insert(word);
            for (const auto& word : current) {
                const auto new_states = state_extend(word);
                for (const auto &state : new_states) {
                    if (state_is_target(state)) found = true;
                    next.insert(state);
                    father[state].push_back(word);
                    // visited.insert(state); // 移动到最上面了
                }
            }

            current.clear();
            swap(current, next);
        }
        vector<vector<string> > result;
        if (found) {
            vector<string> path;
            gen_path(father, path, start, end, result);
        }
        return result;
    }
```

```
private:
    void gen_path(unordered_map<string, vector<string> > &father,
            vector<string> &path, const string &start, const string &word,
            vector<vector<string> > &result) {
        path.push_back(word);
        if (word == start) {
            result.push_back(path);
            reverse(result.back().begin(), result.back().end());
        } else {
            for (const auto& f : father[word]) {
                gen_path(father, path, start, f, result);
            }
        }
        path.pop_back();
    }
};
```

相关题目

- Word Ladder，见 §9.1

9.3 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

```
// LintCode, Surrounded Regions
// BFS, 时间复杂度 O(n), 空间复杂度 O(n)
```

```
class Solution {
public:
    void solve(vector<vector<char>> &board) {
        if (board.empty()) return;

        const int m = board.size();
        const int n = board[0].size();
        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:
    void bfs(vector<vector<char>> &board, int i, int j) {
        typedef pair<int, int> state_t;
        queue<state_t> q;
        const int m = board.size();
        const int n = board[0].size();

        auto is_valid = [&](const state_t &s) {
            const int x = s.first;
            const int y = s.second;
            if (x < 0 || x >= m || y < 0 || y >= n || board[x][y] != 'O')
                return false;
            return true;
        };

        auto state_extend = [&](const state_t &s) {
            vector<state_t> result;
            const int x = s.first;
            const int y = s.second;
            // 上下左右
            const state_t new_states[4] = {{x-1,y}, {x+1,y},
                    {x,y-1}, {x,y+1}};
            for (int k = 0; k < 4;  ++k) {
                if (is_valid(new_states[k])) {
                    // 既有标记功能又有去重功能
                    board[new_states[k].first][new_states[k].second] = '+';
                    result.push_back(new_states[k]);
                }
            }

            return result;
```

```
        };

        state_t start = { i, j };
        if (is_valid(start)) {
            board[i][j] = '+';
            q.push(start);
        }
        while (!q.empty()) {
            auto cur = q.front();
            q.pop();
            auto new_states = state_extend(cur);
            for (auto s : new_states) q.push(s);
        }
    }
};
```

相关题目

- 无

9.4 小结

9.4.1 适用场景

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

9.4.2 思考的步骤

1. 是求路径长度，还是路径本身（或动作序列）？
   (a) 如果是求路径长度，则状态里面要存路径长度（或双队列 + 一个全局变量）
   (b) 如果是求路径本身或动作序列
      i. 要用一棵树存储宽搜过程中的路径
      ii. 是否可以预估状态个数的上限？能够预估状态总数，则开一个大数组，用树的双亲表示法；如果不能预估状态总数，则要使用一棵通用的树。这一步也是第 4 步的必要不充分条件。
2. 如何表示状态？即一个状态需要存储哪些些必要的数据，才能够完整提供如何扩展到下一步状态的所有信息。一般记录当前位置或整体局面。
3. 如何扩展状态？这一步跟第 2 步相关。状态里记录的数据不同，扩展方法就不同。对于固定不变的数据结构（一般题目直接给出，作为输入数据），如二叉树，图等，扩展方法很简单，

直接往下一层走，对于隐式图，要先在第 1 步里想清楚状态所带的数据，想清楚了这点，那如何扩展就很简单了。

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

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

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

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

9.4.3 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 `step`，表示走到当前状态用了多少步，当碰到目标状态，直接输出 `step` 即可。这个方案，可以很方便的变成 A* 算法，把队列换成优先队列即可。

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，状态可以少一个字段，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。

双队列的写法

bfs_template1.cpp

```
/** 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 count;  /** 所花费的步骤数（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.
 * @param[in] father 树
 * @param[in] target 目标节点
 * @return 从起点到 target 的路径
 */
template<typename state_t>
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;
}

/**
 * @brief 广搜.
 * @param[in] state_t 状态, 如整数, 字符串, 一维数组等
 * @param[in] start 起点
```

```
 * @param[in] grid 输入数据
 * @return 从起点到目标状态的一条最短路径
 */
template<typename state_t>
vector<state_t> bfs(const state_t &start, const vector<vector<int>> &grid) {
    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

    int level = 0;  // 层次
    bool found = false; // 是否找到目标
    state_t target; // 符合条件的目标状态

    // 判断当前状态是否为所求目标
    auto state_is_target = [&](const state_t &s) {return true; };
    // 扩展当前状态
    auto state_extend = [&](const state_t &s) {
        vector<state_t> result;
        // ...
        return result;
    };

    current.push(start);
    visited.insert(start);
    while (!current.empty() && !found) {
        ++level;
        while (!current.empty() && !found) {
            const state_t state = current.front();
            current.pop();
            vector<state_t> new_states = state_extend(state);
            for (auto iter = new_states.cbegin();
                    iter != new_states.cend() && ! found; ++iter) {
                const state_t new_state(*iter);

                if (state_is_target(new_state)) {
                    found = true; //找到了
                    target = new_state;
                    father[new_state] = state;
                    break;
                }

                next.push(new_state);
                // visited.insert(new_state); 必须放到 state_extend() 里
                father[new_state] = state;
            }
        }
        swap(next, current); //!!! 交换两个队列
    }

    if (found) {
        return gen_path(father, target);
        //return level + 1;
    } else {
```

```
        return vector<state_t>();
        //return 0;
    }
}
```

bfs_template1.cpp

只用一个队列的写法

双队列的写法，当求路径长度时，不需要在状态里设置一个 `count` 字段记录路径长度，只需全局设置一个整数 `level`，比较节省内存；只用一个队列的写法，当求路径长度时，需要在状态里设置一个 `count` 字段，不过，这种写法有一个好处——可以很容易的变为 A* 算法，把 `queue` 替换为 `priority_queue` 即可。

bfs_template2.cpp

```
// 与模板 1 相同的部分，不再重复
// ...

/**
 * @brief 广搜.
 * @param[in] state_t 状态，如整数，字符串，一维数组等
 * @param[in] start 起点
 * @param[in] grid 输入数据
 * @return 从起点到目标状态的一条最短路径
 */
template<typename state_t>
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

    int level = 0;  // 层次
    bool found = false; // 是否找到目标
    state_t target; // 符合条件的目标状态

    // 判断当前状态是否为所求目标
    auto state_is_target = [&](const state_t &s) {return true; };
    // 扩展当前状态
    auto state_extend = [&](const state_t &s) {
        vector<state_t> result;
        // ...
        return result;
    };

    start.count = 0;
    q.push(start);
    visited.insert(start);
    while (!q.empty() && !found) {
        const state_t state = q.front();
        q.pop();
        vector<state_t> new_states = state_extend(state);
        for (auto iter = new_states.cbegin();
                iter != new_states.cend() && ! found; ++iter) {
```

```
            const state_t new_state(*iter);

            if (state_is_target(new_state)) {
                found = true; //找到了
                target = new_state;
                father[new_state] = state;
                break;
            }

            q.push(new_state);
            // visited.insert(new_state); 必须放到 state_extend() 里
            father[new_state] = state;
        }
    }

    if (found) {
        return gen_path(father, target);
        //return level + 1;
    } else {
        return vector<state_t>();
        //return 0;
    }
}
```

bfs_template2.cpp

第 10 章
深度优先搜索

10.1 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

```
//LintCode, Palindrome Partitioning
// 时间复杂度 O(2^n), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 一个 partition 方案
        dfs(s, path, result, 0, 1);
        return result;
    }

    // s[0, prev-1] 之间已经处理, 保证是回文串
    // prev 表示 s[prev-1] 与 s[prev] 之间的空隙位置, start 同理
    void dfs(string &s, vector<string>& path,
            vector<vector<string>> &result, size_t prev, size_t start) {
        if (start == s.size()) { // 最后一个隔板
```

```
            if (isPalindrome(s, prev, start - 1)) { // 必须使用
                path.push_back(s.substr(prev, start - prev));
                result.push_back(path);
                path.pop_back();
            }
            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.push_back(s.substr(prev, start - prev));
            dfs(s, path, result, start, start + 1);
            path.pop_back();
        }
    }

    bool isPalindrome(const string &s, int start, int end) {
        while (start < end && s[start] == s[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。

```
//LintCode, Palindrome Partitioning
// 时间复杂度 O(2^n)，空间复杂度 O(n)
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 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
    void DFS(string &s, vector<string>& path,
            vector<vector<string>> &result, int start) {
        if (start == s.size()) {
            result.push_back(path);
            return;
        }
        for (int i = start; i < s.size(); i++) {
            if (isPalindrome(s, start, i)) { // 从 i 位置砍一刀
                path.push_back(s.substr(start, i - start + 1));
                DFS(s, path, result, i + 1);  // 继续往下砍
                path.pop_back(); // 撤销上上行
```

```
            }
        }
    }
    bool isPalindrome(const string &s, int start, int end) {
        while (start < end && s[start] == s[end]) {
            ++start;
            --end;
        }
        return start >= end;
    }
};
```

动规

```
// LintCode, Palindrome Partitioning
// 动规, 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    vector<vector<string> > partition(string s) {
        const int n = s.size();
        bool p[n][n]; // whether s[i,j] is palindrome
        fill_n(&p[0][0], n * n, false);
        for (int i = n - 1; i >= 0; --i)
            for (int j = i; j < n; ++j)
                p[i][j] = s[i] == s[j] && ((j - i < 2) || p[i + 1][j - 1]);

        vector<vector<string> > sub_palins[n]; // sub palindromes of s[0,i]
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i; j < n; ++j)
                if (p[i][j]) {
                    const string palindrome = s.substr(i, j - i + 1);
                    if (j + 1 < n) {
                        for (auto v : sub_palins[j + 1]) {
                            v.insert(v.begin(), palindrome);
                            sub_palins[i].push_back(v);
                        }
                    } else {
                        sub_palins[i].push_back(vector<string> { palindrome });
                    }
                }
        }
        return sub_palins[0];
    }
};
```

相关题目

- Palindrome Partitioning II，见 §13.3

10.2 Unique Paths

描述

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

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

How many possible unique paths are there?

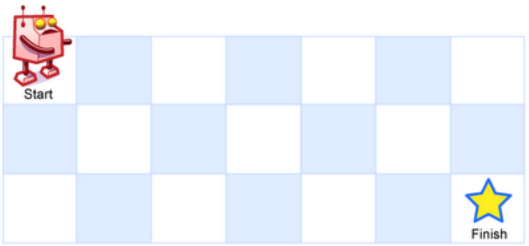


图 10-1 Above is a 3×7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

10.2.1 深搜

深搜，小集合可以过，大集合会超时

代码

```
// LintCode, 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)
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);
    }
};
```

10.2.2 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。

代码

```
// LintCode, 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)
class Solution {
public:
    int uniquePaths(int m, int n) {
        // 0 行和 0 列未使用
        this->f = vector<vector<int> >(m + 1, vector<int>(n + 1, 0));
        return dfs(m, n);
    }
private:
    vector<vector<int> > f;  // 缓存

    int dfs(int x, int y) {
        if (x < 1 || y < 1) return 0; // 数据非法, 终止条件

        if (x == 1 && y == 1) return 1; // 回到起点, 收敛条件

        return getOrUpdate(x - 1, y) + getOrUpdate(x, y - 1);
    }

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

10.2.3 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]
```

代码

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

```
        }
        return f[n - 1];
    }
};
```

10.2.4 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}。

代码

```
// LintCode, 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，见 §10.3
- Minimum Path Sum, 见 §13.8

10.3 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.

10.3.1 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。

代码

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

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

        // (x,y) 是障碍
        if (obstacleGrid[x-1][y-1]) return 0;

        if (x == 1 and y == 1) return 1; // 回到起点, 收敛条件

        return getOrUpdate(obstacleGrid, x - 1, y) +
               getOrUpdate(obstacleGrid, x, y - 1);
```

```
    }

    int getOrUpdate(const vector<vector<int> > &obstacleGrid,
            int x, int y) {
        if (f[x][y] > 0) return f[x][y];
        else return f[x][y] = dfs(obstacleGrid, x, y);
    }
};
```

10.3.2 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，那么后面的行应该为 0。

代码

```
// LintCode, Unique Paths II
// 动规, 滚动数组
// 时间复杂度 O(n^2), 空间复杂度 O(n)
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();
        if (obstacleGrid[0][0] || obstacleGrid[m-1][n-1]) return 0;

        vector<int> f(n, 0);
        f[0] = obstacleGrid[0][0] ? 0 : 1;

        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                f[j] = obstacleGrid[i][j] ? 0 : (j == 0 ? 0 : f[j - 1]) + f[j];

        return f[n - 1];
    }
};
```

相关题目

- Unique Paths，见 §10.2
- Minimum Path Sum, 见 §13.8

10.4 N-Queens

描述

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

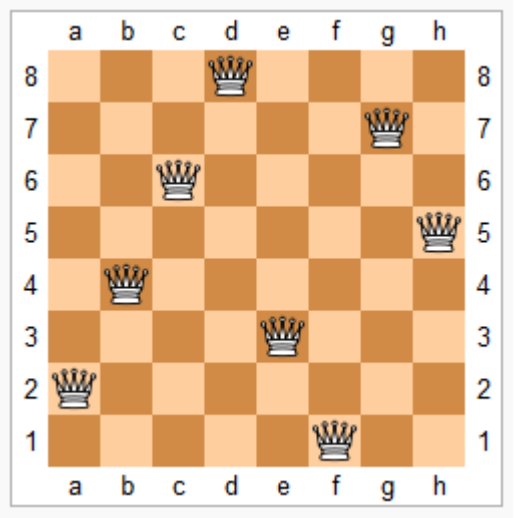


图 10-2 Eight Queens

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。

代码

```
// LintCode, N-Queens
// 深搜 + 剪枝
// 时间复杂度 O(n!), 空间复杂度 O(n)
class Solution {
public:
```

```
    vector<vector<string> > solveNQueens(int n) {
        this->columns = vector<int>(n, 0);
        this->main_diag = vector<int>(2 * n, 0);
        this->anti_diag = vector<int>(2 * n, 0);

        vector<vector<string> > result;
        vector<int> C(n, 0);  // C[i] 表示第 i 行皇后所在的列编号
        dfs(C, result, 0);
        return result;
    }
private:
    // 这三个变量用于剪枝
    vector<int> columns;  // 表示已经放置的皇后占据了哪些列
    vector<int> main_diag;  // 占据了哪些主对角线
    vector<int> anti_diag;  // 占据了哪些副对角线

    void dfs(vector<int> &C, vector<vector<string> > &result, int row) {
        const int N = C.size();
        if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            vector<string> solution;
            for (int i = 0; i < N; ++i) {
                string s(N, '.');
                for (int j = 0; j < N; ++j) {
                    if (j == C[i]) s[j] = 'Q';
                }
                solution.push_back(s);
            }
            result.push_back(solution);
            return;
        }

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            const bool ok = columns[j] == 0 && main_diag[row + j] == 0 &&
                    anti_diag[row - j + N] == 0;
            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] = anti_diag[row - j + N] = 1;
            dfs(C, result, row + 1);
            // 撤销动作
            // C[row] = 0;
            columns[j] = main_diag[row + j] = anti_diag[row - j + N] = 0;
        }
    }
};
```

相关题目

- N-Queens II，见 §10.5

10.5 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

```
// LintCode, N-Queens II
// 深搜 + 剪枝
// 时间复杂度 O(n!), 空间复杂度 O(n)
class Solution {
public:
    int totalNQueens(int n) {
        this->count = 0;
        this->columns = vector<int>(n, 0);
        this->main_diag = vector<int>(2 * n, 0);
        this->anti_diag = vector<int>(2 * n, 0);

        vector<int> C(n, 0);  // C[i] 表示第 i 行皇后所在的列编号
        dfs(C, 0);
        return this->count;
    }
private:
    int count; // 解的个数
    // 这三个变量用于剪枝
    vector<int> columns;  // 表示已经放置的皇后占据了哪些列
    vector<int> main_diag;  // 占据了哪些主对角线
    vector<int> anti_diag;  // 占据了哪些副对角线

    void dfs(vector<int> &C, int row) {
        const int N = C.size();
        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
            const bool ok = columns[j] == 0 &&
                    main_diag[row + j] == 0 &&
                    anti_diag[row - j + N] == 0;
            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] =
                        anti_diag[row - j + N] = 1;
                dfs(C, row + 1);
                // 撤销动作
                // C[row] = 0;
                columns[j] = main_diag[row + j] =
                        anti_diag[row - j + N] = 0;
            }
        }
};
```

相关题目

- N-Queens，见 §10.4

10.6 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

```
// LintCode, Restore IP Addresses
// 时间复杂度 O(n^4), 空间复杂度 O(n)
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        string ip; // 存放中间结果
        dfs(s, 0, 0, ip, result);
        return result;
    }

    /**
     * @brief 解析字符串
     * @param[in] s 字符串, 输入数据
     * @param[in] startIndex 从 s 的哪里开始
     * @param[in] step 当前步骤编号, 从 0 开始编号, 取值为 0,1,2,3,4 表示结束了
     * @param[out] intermediate 当前解析出来的中间结果
     * @param[out] result 存放所有可能的 IP 地址
     * @return 无
```

```
     */
    void dfs(string s, size_t start, size_t step, string ip,
            vector<string> &result) {
        if (start == s.size() && step == 4) {  // 找到一个合法解
            ip.resize(ip.size() - 1);
            result.push_back(ip);
            return;
        }

        if (s.size() - start > (4 - step) * 3)
            return;  // 剪枝
        if (s.size() - start < (4 - step))
            return;  // 剪枝

        int num = 0;
        for (size_t i = start; i < start + 3; i++) {
            num = num * 10 + (s[i] - '0');

            if (num <= 255) {  // 当前结点合法，则继续往下递归
                ip += s[i];
                dfs(s, i + 1, step + 1, ip + '.', result);
            }
            if (num == 0) break;  // 不允许前缀 0，但允许单个 0
        }
    }
};
```

相关题目

- 无

10.7 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

```
// LintCode, Combination Sum
// 时间复杂度 O(n!), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > combinationSum(vector<int> &nums, int target) {
        sort(nums.begin(), nums.end());
        vector<vector<int> > result; // 最终结果
        vector<int> intermediate; // 中间结果
        dfs(nums, target, 0, intermediate, result);
        return result;
    }

private:
    void dfs(vector<int>& nums, int gap, int start, vector<int>& intermediate,
            vector<vector<int> > &result) {
        if (gap == 0) {  // 找到一个合法解
            result.push_back(intermediate);
            return;
        }
        for (size_t i = start; i < nums.size(); i++) { // 扩展状态
            if (gap < nums[i]) return; // 剪枝

            intermediate.push_back(nums[i]); // 执行扩展动作
            dfs(nums, gap - nums[i], i, intermediate, result);
            intermediate.pop_back();  // 撤销动作
        }
    }
};
```

相关题目

- Combination Sum II，见 §10.8

10.8 Combination Sum II

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 once 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, $a_1 > a_2 > ... > a_k$).
- 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

```
// LintCode, Combination Sum II
// 时间复杂度 O(n!), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > combinationSum2(vector<int> &nums, int target) {
        sort(nums.begin(), nums.end()); // 跟第 50 行配合,
                                        // 确保每个元素最多只用一次
        vector<vector<int> > result;
        vector<int> intermediate;
        dfs(nums, target, 0, intermediate, result);
        return result;
    }
private:
    // 使用 nums[start, nums.size()) 之间的元素, 能找到的所有可行解
    static void dfs(vector<int> &nums, int gap, int start,
            vector<int> &intermediate, vector<vector<int> > &result) {
        if (gap == 0) {  //  找到一个合法解
            result.push_back(intermediate);
            return;
        }

        int previous = -1;
        for (size_t i = start; i < nums.size(); i++) {
            // 如果上一轮循环没有选 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];

            intermediate.push_back(nums[i]);
            dfs(nums, gap - nums[i], i + 1, intermediate, result);
            intermediate.pop_back();  // 恢复环境
        }
    }
};
```

相关题目

- Combination Sum，见 §10.7

10.9 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

```
// LintCode, Generate Parentheses
// 时间复杂度 O(TODO), 空间复杂度 O(n)
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        if (n > 0) generate(n, "", 0, 0, result);
        return result;
    }
    // l 表示 ( 出现的次数, r 表示 ) 出现的次数
    void generate(int n, string s, int l, int r, vector<string> &result) {
        if (l == n) {
            result.push_back(s.append(n - r, ')'));
            return;
        }
        generate(n, s + '(', l + 1, r, result);
        if (l > r) generate(n, s + ")", l, r + 1, result);
    }
};
```

代码 2

另一种递归写法，更加简洁。

```
// LintCode, Generate Parentheses
// @author 连城 (http://weibo.com/lianchengzju)
class Solution {
public:
```

```
    vector<string> generateParenthesis (int n) {
        if (n == 0) return vector<string>(1, "");
        if (n == 1) return vector<string> (1, "()");
        vector<string> result;

        for (int i = 0; i < n; ++i)
            for (auto inner : generateParenthesis (i))
                for (auto outer : generateParenthesis (n - 1 - i))
                    result.push_back ("(" + inner + ")" + outer);

        return result;
    }
};
```

相关题目

- Valid Parentheses, 见 §4.1.1
- Longest Valid Parentheses, 见 §4.1.2

10.10 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.

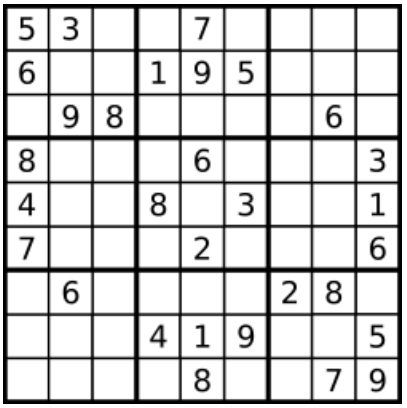


图 10-3 A sudoku puzzle...

图 10-4 ...and its solution numbers marked in red

分析

无。

代码

```
// LintCode, Sudoku Solver
// 时间复杂度 O(9^4), 空间复杂度 O(1)
class Solution {
public:
    bool solveSudoku(vector<vector<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] = '1' + k;
                        if (isValid(board, i, j) && solveSudoku(board))
                            return true;
                        board[i][j] = '.';
                    }
                    return false;
                }
            }
        return true;
    }
private:
    // 检查 (x, y) 是否合法
    bool isValid(const vector<vector<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, 见 §2.1.14

10.11 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

```
// LintCode, Word Search
// 深搜, 递归
// 时间复杂度 O(n^2*m^2), 空间复杂度 O(n^2)
class Solution {
public:
    bool exist(vector<vector<char> > &board, string word) {
        const int m = board.size();
        const int n = board[0].size();
        vector<vector<bool> > visited(m, vector<bool>(n, false));
```

```
        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 bool dfs(const vector<vector<char> > &board, const string &word,
            int index, int x, int y, vector<vector<bool> > &visited) {
        if (index == word.size())
            return true; // 收敛条件

        if (x < 0 || y < 0 || x >= board.size() || y >= board[0].size())
            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[index]) return false; // 不相等，剪枝

        visited[x][y] = true;
        bool 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;
    }
};
```

相关题目

- 无

10.12 小结

10.12.1 适用场景

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

状态转换图：树或者图。

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

10.12.2 思考的步骤

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

(a) 如果是路径条数，则不需要存储路径。

(b) 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. 关于判重

   (a) 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。

   (b) 怎样判重？跟广搜相同，见第 §9.4 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。

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

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

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

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

8. 如何加速？

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

(b) 缓存。

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。

10.12.3 代码模板

dfs_template.cpp

```
/**
 * 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
    }
}
```

dfs_template.cpp

10.12.4 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。

10.12.5 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。

第 11 章
分治法

11.1 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

```
//LintCode, 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)
class Solution {
public:
    double pow(double x, int n) {
        if (n < 0) return 1.0 / power(x, -n);
        else return power(x, n);
    }
private:
    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)，见 §11.2

11.2 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

```
// LintCode, Sqrt(x)
// 二分查找
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    int sqrt(int x) {
        int left = 1, right = x / 2;
        int last_mid;  // 记录最近一次 mid

        if (x < 2) return x;

        while(left <= right) {
            const int mid = left + (right - left) / 2;
            if(x / mid > mid) { // 不要用 x > mid * mid, 会溢出
                left = mid + 1;
                last_mid = mid;
            } else if(x / mid < mid) {
                right = mid - 1;
            } else {
                return mid;
            }
        }
        return last_mid;
    }
};
```

相关题目

- Pow(x)，见 §11.1

第 12 章
贪心法

12.1 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

```
// LintCode, Jump Game
// 思路 1，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    bool canJump(const vector<int> &A) {
        const int n = A.size();
        int reach = 1; // 最右能跳到哪里
        for (int i = 0; i < reach && reach < n; ++i)
```

```
            reach = max(reach,  i + 1 + A[i]);
        return reach >= n;
    }
};
```

代码 2

```
// LintCode, Jump Game
// 思路 2, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    bool canJump (const vector<int> &A) {
        const int n = A.size();
        if (n == 0) return true;
        // 逆向下楼梯, 最左能下降到第几层
        int left_most = n - 1;

        for (int i = n - 2; i >= 0; --i)
            if (i + A[i] >= left_most)
                left_most = i;

        return left_most == 0;
    }
};
```

代码 3

```
// LintCode, Jump Game
// 思路三, 动规, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    bool canJump(const vector<int> &A) {
        const int n = A.size();
        vector<int> f(n, 0);
        f[0] = 0;
        for (int i = 1; i < n; i++) {
            f[i] = max(f[i - 1], A[i - 1]) - 1;
            if (f[i] < 0) return false;;
        }
        return f[n - 1] >= 0;
    }
};
```

相关题目

- Jump Game II，见 §12.2

12.2 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

```
// LintCode, Jump Game II
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int jump(int A[], int n) {
        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 (n == 1) return 0;

        while (left <= right) { // 尝试从每一层跳最远
            ++step;
            const int old_right = right;
            for (int i = left; i <= old_right; ++i) {
                int new_right = i + A[i];
                if (new_right >= n - 1) return step;

                if (new_right > right) right = new_right;
            }
            left = old_right + 1;
        }
        return 0;
    }
};
```

代码 2

```
// LintCode, Jump Game II
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
```

```
    int jump(const vector<int> &A) {
        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 < A.size(); ++i) {
            if (i > last) {
                last = cur;
                ++result;
            }
            cur = max(cur, i + A[i]);
        }

        return result;
    }
};
```

相关题目

- Jump Game，见 §12.1

12.3 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

```
// LintCode, Best Time to Buy and Sell Stock
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if (prices.size() < 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.size(); i++) {
```

```
            profit = max(profit, prices[i] - cur_min);
            cur_min = min(cur_min, prices[i]);
        }
        return profit;
    }
};
```

相关题目

- Best Time to Buy and Sell Stock II，见 §12.4
- Best Time to Buy and Sell Stock III，见 §13.5

12.4 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

```
// LintCode, Best Time to Buy and Sell Stock II
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int sum = 0;
        for (int i = 1; i < prices.size(); 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，见 §12.3
- Best Time to Buy and Sell Stock III，见 §13.5

12.5 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 $O(n)$。如图 12-1所示。

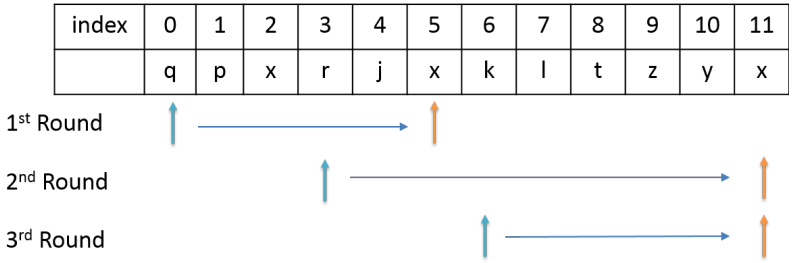


图 12-1 不含重复字符的最长子串

代码

```
// LintCode, Longest Substring Without Repeating Characters
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        const int ASCII_MAX = 26;
        int last[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

        fill(last, last + ASCII_MAX, -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.size(); i++) {
            if (last[s[i] - 'a'] >= start) {
                max_len = max(i - start, max_len);
                start = last[s[i] - 'a'] + 1;
            }
            last[s[i] - 'a'] = i;
        }
        return max((int)s.size() - start, max_len);  // 别忘了最后一次, 例如"abcd"
```

```
    }
};
```

相关题目

- 无

12.6 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

```
// LintCode, Container With Most Water
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int maxArea(vector<int> &height) {
        int start = 0;
        int end = height.size() - 1;
        int result = INT_MIN;
        while (start < end) {
            int area = min(height[end], height[start]) * (end - start);
            result = max(result, area);
            if (height[start] <= height[end]) {
                start++;
            } else {
                end--;
            }
        }
        return result;
    }
};
```

相关题目

- Trapping Rain Water, 见 §2.1.15

- Largest Rectangle in Histogram, 见 §4.1.3

第 13 章
动态规划

13.1 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

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

代码

```
// LintCode, Triangle
// 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    int minimumTotal (vector<vector<int>>& triangle) {
        for (int i = triangle.size() - 2; i >= 0; --i)
            for (int j = 0; j < i + 1; ++j)
                triangle[i][j] += min(triangle[i + 1][j],
                        triangle[i + 1][j + 1]);
```

```
        return triangle [0][0];
    }
};
```

相关题目

- 无

13.2 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：

$$\begin{aligned} f[j] &= \max\{f[j-1]+S[j], S[j]\}, \text{其中} 1 \le j \le n \\ target &= \max\{f[j]\}, \text{其中} 1 \le j \le n \end{aligned}$$

解释如下：

- 情况一，S[j] 不独立，与前面的某些数组成一个连续子序列，则最大连续子序列和为 $f[j-1]+S[j]$。
- 情况二，S[j] 独立划分成为一段，即连续子序列仅包含一个数 S[j]，则最大连续子序列和为 $S[j]$。

其他思路：

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

动规

```
// LintCode, Maximum Subarray
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int maxSubArray(const vector<int> &A) {
        int result = INT_MIN, f = 0;
        for (int i = 0; i < A.size(); ++i) {
            f = max(f + A[i], A[i]);
            result = max(result, f);
        }
        return result;
    }
};
```

思路 5

```
// LintCode, Maximum Subarray
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    int maxSubArray(const vector<int> &A) {
        return mcss(A);
    }
private:
    // 思路 5, 求最大连续子序列和
    static int mcss(const vector<int> &A) {
        int i, result, cur_min;
        const int n = A.size();
        vector<int> sum(n + 1, 0);  // 前 n 项和

        sum[0] = 0;
        result = INT_MIN;
        cur_min = sum[0];
        for (i = 1; i <= n; i++) {
            sum[i] = sum[i - 1] + A[i - 1];
        }
        for (i = 1; i <= n; i++) {
            result = max(result, sum[i] - cur_min);
            cur_min = min(cur_min, sum[i]);
        }
        return result;
    }
};
```

相关题目

- Binary Tree Maximum Path Sum，见 §5.4.5

13.3 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

```
// LintCode, Palindrome Partitioning II
// 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
public:
    int minCut(string s) {
        const int n = s.size();
        int f[n+1];
        bool p[n][n];
        fill_n(&p[0][0], n * n, false);
        //the worst case is cutting by each char
        for (int i = 0; i <= n; i++)
            f[i] = n - 1 - i; // 最后一个 f[n]=-1
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                if (s[i] == s[j] && (j - i < 2 || p[i + 1][j - 1])) {
                    p[i][j] = true;
                    f[i] = min(f[i], f[j + 1] + 1);
                }
            }
```

```
        }
        return f[0];
    }
};
```

相关题目

- Palindrome Partitioning，见 §10.1

13.4 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

```
// LintCode, Maximal Rectangle
// 时间复杂度 O(n^2), 空间复杂度 O(n)
class Solution {
public:
    int maximalRectangle(vector<vector<char> > &matrix) {
        if (matrix.empty())  return 0;

        const int m = matrix.size();
        const int n = matrix[0].size();
        vector<int> H(n, 0);
        vector<int> L(n, 0);
        vector<int> R(n, 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] = 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] = min(R[j], right);
                    ret = max(ret, H[j]*(R[j]-L[j]));
                } else {
                    right = j;
                }
            }
        }
        return ret;
    }
};
```

相关题目

- 无

13.5 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

```
// LintCode, Best Time to Buy and Sell Stock III
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() < 2) return 0;

        const int n = prices.size();
        vector<int> f(n, 0);
```

```
        vector<int> g(n, 0);

        for (int i = 1, valley = prices[0]; i < n; ++i) {
            valley = min(valley, prices[i]);
            f[i] = max(f[i - 1], prices[i] - valley);
        }

        for (int i = n - 2, peak = prices[n - 1]; i >= 0; --i) {
            peak = max(peak, prices[i]);
            g[i] = max(g[i], peak - prices[i]);
        }

        int max_profit = 0;
        for (int i = 0; i < n; ++i)
            max_profit = max(max_profit, f[i] + g[i]);

        return max_profit;
    }
};
```

相关题目

- Best Time to Buy and Sell Stock，见 §12.3
- Best Time to Buy and Sell Stock II，见 §12.4

13.6 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

```
// LintCode, Interleaving String
// 递归，会超时，仅用来帮助理解
```

```
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s3.length() != s1.length() + s2.length())
            return false;

        return isInterleave(begin(s1), end(s1), begin(s2), end(s2),
                begin(s3), end(s3));
    }

    template<typename InIt>
    bool isInterleave(InIt first1, InIt last1, InIt first2, InIt last2,
            InIt first3, InIt last3) {
        if (first3 == last3)
            return first1 == last1 && first2 == last2;

        return (*first1 == *first3
                && isInterleave(next(first1), last1, first2, last2,
                        next(first3), last3))
                || (*first2 == *first3
                        && isInterleave(first1, last1, next(first2), last2,
                                next(first3), last3));
    }
};
```

动规

```
// LintCode, Interleaving String
// 二维动规, 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s3.length() != s1.length() + s2.length())
            return false;

        vector<vector<bool>> f(s1.length() + 1,
                vector<bool>(s2.length() + 1, true));

        for (size_t i = 1; i <= s1.length(); ++i)
            f[i][0] = f[i - 1][0] && s1[i - 1] == s3[i - 1];

        for (size_t i = 1; i <= s2.length(); ++i)
            f[0][i] = f[0][i - 1] && s2[i - 1] == s3[i - 1];

        for (size_t i = 1; i <= s1.length(); ++i)
            for (size_t j = 1; j <= s2.length(); ++j)
                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]);

        return f[s1.length()][s2.length()];
    }
};
```

动规 + 滚动数组

```
// LintCode, 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)
class Solution {
public:
    bool 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);

        vector<bool> f(s2.length() + 1, true);

        for (size_t i = 1; i <= s2.length(); ++i)
            f[i] = s2[i - 1] == s3[i - 1] && f[i - 1];

        for (size_t i = 1; i <= s1.length(); ++i) {
            f[0] = s1[i - 1] == s3[i - 1] && f[0];

            for (size_t j = 1; j <= s2.length(); ++j)
                f[j] = (s1[i - 1] == s3[i + j - 1] && f[j])
                        || (s2[j - 1] == s3[i + j - 1] && f[j - 1]);
        }

        return f[s2.length()];
    }
};
```

相关题目

- 无

13.7 Scramble String

描述

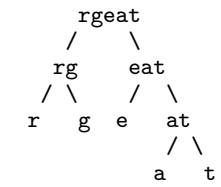
Given a string $s1$, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

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

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

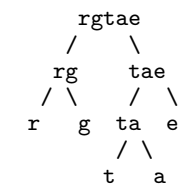
To scramble the string, we may choose any non-leaf node and swap its two children.

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



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

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



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

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

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

```
// LintCode, Interleaving 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)
class Solution {
```

```
public:
    bool isScramble(string s1, string s2) {
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
private:
    typedef string::iterator Iterator;
    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

        if (length == 1) return *first1 == *first2;

        for (int i = 1; i < length; ++i)
            if ((isScramble(first1, first1 + i, first2)
                 && isScramble(first1 + i, last1, first2 + i))
                    || (isScramble(first1, first1 + i, last2 - i)
                            && isScramble(first1 + i, last1, first2)))
                return true;

        return false;
    }
};
```

动规

```
// LintCode, Interleaving String
// 动规, 时间复杂度 O(n^3), 空间复杂度 O(n^3)
class Solution {
public:
    bool isScramble(string s1, string s2) {
        const int N = s1.size();
        if (N != s2.size()) 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
        bool f[N + 1][N][N];
        fill_n(&f[0][0][0], (N + 1) * N * N, false);

        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                f[1][i][j] = s1[i] == s2[j];

        for (int n = 1; n <= N; ++n) {
            for (int i = 0; i + n <= N; ++i) {
                for (int j = 0; j + n <= N; ++j) {
                    for (int k = 1; k < n; ++k) {
                        if ((f[k][i][j] && f[n - k][i + k][j + k]) ||
                                (f[k][i][j + n - k] && f[n - k][i + k][j])) {
                            f[n][i][j] = true;
                            break;
                        }
                    }
                }
```

```
            }
        }
        return f[N][0][0];
    }
};
```

递归 + 剪枝

```
// LintCode, Interleaving String
// 递归 + 剪枝
// 时间复杂度 O(n^6), 空间复杂度 O(1)
class Solution {
public:
    bool isScramble(string s1, string s2) {
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
private:
    typedef string::iterator Iterator;
    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);
        if (length == 1) return *first1 == *first2;

        // 剪枝, 提前返回
        int A[26]; // 每个字符的计数器
        fill(A, A + 26, 0);
        for(int i = 0; i < length; i++) A[*(first1+i)-'a']++;
        for(int i = 0; i < length; i++) A[*(first2+i)-'a']--;
        for(int i = 0; i < 26; i++) if (A[i] != 0) return false;

        for (int i = 1; i < length; ++i)
            if ((isScramble(first1, first1 + i, first2)
                 && isScramble(first1 + i, last1, first2 + i))
                    || (isScramble(first1, first1 + i, last2 - i)
                            && isScramble(first1 + i, last1, first2)))
                return true;

        return false;
    }
};
```

备忘录法

```
// LintCode, Interleaving String
// 递归 +map 做 cache
// 时间复杂度 O(n^3), 空间复杂度 O(n^3)
class Solution {
public:
    bool isScramble(string s1, string s2) {
        cache.clear();
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
```

```
private:
    typedef string::const_iterator Iterator;
    map<tuple<Iterator, Iterator, Iterator>, bool> cache;

    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

        if (length == 1) return *first1 == *first2;

        for (int i = 1; i < length; ++i)
            if ((getOrUpdate(first1, first1 + i, first2)
                    && getOrUpdate(first1 + i, last1, first2 + i))
                    || (getOrUpdate(first1, first1 + i, last2 - i)
                            && getOrUpdate(first1 + i, last1, first2)))
                return true;

        return false;
    }

    bool getOrUpdate(Iterator first1, Iterator last1, Iterator first2) {
        auto key = make_tuple(first1, last1, first2);
        auto pos = cache.find(key);

        return (pos != cache.end()) ?
                pos->second : (cache[key] = isScramble(first1, last1, first2));
    }
};
```

备忘录法

```
typedef string::const_iterator Iterator;
typedef tuple<Iterator, Iterator, Iterator> Key;
// 定制一个哈希函数
namespace std {
template<> struct hash<Key> {
    size_t operator()(const Key & x) const {
        Iterator first1, last1, first2;
        tie(first1, last1, first2) = x;

        int result = *first1;
        result = result * 31 + *last1;
        result = result * 31 + *first2;
        result = result * 31 + *(next(first2, distance(first1, last1)-1));
        return result;
    }
};
}

// LintCode, Interleaving String
// 递归 +unordered_map 做 cache, 比 map 快
// 时间复杂度 O(n^3), 空间复杂度 O(n^3)
class Solution {
```

```
public:
    unordered_map<Key, bool> cache;

    bool isScramble(string s1, string s2) {
        cache.clear();
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }

    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

        if (length == 1)
            return *first1 == *first2;

        for (int i = 1; i < length; ++i)
            if ((getOrUpdate(first1, first1 + i, first2)
                    && getOrUpdate(first1 + i, last1, first2 + i))
                    || (getOrUpdate(first1, first1 + i, last2 - i)
                            && getOrUpdate(first1 + i, last1, first2)))
                return true;

        return false;
    }

    bool getOrUpdate(Iterator first1, Iterator last1, Iterator first2) {
        auto key = make_tuple(first1, last1, first2);
        auto pos = cache.find(key);

        return (pos != cache.end()) ?
                pos->second : (cache[key] = isScramble(first1, last1, first2));
    }
};
```

相关题目

- 无

13.8 Minimum Path Sum

描述

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

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

分析

跟 Unique Paths (见 §10.2) 很类似。

设状态为 `f[i][j]`，表示从起点 $(0,0)$ 到达 (i,j) 的最小路径和，则状态转移方程为：

```
f[i][j]=min(f[i-1][j], f[i][j-1])+grid[i][j]
```

备忘录法

```
// LintCode, Minimum Path Sum
// 备忘录法
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        const int m = grid.size();
        const int n = grid[0].size();
        this->f = vector<vector<int> >(m, vector<int>(n, -1));
        return dfs(grid, m-1, n-1);
    }
private:
    vector<vector<int> > f;  // 缓存

    int dfs(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX; // 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 min(getOrUpdate(grid, x - 1, y),
                getOrUpdate(grid, x, y - 1)) + grid[x][y];
    }

    int getOrUpdate(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX; // 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);
    }
};
```

动规

```
// LintCode, Minimum Path Sum
// 二维动规
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        if (grid.size() == 0) return 0;
        const int m = grid.size();
        const int n = grid[0].size();

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

动规 + 滚动数组

```
// LintCode, Minimum Path Sum
// 二维动规 + 滚动数组
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        const int m = grid.size();
        const int n = grid[0].size();

        int f[n];
        fill(f, f+n, INT_MAX); // 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] = min(f[j - 1], f[j]) + grid[i][j];
            }
        }
        return f[n - 1];
    }
};
```

相关题目

- Unique Paths, 见 §10.2
- Unique Paths II, 见 §10.3

13.9 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`，
   (a) 如果将 c 替换成 d，则 `f[i][j]=f[i-1][j-1]+1`；
   (b) 如果在 c 后面添加一个 d，则 `f[i][j]=f[i][j-1]+1`；
   (c) 如果将 c 删除，则 `f[i][j]=f[i-1][j]+1`；

动规

```
// LintCode, Edit Distance
// 二维动规，时间复杂度 O(n*m)，空间复杂度 O(n*m)
class Solution {
public:
    int minDistance(const string &word1, const string &word2) {
        const size_t n = word1.size();
        const size_t m = word2.size();
        // 长度为 n 的字符串，有 n+1 个隔板
        int f[n + 1][m + 1];
        for (size_t i = 0; i <= n; i++)
            f[i][0] = i;
        for (size_t j = 0; j <= m; j++)
            f[0][j] = j;

        for (size_t i = 1; i <= n; i++) {
            for (size_t j = 1; j <= m; j++) {
                if (word1[i - 1] == word2[j - 1])
                    f[i][j] = f[i - 1][j - 1];
                else {
                    int mn = min(f[i - 1][j], f[i][j - 1]);
                    f[i][j] = 1 + min(f[i - 1][j - 1], mn);
                }
            }
        }
        return f[n][m];
    }
};
```

动规 + 滚动数组

```
// LintCode, 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)
class Solution {
public:
    int minDistance(const string &word1, const string &word2) {
        if (word1.length() < word2.length())
            return minDistance(word2, word1);

        int f[word2.length() + 1];
        int upper_left = 0; // 额外用一个变量记录 f[i-1][j-1]

        for (size_t i = 0; i <= word2.size(); ++i)
            f[i] = i;

        for (size_t i = 1; i <= word1.size(); ++i) {
            upper_left = f[0];
            f[0] = i;

            for (size_t j = 1; j <= word2.size(); ++j) {
                int upper = f[j];

                if (word1[i - 1] == word2[j - 1])
                    f[j] = upper_left;
                else
                    f[j] = 1 + min(upper_left, min(f[j], f[j - 1]));

                upper_left = upper;
            }
        }

        return f[word2.length()];
    }
};
```

相关题目

- 无

13.10 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 Climbing Stairs (见 §2.1.18) 很类似，不过多加一些判断逻辑。

代码

```
// LintCode, Decode Ways
// 动规, 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    int numDecodings(const string &s) {
        if (s.empty() || s[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 (size_t i = 1; i <= s.size(); ++i) {
            if (s[i-1] == '0') cur = 0;

            if (i < 2 || !(s[i - 2] == '1' ||
                     (s[i - 2] == '2' && s[i - 1] <= '6')))
                prev = 0;

            int tmp = cur;
            cur = prev + cur;
            prev = tmp;
        }
        return cur;
    }
};
```

相关题目

- Climbing Stairs, 见 §2.1.18

13.11 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

```
// LintCode, 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)
class Solution {
public:
    int numDistinct(const string &S, const string &T) {
        vector<int> f(T.size() + 1);
        f[0] = 1;
        for (int i = 0; i < S.size(); ++i) {
            for (int j = T.size() - 1; j >= 0; --j) {
                f[j + 1] += S[i] == T[j] ? f[j] : 0;
            }
        }

        return f[T.size()];
    }
};
```

相关题目

- 无

13.12 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 = "lintcode",

dict = ["leet", "code"].

Return true because "lintcode" 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

$$f(i) = any\_of(f(j)\&\&s[j+1,i] \in dict), 0 \leq j < i$$

深搜

```
// LintCode, Word Break
// 深搜, 超时
// 时间复杂度 O(2^n), 空间复杂度 O(n)
class Solution {
public:
    bool wordBreak(string s, unordered_set<string> &dict) {
        return dfs(s, dict, 0, 0);
    }
private:
    static bool dfs(const string &s, unordered_set<string> &dict,
            size_t start, size_t cur) {
        if (cur == s.size()) {
            return dict.find(s.substr(start, cur-start+1)) != dict.end();
        }
        if (dfs(s, dict, start, cur+1)) return true;
        if (dict.find(s.substr(start, cur-start+1)) != dict.end())
            if (dfs(s, dict, cur+1, cur+1)) return true;
        return false;
    }
};
```

动规

```
// LintCode, Word Break
// 动规, 时间复杂度 O(n^2), 空间复杂度 O(n)
class Solution {
public:
    bool wordBreak(string s, unordered_set<string> &dict) {
        // 长度为 n 的字符串有 n+1 个隔板
        vector<bool> f(s.size() + 1, false);
        f[0] = true; // 空字符串
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (f[j] && dict.find(s.substr(j, i - j)) != dict.end()) {
                    f[i] = true;
                    break;
                }
            }
        }
        return f[s.size()];
    }
};
```

相关题目

- Word Break II, 见 §13.13

13.13 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

```
// LintCode, Word Break II
// 动规, 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
public:
    vector<string> wordBreak(string s, unordered_set<string> &dict) {
        // 长度为 n 的字符串有 n+1 个隔板
        vector<bool> f(s.length() + 1, false);
        // 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
        vector<vector<bool> > prev(s.length() + 1, vector<bool>(s.length()));
        f[0] = true;
        for (size_t i = 1; i <= s.length(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (f[j] && dict.find(s.substr(j, i - j)) != dict.end()) {
                    f[i] = true;
                    prev[i][j] = true;
                }
            }
        }
        vector<string> result;
        vector<string> path;
        gen_path(s, prev, s.length(), path, result);
        return result;

    }
```

```
private:
    // DFS 遍历树, 生成路径
    void gen_path(const string &s, const vector<vector<bool> > &prev,
            int cur, vector<string> &path, vector<string> &result) {
        if (cur == 0) {
            string tmp;
            for (auto iter = path.crbegin(); iter != path.crend(); ++iter)
                tmp += *iter + " ";
            tmp.erase(tmp.end() - 1);
            result.push_back(tmp);
        }
        for (size_t i = 0; i < s.size(); ++i) {
            if (prev[cur][i]) {
                path.push_back(s.substr(i, cur - i));
                gen_path(s, prev, i, path, result);
                path.pop_back();
            }
        }
    }
};
```

相关题目

- Word Break, 见 §13.12

第 14 章
图

无向图的节点定义如下：

```
// 无向图的节点
struct UndirectedGraphNode {
    int label;
    vector<UndirectedGraphNode *> neighbors;
    UndirectedGraphNode(int x) : label(x) {};
};
```

14.1 Clone Graph

描述

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

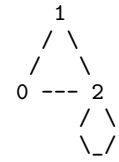
OJ's undirected graph serialization: Nodes are labeled uniquely.

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

```
// LintCode, Clone Graph
// DFS, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    UndirectedGraphNode *cloneGraph(const UndirectedGraphNode *node) {
        if(node == nullptr) return nullptr;
        // key is original node, value is copied node
        unordered_map<const UndirectedGraphNode *,
                      UndirectedGraphNode *> copied;
        clone(node, copied);
        return copied[node];
    }
private:
    // DFS
    static UndirectedGraphNode* clone(const UndirectedGraphNode *node,
            unordered_map<const UndirectedGraphNode *,
            UndirectedGraphNode *> &copied) {
        // a copy already exists
        if (copied.find(node) != copied.end()) return copied[node];

        UndirectedGraphNode *new_node = new UndirectedGraphNode(node->label);
        copied[node] = new_node;
        for (auto nbr : node->neighbors)
            new_node->neighbors.push_back(clone(nbr, copied));
        return new_node;
    }
};
```

BFS

```
// LintCode, Clone Graph
// BFS, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
public:
    UndirectedGraphNode *cloneGraph(const UndirectedGraphNode *node) {
        if (node == nullptr) return nullptr;
        // key is original node, value is copied node
        unordered_map<const UndirectedGraphNode *,
                      UndirectedGraphNode *> copied;
        // each node in queue is already copied itself
        // but neighbors are not copied yet
        queue<const UndirectedGraphNode *> q;
        q.push(node);
        copied[node] = new UndirectedGraphNode(node->label);
        while (!q.empty()) {
            const UndirectedGraphNode *cur = q.front();
            q.pop();
            for (auto nbr : cur->neighbors) {
                // a copy already exists
                if (copied.find(nbr) != copied.end()) {
                    copied[cur]->neighbors.push_back(copied[nbr]);
```

```
                } else {
                    UndirectedGraphNode *new_node =
                            new UndirectedGraphNode(nbr->label);
                    copied[nbr] = new_node;
                    copied[cur]->neighbors.push_back(new_node);
                    q.push(nbr);
                }
            }
        }
        return copied[node];
    }
};
```

相关题目

- 无

第 15 章
细节实现题

这类题目不考特定的算法，纯粹考察写代码的熟练度。

15.1 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

```
//LintCode, Reverse Integer
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    int reverse (int x) {
        int r = 0;

        for (; x; x /= 10)
            r = r * 10 + x % 10;
```

```
        return r;
    }
};
```

相关题目

- Palindrome Number, 见 §15.2

15.2 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

```
//LintCode, Palindrome Number
// 时间复杂度 O(1), 空间复杂度 O(1)
class Solution {
public:
    bool 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, 见 §15.1
- Valid Palindrome, 见 §3.1

15.3 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) { }
};

//LintCode, Insert Interval
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {
            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            } else if (newInterval.start > it->end) {
                it++;
```

```
                continue;
            } else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

相关题目

- Merge Intervals，见 §15.4

15.4 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

```
struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) { }
    Interval(int s, int e) : start(s), end(e) { }
};

//LintCode, 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)
class Solution {
public:
    vector<Interval> merge(vector<Interval> &intervals) {
        vector<Interval> result;
        for (int i = 0; i < intervals.size(); i++) {
            insert(result, intervals[i]);
        }
        return result;
```

```
    }
private:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {
            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            } else if (newInterval.start > it->end) {
                it++;
                continue;
            } else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

相关题目

- Insert Interval，见 §15.3

15.5 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

```
// LintCode, 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);
    }
};
```

相关题目

- 无

15.6 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

```
// LintCode, Multiply Strings
// @author 连城 (http://weibo.com/lianchengzju)
// 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)
typedef vector<int> bigint;

bigint make_bigint(string const& repr) {
    bigint n;
    transform(repr.rbegin(), repr.rend(), back_inserter(n),
            [](char c) { return c - '0'; });
    return n;
}

string to_string(bigint const& n) {
    string str;
    transform(find_if(n.rbegin(), prev(n.rend()),
            [](char c) { return c > '\0'; }), n.rend(), back_inserter(str),
            [](char c) { return c + '0'; });
    return str;
}

bigint operator*(bigint const& x, bigint const& y) {
    bigint z(x.size() + y.size());

    for (size_t i = 0; i < x.size(); ++i)
        for (size_t j = 0; j < y.size(); ++j) {
            z[i + j] += x[i] * y[j];
            z[i + j + 1] += z[i + j] / 10;
            z[i + j] %= 10;
        }

    return z;
}
```

```
class Solution {
public:
    string multiply(string num1, string num2) {
        return to_string(make_bigint(num1) * make_bigint(num2));
    }
};
```

代码 2

```
// LintCode, Multiply Strings
// 9 个字符对应一个 int64_t
// 时间复杂度 O(n*m/81), 空间复杂度 O((n+m)/9)
/** 大整数类. */
class BigInt {
public:
    /**
     * @brief 构造函数, 将字符串转化为大整数.
     * @param[in] s 输入的字符串
     * @return 无
     */
    BigInt(string s) {
        vector<int64_t> result;
        result.reserve(s.size() / RADIX_LEN + 1);

        for (int i = s.size(); i > 0; i -= RADIX_LEN) {  // [i-RADIX_LEN, i)
            int temp = 0;
            const int low = max(i - RADIX_LEN, 0);
            for (int j = low; j < i; j++) {
                temp = temp * 10 + s[j] - '0';
            }
            result.push_back(temp);
        }
        elems = result;
    }
    /**
     * @brief 将整数转化为字符串.
     * @return 字符串
     */
    string toString() {
        stringstream result;
        bool started = false; // 用于跳过前导 0
        for (auto i = elems.rbegin(); i != elems.rend(); i++) {
            if (started) { // 如果多余的 0 已经都跳过, 则输出
                result << setw(RADIX_LEN) << setfill('0') << *i;
            } else {
                result << *i;
                started = true; // 碰到第一个非 0 的值, 就说明多余的 0 已经都跳过
            }
        }

        if (!started) return "0"; // 当 x 全为 0 时
        else return result.str();
```

```
    }

    /**
     * @brief 大整数乘法.
     * @param[in] x x
     * @param[in] y y
     * @return 大整数
     */
    static BigInt multiply(const BigInt &x, const BigInt &y) {
        vector<int64_t> z(x.elems.size() + y.elems.size(), 0);

        for (size_t i = 0; i < y.elems.size(); i++) {
            for (size_t j = 0; j < x.elems.size(); j++) { // 用 y[i] 去乘以 x 的各位
                //  两数第 i, j 位相乘, 累加到结果的第 i+j 位
                z[i + j] += y.elems[i] * x.elems[j];

                if (z[i + j] >= BIGINT_RADIX) { //  看是否要进位
                    z[i + j + 1] += z[i + j] / BIGINT_RADIX; //  进位
                    z[i + j] %= BIGINT_RADIX;
                }
            }
        }
        while (z.back() == 0) z.pop_back();  // 没有进位, 去掉最高位的 0
        return BigInt(z);
    }

private:
    typedef long long int64_t;
    /** 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
     */
    const static int BIGINT_RADIX = 1000000000;
    const static int RADIX_LEN = 9;
    /** 万进制整数. */
    vector<int64_t> elems;
    BigInt(const vector<int64_t> num) : elems(num) {}
};


class Solution {
public:
    string multiply(string num1, string num2) {
        BigInt x(num1);
        BigInt y(num2);
        return BigInt::multiply(x, y).toString();
    }
};
```

相关题目

- 无

15.7 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

```
// LintCode, Substring with Concatenation of All Words
// 时间复杂度 O(n*m), 空间复杂度 O(m)
class Solution {
public:
    vector<int> findSubstring(string s, vector<string>& dict) {
        size_t wordLength = dict.front().length();
        size_t catLength = wordLength * dict.size();
        vector<int> result;

        if (s.length() < catLength) return result;

        unordered_map<string, int> wordCount;

        for (auto const& word : dict) ++wordCount[word];

        for (auto i = begin(s); i <= prev(end(s), catLength); ++i) {
            unordered_map<string, int> unused(wordCount);

            for (auto j = i; j != next(i, catLength); j += wordLength) {
                auto pos = unused.find(string(j, next(j, wordLength)));

                if (pos == unused.end() || pos->second == 0) break;

                if (--pos->second == 0) unused.erase(pos);
            }

            if (unused.size() == 0) result.push_back(distance(begin(s), i));
        }

        return result;
    }
};
```

相关题目

- 无

15.8 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

```
// LintCode, Pascal's Triangle
// 时间复杂度 O(n^2), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > result;
        if(numRows == 0) return result;

        result.push_back(vector<int>(1,1)); //first row

        for(int i = 2; i <= numRows; ++i) {
            vector<int> current(i,1);  // 本行
            const vector<int> &prev = result[i-2];  // 上一行

            for(int j = 1; j < i - 1; ++j) {
                current[j] = prev[j-1] + prev[j]; // 左上角和右上角之和
            }
            result.push_back(current);
        }
```

```
        return result;
    }
};
```

从右到左

```
// LintCode, Pascal's Triangle
// 时间复杂度 O(n^2), 空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > result;
        vector<int> array;
        for (int i = 1; i <= numRows; i++) {
            for (int j = i - 2; j > 0; j--) {
                array[j] = array[j - 1] + array[j];
            }
            array.push_back(1);
            result.push_back(array);
        }
        return result;
    }
};
```

相关题目

- Pascal's Triangle II，见 §15.9

15.9 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

```
// LintCode, 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，见 §15.8

15.10 Spiral Matrix

描述

Given a matrix of $m \times n$ 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。

代码

```
// LintCode, Spiral Matrix
// @author 龚陆安 (http://weibo.com/luangong)
// 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int> >& matrix) {
        vector<int> result;
        if (matrix.empty()) return result;
        int beginX = 0, endX = matrix[0].size() - 1;
        int beginY = 0, endY = matrix.size() - 1;
        while (true) {
            // From left to right
```

```
            for (int j = beginX; j <= endX; ++j) result.push_back(matrix[beginY][j]);
            if (++beginY > endY) break;
            // From top to bottom
            for (int i = beginY; i <= endY; ++i) result.push_back(matrix[i][endX]);
            if (beginX > --endX) break;
            // From right to left
            for (int j = endX; j >= beginX; --j) result.push_back(matrix[endY][j]);
            if (beginY > --endY) break;
            // From bottom to top
            for (int i = endY; i >= beginY; --i) result.push_back(matrix[i][beginX]);
            if (++beginX > endX) break;
        }
        return result;
    }
};
```

相关题目

- Spiral Matrix II，见 §15.11

15.11 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

```
// LintCode, Spiral Matrix II
// 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));
        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

```
// LintCode, 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, 见 §15.10

15.12 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

```
// LintCode, ZigZag Conversion
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    string convert(string s, int nRows) {
        if (nRows <= 1 || s.size() <= 1) return s;
        string result;
        for (int i = 0; i < nRows; i++) {
            for (int j = 0, index = i; index < s.size();
                    j++, index = (2 * nRows - 2) * j + i) {
                result.append(1, s[index]);  // 垂直元素
```

```
                if (i == 0 || i == nRows - 1) continue;   // 斜对角元素
                if (index + (nRows - i - 1) * 2 < s.size())
                    result.append(1, s[index + (nRows - i - 1) * 2]);
            }
        }
        return result;
    }
};
```

相关题目

- 无

15.13 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。

代码 1

```
// LintCode, Divide Two Integers
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    int divide(int dividend, int divisor) {
        // 当 dividend = INT_MIN 时, -dividend 会溢出, 所以用 long long
        long long a = dividend >= 0 ? dividend : -(long long)dividend;
        long long b = divisor >= 0 ? divisor : -(long long)divisor;

        // 当 dividend = INT_MIN 时, divisor = -1 时, 结果会溢出, 所以用 long long
        long long result = 0;
        while (a >= b) {
            long long c = b;
            for (int i = 0; a >= c; ++i, c <<= 1) {
                a -= c;
                result += 1 << i;
            }
        }

        return ((dividend^divisor) >> 31) ? (-result) : (result);
```

```
    }
};
```

代码 2

```
// LintCode, Divide Two Integers
// 时间复杂度 O(logn), 空间复杂度 O(1)
class Solution {
public:
    int divide(int dividend, int divisor) {
        int result = 0; // 当 dividend = INT_MIN 时, divisor = -1 时, 结果会溢出
        const bool sign = (dividend > 0 && divisor < 0) ||
                (dividend < 0 && divisor > 0); // 异号

        // 当 dividend = INT_MIN 时, -dividend 会溢出, 所以用 unsigned int
        unsigned int a = dividend >= 0 ? dividend : -dividend;
        unsigned int b = divisor >= 0 ? divisor : -divisor;

        while (a >= b) {
            int multi = 1;
            unsigned int bb = b;
            while (a >= bb) {
                a -= bb;
                result += multi;

                if (bb < INT_MAX >> 1) { // 防止溢出
                    bb += bb;
                    multi += multi;
                }
            }
        }
        if (sign) return -result;
        else return result;
    }
};
```

相关题目

- 无

15.14 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

```
// LintCode, Text Justification
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<string> fullJustify(vector<string> &words, int L) {
        vector<string> result;
        const int n = words.size();
        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].size() + (i - begin) > L) {
                result.push_back(connect(words, begin, i - 1, len, L, false));
                begin = i;
                len = 0;
            }
            len += words[i].size();
        }
        // 最后一行不足 L
        result.push_back(connect(words, begin, n - 1, len, L, 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
     */
    string connect(vector<string> &words, int begin, int end,
            int len, int L, bool is_last) {
        string s;
        int n = end - begin + 1;
        for (int i = 0; i < n; ++i) {
            s += words[begin + i];
            addSpaces(s, i, n - 1, L - len, is_last);
        }

        if (s.size() < L) s.append(L - s.size(), ' ');
        return s;
    }

    /**
     * @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
     */
    void addSpaces(string &s, int i, int n, int L, bool is_last) {
        if (n < 1 || i > n - 1) return;
        int spaces = is_last ? 1 : (L / n + (i < (L % n) ? 1 : 0));
        s.append(spaces, ' ');
    }
};
```

相关题目

- 无

15.15 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

```
// LintCode, 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)
class Solution {
public:
    int maxPoints(vector<Point> &points) {
        if (points.size() < 3) return points.size();
        int result = 0;

        for (int i = 0; i < points.size() - 1; i++) {
            for (int j = i + 1; j < points.size(); j++) {
                int sign = 0;
                int a, b, c;
                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.size(); 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

```
// LintCode, 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)
class Solution {
public:
```

```
    int maxPoints(vector<Point> &points) {
        if (points.size() < 3) return points.size();
        int result = 0;

        unordered_map<double, int> slope_count;
        for (int i = 0; i < points.size()-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.size(); j++) {
                double slope; // 斜率
                if (points[i].x == points[j].x) {
                    slope = std::numeric_limits<double>::infinity();
                    if (points[i].y == points[j].y) {
                        ++ samePointNum;
                        continue;
                    }
                } else {
                    slope = 1.0 * (points[i].y - points[j].y) /
                        (points[i].x - points[j].x);
                }

                int count = 0;
                if (slope_count.find(slope) != slope_count.end())
                    count = ++slope_count[slope];
                else {
                    count = 2;
                    slope_count[slope] = 2;
                }

                if (point_max < count) point_max = count;
            }
            result = max(result, point_max + samePointNum);
        }
        return result;
    }
};
```

相关题目

- 无