LeetCode 题解

戴方勤 (soulmachine@gmail.com)

https://gitcafe.com/soulmachine/LeetCode

最后更新 2013-10-4

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 LeetCode Online Judge(http://leetcode.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 LeetCode 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。

GitCafe 地址

本书是开源的，项目地址：https://gitcafe.com/soulmachine/LeetCode

[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)` 是否小于某个阀值，例如 `1e-9`。

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

代码

```
// LeetCode, Remove Duplicates from Sorted Array
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        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;
    }
};
```

```
// LeetCode, Remove Duplicates from Sorted Array, 使用 STL
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        return removeDuplicates(A, A + n, A) - A;
    }

    template<typename InIt, typename OutIt>
    OutIt removeDuplicates(InIt first, InIt last, OutIt output) {
        while (first != last) {
            *output++ = *first;
            first = find_if(first, last,
                    bind1st(not_equal_to<int>(), *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

```
// LeetCode, Remove Duplicates from Sorted Array II
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        if (n == 0) return 0;

        int occur = 1;
        int index = 0;
        for (int i = 1; i < n; i++) {
            if (A[index] == A[i]) {
```

```
                if (occur < 2) {
                    A[++index] = A[i];
                    occur++;
                }
            } else {
                A[++index] = A[i];
                occur = 1;
            }
        }
        return index + 1;
    }
};
```

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

```
// LeetCode, Search in Rotated Sorted Array
class Solution {
public:
    int search(int A[], int n, int target) {
        int first = 0, last = n;
        while (first != last) {
            const int mid = (first + last) / 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

```
// LeetCode, Search in Rotated Sorted Array II
class Solution {
public:
    bool search(int A[], int n, int target) {
        int first = 0, last = n;
        while (first != last) {
            const int mid = (first + last) / 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, A[start] == A[mid]
                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

```
// LeetCode, Median of Two Sorted Arrays
class Solution {
public:
    double findMedianSortedArrays(int A[], int m, int B[], int n) {
        int total = m + n;
        if (total & 0x1)
            return find_kth(A, m, B, n, total / 2 + 1);
        else
            return (find_kth(A, m, B, n, total / 2)
                    + find_kth(A, m, B, n, total / 2 + 1)) / 2;
    }
private:
    static double find_kth(int A[], int m, int B[], int n, int k) {
        //always assume that m is equal or smaller than n
        if (m > n) return find_kth(B, n, A, m, k);
        if (m == 0) return B[k - 1];
        if (k == 1) return min(A[0], B[0]);

        //divide k into two parts
        int pa = min(k / 2, m), pb = k - pa;
        if (A[pa - 1] < B[pb - 1])
            return find_kth(A + pa, m - pa, B, n, k - pa);
        else if (A[pa - 1] > B[pb - 1])
            return find_kth(A, m, B + pb, n - pb, k - pb);
        else
            return A[pa - 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
class Solution {
public:
    int longestConsecutive(vector<int> const& 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;
```

```
    }
};
```

相关题目

- 无

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

```
//LeetCode, Add Two Numbers
//跟 Add Binary 很类似
class Solution {
public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        ListNode head(-1); // 头节点
        int carry = 0;
        ListNode *prev = &head;
        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 head.next;
    }
};
```

相关题目

- Add Binary，见 §3.4

2.2.2 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

```
// LeetCode, Reverse Linked List II
class Solution {
public:
    ListNode *reverseBetween(ListNode *head, int m, int n) {
        if(m >= n) return head;
        ListNode dummy(0);
        ListNode *h = &dummy;
        h->next = head;

        int count = 0;
        ListNode *p = h, *pm, *pn;
        for (; p; p = p->next) {
            if (count == m-1) pm = p;
            if (count == n) {
                pn = p;
                break;
            }
            count++;
```

```
        }
        p = pm;
        pm = pm->next;
        if (m == 1) head = pn;  // 若 m=1, 则 pn 就变为首节点

        p->next = pn;
        p = pm->next;
        pm->next = pn->next;

        ListNode *q = p->next; // pm->p->q
        while(pm != pn) {
            p->next = pm;
            pm = p;
            p = q;
            if(q) q = q->next;
        }

        return head;
    }
};
```

相关题目

- 无

2.2.3 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

```
// LeetCode, Partition List
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        if (head == nullptr) return head;

        ListNode left_dummy(0); // 头结点
        ListNode right_dummy(0); // 头结点
```

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

        for (; head; head = head->next) {
            if (head->val < x) {
                left_cur->next = head;
                left_cur = head;
            } else {
                right_cur->next = head;
                right_cur = head;
            }
        }

        left_cur->next = right_dummy.next;
        right_cur->next = nullptr;

        return left_dummy.next;
    }
};
```

相关题目

- 无

2.2.4 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

代码

```
// LeetCode, Remove Duplicates from Sorted List
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return nullptr;
        ListNode * prev = head;
        ListNode *cur = head->next;
        while (cur != nullptr) {
```

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

相关题目

- Remove Duplicates from Sorted List II，见 §2.2.5

2.2.5 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

代码

```
// LeetCode, Remove Duplicates from Sorted List II
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 §2.2.4

2.2.6 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

```
// LeetCode, Remove Rotate List
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

2.2.7 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

```
// LeetCode, Remove Nth Node From End of List
class Solution {
public:
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        ListNode dummy(0);
        dummy.next = head;
        ListNode *p = &dummy, *q = &dummy;

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

        while(q->next) { // 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

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
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;
                continue;
            }
            if (!::isalnum(*right)) {
                --right;
                continue;
            }
            if (*left == *right) {
```

```
                ++left;
                --right;
            } else {
                return false;
            }
        }
        return true;
    }
};
```

相关题目

- 无

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。

代码

```
// LeetCode, Implement strStr()
// 暴力解法，复杂度 O(N*M)
class Solution {
public:
    char *strStr(const char *haystack, const char *needle) {
        // if needle is empty return the full string
        if (!*needle) return (char*) haystack;

        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;

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

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

```
// LeetCode, String to Integer (atoi)
class Solution {
public:
    int atoi(const char *str) {
        int num = 0;
        int sign = 1;
        const int len = strlen(str);
        int i = 0;

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

        if (str[i] == '+') i++;

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

        for (; i < len; 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

```
//LeetCode, Add Binary
class Solution {
public:
    string addBinary(string a, string b) {
        string result;
        const size_t max_len = 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 < max_len; 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://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html 。

代码

```
// LeetCode, Longest Palindromic Substring
// 备忘录法, TLE
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);
    }
};
```

```
// LeetCode, Longest Palindromic Substring
// 动规
class Solution {
public:
    string longestPalindrome(string s) {
        const int len = s.size();
        int f[len][len];
        memset(f, 0, len * len * sizeof(int)); //TODO: fill, fill_n
        int maxL = 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] = 1;
            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] && maxL < (i - j + 1)) {
                    maxL = i - j + 1;
                    start = j;
                }
            }
        }
        return s.substr(start, maxL);
    }
};
```

```
// LeetCode, Longest Palindromic Substring
// Manacher' s Algorithm
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);
        int n = T.length();
        // 以 T[i] 为中心, 向左/右扩张的长度, 不包含 T[i] 自己,
        // 因此 P[i] 是源字符串中回文串的长度
        int *P = new int[n];
        int C = 0, R = 0;

        for (int i = 1; i < n - 1; i++) {
            int 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 maxLen = 0;
        int centerIndex = 0;
        for (int i = 1; i < n - 1; i++) {
            if (P[i] > maxLen) {
                maxLen = P[i];
                centerIndex = i;
            }
        }
        delete[] P;

        return s.substr((centerIndex - 1 - maxLen) / 2, maxLen);
    }
};
```

相关题目

- 无

第 4 章
树

LeetCode 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) { }
};
```

4.1 二叉树的遍历

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

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

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

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

4.1.1 Binary Tree 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

代码

```
//LeetCode, Binary Tree Level Order Traversal
class Solution {
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int> > result;
        if(root == nullptr) return result;

        queue<TreeNode*> current_level, next_level;
        vector<int> level; // elments in level level

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

相关题目

- Binary Tree Level Order Traversal II，见 §4.1.2
- Binary Tree Zigzag Level Order Traversal，见 §4.1.3

4.1.2 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 Binary Tree Level Order Traversal I（见 §4.1.1）的基础上，用一个 list 作为栈，每次在头部插入，就可以实现倒着输出

代码

```
//LeetCode, Binary Tree Level Order Traversal II
//在 Binary Tree Level Order Traversal I 的基础上，用一个 list 作为栈
//每次在头部插入，就可以实现倒着输出
class Solution {
public:
    vector<vector<int> > levelOrderBottom(TreeNode *root) {
        list<vector<int> > tmp_result;

        queue<TreeNode *> q;
        q.push(root);
        q.push(nullptr); // level separator

        vector<int> level;  // elements in one level
        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(level.size() > 0) {
                    tmp_result.push_front(level);
```

```
                    level.clear();
                    q.push(nullptr);
                }
            }
        }

        vector<vector<int> > ret;
        for (auto e : tmp_result) {
            ret.push_back(e);
        }
        return ret;
    }
};
```

相关题目

- Binary Tree Level Order Traversal，见 §4.1.1
- Binary Tree Zigzag Level Order Traversal，见 §4.1.3

4.1.3 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。

代码

```
//LeetCode, 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。
class Solution {
```

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

        queue<TreeNode*> q;
        bool l2r = 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 (l2r) {
                    result.push_back(level);
                } else {
                    vector<int> tmp;
                    for (auto iter = level.rbegin(); iter != level.rend(); ++iter) {
                        tmp.push_back(*iter);
                    }
                    result.push_back(tmp);
                }
                level.clear();
                l2r = !l2r;

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

        return result;
    }
};
```

相关题目

- Binary Tree Level Order Traversal，见 §4.1.1
- Binary Tree Level Order Traversal II，见 §4.1.2

4.1.4 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

不用递归，可用栈，Morris 中序遍历或者线索二叉树哦。

代码

栈

```
// LeetCode, Binary Tree Inorder Traversal
// 使用栈
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

```
// LeetCode, Binary Tree Inorder Traversal
// Morris 中序遍历
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode* prev = nullptr;
        TreeNode* cur = root;

        while (cur != nullptr) {
```

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

相关题目

- Recover Binary Search Tree，见 §4.1.5

4.1.5 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

```
// LeetCode, Recover Binary Search Tree
// Morris 中序遍历
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 §4.1.4

4.1.6 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

代码

递归版

```
// LeetCode, Same Tree
// 迭代
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        if (!p && !q) return true;   // 终止条件
        if (!p || !q) return false;  // 剪纸
        return p->val == q->val      // 三方合并
                && isSameTree(p->left, q->left)
                && isSameTree(p->right, q->right);
    }
};
```

迭代版

```
// LeetCode, Same Tree
// 迭代
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 §4.1.7

4.1.7 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

代码

递归版

```
// LeetCode, Symmetric Tree
// 递归版
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

```
// LeetCode, Symmetric Tree
// 迭代版
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 §4.1.6

4.1.8 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

```
// LeetCode, Balanced Binary Tree
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 lhs = balancedHeight (root->left);
        int rhs = balancedHeight (root->right);

        if (lhs < 0 || rhs < 0 || abs(lhs - rhs) > 1) return -1;  // 剪枝

        return max(lhs, rhs) + 1; // 三方合并
    }
};
```

相关题目

- 无

4.1.9 Flatten Binary Tree to Linked List

描述

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

For example, Given

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

The flattened tree should look like:

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

分析

无

代码

递归版

```
// LeetCode, Flatten Binary Tree to Linked List
// 递归版 1, 作者: 王顺达, http://weibo.com/u/1234984145
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;
    }
};
```

```
// LeetCode, Flatten Binary Tree to Linked List
// 递归版 2
class Solution {
public:
    void flatten(TreeNode *root) {
        if (root == nullptr) return;  // 终止条件
        //1.flat the left subtree
        if (root->left) flatten(root->left);

        //2.flatten the right subtree
        if (root->right) flatten(root->right);

        //3.if no left return
        if (nullptr == root->left) return;

        //4.insert left sub tree between root and root->right
        //4.1.find the last node in left
        TreeNode *p = root->left;
        while(p->right) p = p->right;
        p->right = root->right;
        //4.2.connect right sub tree after left sub tree
        p->right = root->right;
        //4.3.move left sub tree to the root's right sub tree
        root->right = root->left;
        root->left = nullptr;
    }
};
```

迭代版

```
// LeetCode, Flatten Binary Tree to Linked List
// 迭代版
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

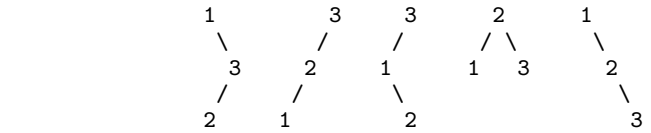
4.2 二叉查找树

4.2.1 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。

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

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

当数组为 $1, 2, 3, ..., n$ 时，基于以下原则的构建的 BST 树具有唯一性：**以 i 为根节点的树，其左子树由 [1, i-1] 构成，其右子树由 [i+1, n] 构成。**

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

如果数组为空，毫无疑问，只有一种 BST，即空树，$f(0) = 1$。

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

如果数组有两个元素 1,2，那么有如下两种可能

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

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

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

```
// LeetCode, Unique Binary Search Trees
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

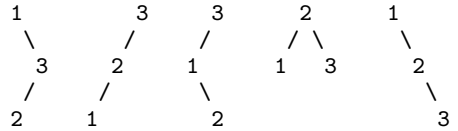
- Unique Binary Search Trees II，见 §4.2.2

4.2.2 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.



分析

见前面一题。

代码

```
// LeetCode, Unique Binary Search Trees II
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 §4.2.1

4.2.3 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

```
// LeetCode, Validate Binary Search Tree
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 §4.2.3

4.2.4 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

```
// LeetCode, Convert Sorted Array to Binary Search Tree
// 分治法
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
        if (length == 1) return new TreeNode(*first); // 收敛条件

        // 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 §4.2.5

4.2.5 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://leetcode.com/2010/11/convert-sorted-list-to-balanced-binary.html

代码

分治法，类似于 Convert Sorted Array to Binary Search Tree，自顶向下，复杂度 $O(n \log n)$。

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

```
// LeetCode, Convert Sorted List to Binary Search Tree
// bottom-up, 复杂度 O(nlogn)
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 §4.2.4

4.3 二叉树的深度

4.3.1 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

代码

递归版

```
// LeetCode, Minimum Depth of Binary Tree
// 递归版
class Solution {
public:
    int minDepth(TreeNode *root) {
        if (root == nullptr) return 0;
        else {
            d = INT_MAX;
            minDepth(root, 1);
            return d;
        }
    }
private:
    int d;
    void minDepth(TreeNode *root, int cur) {
        if (root != nullptr) {
            if (root->left == nullptr && root->right == nullptr) { // 叶子节点
                d = min(cur, d);
                return;
            }
            if (cur < d) { // 剪枝
                minDepth(root->left, cur+1);
                minDepth(root->right, cur+1);
            }
        }
    }
};
```

迭代版

```
// LeetCode, Minimum Depth of Binary Tree
// 迭代版
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 §4.3.2

4.3.2 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

```
// LeetCode, Maximum Depth of Binary Tree
class Solution {
public:
    int maxDepth(TreeNode *root) {
        if (root == nullptr) return 0;

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

相关题目

- Minimum Depth of Binary Tree，见 §4.3.1

4.4 二叉树的构建

4.4.1 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

```
// LeetCode, Construct Binary Tree from Preorder and Inorder Traversal
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 §4.4.2

4.4.2 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

```
// LeetCode, Construct Binary Tree from Inorder and Postorder Traversal
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 §4.4.1

4.5 二叉树的 DFS

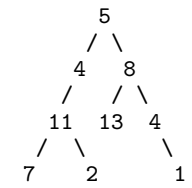
二叉树的先序、中序、后序遍历都可以看做是 DFS，此外还有其他遍历顺序，共有 $3! = 6$ 种。其他 3 种顺序是 `root->r->l`，`r->root->l`，`r->l->root`。

4.5.1 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

```
// LeetCode, Path Sum
class Solution {
public:
    bool hasPathSum(TreeNode *root, int sum) {
        if (root == nullptr) return false;

        if (root->left == nullptr && root->right == nullptr) { // leaf
            if (sum == root->val) return true;
            else return false;
        }
        if (hasPathSum(root->left, sum - root->val)) return true;
        if (hasPathSum(root->right, sum - root->val)) return true;
    }
};
```

相关题目

- Path Sum II，见 §4.5.2

4.5.2 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`,

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

```
// LeetCode, Path Sum II
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 §4.5.1

4.5.3 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 §11.2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

```
// LeetCode, Binary Tree Maximum Path Sum
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 §11.2

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

代码

```
// LeetCode, Populating Next Right Pointers in Each Node
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 §4.7

4.7 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，因此用广搜。

同时注意，这题的代码原封不动，可以解决上一题！

代码

```
// LeetCode, Populating Next Right Pointers in Each Node II
class Solution {
public:
    void connect(TreeLinkNode *root) {
        vector<vector<int> > result;
        if (root == nullptr)
            return;

        vector<TreeLinkNode*> nextLevel, currentLevel;
        currentLevel.push_back(root);
        while (!currentLevel.empty()) {
            while (!currentLevel.empty()) {
                TreeLinkNode* node = currentLevel.front();
                currentLevel.erase(currentLevel.begin());

                if (!currentLevel.empty())
                    node->next = currentLevel.front();
                else
                    node->next = nullptr;

                if (node->left != nullptr)
                    nextLevel.push_back(node->left);
                if (node->right != nullptr)
                    nextLevel.push_back(node->right);
            }
            swap(nextLevel, currentLevel); //!!! how to use swap
        }
    }
};
```

相关题目

- Populating Next Right Pointers in Each Node，见 §4.6

第 5 章
排序

5.1 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

```
//LeetCode, Merge Sorted Array
class Solution {
public:
    void merge(int A[], int m, int B[], int n) {
        int ia = m - 1, ib = n - 1, icur = m + n - 1;
        while(ia >= 0 && ib >= 0) {
            A[icur--] = A[ia] >= B[ib] ? A[ia--] : B[ib--];
        }
        while(ib >= 0) {
            A[icur--] = B[ib--];
        }
    }
};
```

相关题目

- Merge Two Sorted Lists，见 §5.2
- Merge k Sorted Lists，见 §5.3

5.2 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

```
//LeetCode, Merge Two Sorted Lists
class Solution {
public:
    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 §5.1
- Merge k Sorted Lists，见 §5.3

5.3 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 §5.2）的函数

代码

```
//LeetCode, Merge k Sorted Lists
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 §5.1
- Merge Two Sorted Lists，见 §5.2

第 6 章
暴力枚举法

6.1 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],
  []
]
```

6.1.1 增量构造法

每个元素，都有两种选择，选或者不选。

代码

```
// LeetCode, Subsets
// 增量构造法，朴素深搜
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        vector<vector<int> > result;
        vector<int> cur;
        sort(S.begin(), S.end()); // 本题对顺序有要求，需要排序
```

```
        subsets(S, cur, 0, result);
        return result;
    }

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

6.1.2 位向量法

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

代码

```
// LeetCode, Subsets
// 位向量法，也属于朴素深搜
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        vector<vector<int> > result;
        vector<bool> selected(S.size(), false);
        sort(S.begin(), S.end()); // 本题对顺序有要求，需要排序

        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);
        }
};
```

6.1.3 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 $B_1|B_2, B_1\&B_2, B_1B_2$ 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。

代码

```
// LeetCode, Subsets
// 二进制法
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        vector<vector<int> > result;
        sort(S.begin(), S.end()); // 本题对顺序有要求，需要排序
        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 §6.2

6.2 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。

代码

```
// LeetCode, Subsets II
// 增量构造法
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > result;
        sort(S.begin(), S.end()); // 本题对顺序有要求，需要排序

        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();
            }
        }
    }
};

// LeetCode, Subsets II
// 位向量法
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > result;
        sort(S.begin(), S.end()); // 本题对顺序有要求，需要排序
        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);
        }
    }
};
```

相关题目

- Subsets，见 §6.1

6.3 Permutations

6.3.1 next_permutation()

偷懒的做法，可以直接使用 `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。

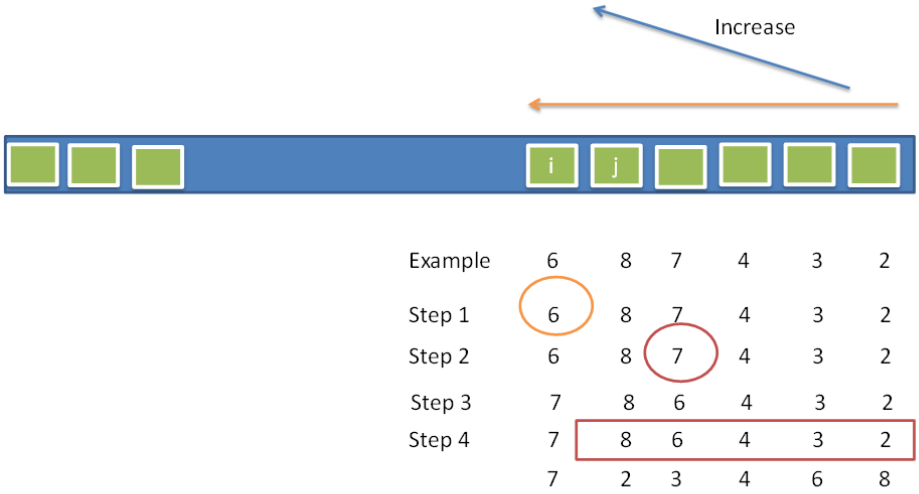
代码

```
// LeetCode, Permutations
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;
    }
};
```

6.3.2 重新实现 next_permutation()

算法过程如图 6-1所示（来自 http://fisherlei.blogspot.com/2012/12/leetcode-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 6-1 下一个排列算法流程

代码

```
// LeetCode, Permutations
// 重新实现 next_permutation()
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()));

        return permutations;
    }

    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 and !(*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;
    }
};
```

6.3.3 深搜

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

```
// LeetCode, Permutations
// 深搜
class Solution {
public:
    vector<vector<int> > permute(vector<int>& num) {
        sort(num.begin(), num.end());

        vector<vector<int>> result;
        vector<int> p(num.size(), 0);  // 中间结果

        permute(num.begin(), num.end(), 0, p, result);
        return result;
    }
```

```
private:
    typedef vector<int>::const_iterator Iter;
    void permute(Iter first, Iter last, int cur, vector<int> &p,
            vector<vector<int> > &result) {
        if ((first + cur) == last) {  // 收敛条件
            result.push_back(p);
        }

        // 扩展状态
        for (auto i = first; i != last; i++) {
            bool used = false;
            // 查找 *i 是否在 p[0, cur) 中出现过
            for (auto j = p.begin(); j != p.begin() + cur; j++) {
                if (*i == *j) {
                    used = true;
                    break;
                }
            }
            if (!used) {
                p[cur] = *i;
                permute(first, last, cur + 1, p, result);
                // 不需要恢复 P[cur], 返回上层 cur 会自动减 1, P[cur] 会被覆盖
            }
        }
    }
};
```

相关题目

- Permutations II，见 §6.4

6.4 Permutations II

6.4.1 next_permutation()

上一题中的代码，见 §6.3.1 节，也适用于本题。

6.4.2 重新实现 next_permutation()

上一题中的代码，见 §6.3.2 节，也适用于本题。

6.4.3 深搜

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

```
// LeetCode, Permutations II
// 深搜
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

- Permutations ，见 §6.3

第 7 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。

7.1 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

```
//LeetCode, Word Ladder
class Solution {
```

```
public:
    typedef string state_t;
    int ladderLength(string start, string end,
            const unordered_set<string> &dict) {
        if (start.size() != end.size()) return 0;
        if (start.empty() || end.empty()) return 0;

        queue<string> next, current; // 当前层，下一层
        unordered_set<string> visited; // 判重
        unordered_map<string, string > father;
        int level = 0;  // 层次
        bool found = false;

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

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

                        swap(c, new_word[i]);
                        if (new_word == end) {
                            found = true; //找到了
                            father[new_word] = str;
                            break;
                        }

                        if (dict.count(new_word) > 0
                                && !visited.count(new_word)) {
                            next.push(new_word);
                            visited.insert(new_word);
                            father[new_word] = str;
                        }
                        swap(c, new_word[i]); // 恢复该单词
                    }
                }
            }
            swap(next, current); //!!! 交换两个队列
        }
        if (found) return level+1;
        else return 0;
    }
};
```

相关题目

- Word Ladder II，见 §7.2

7.2 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

```
//LeetCode, Word Ladder II
class Solution {
public:
    vector<vector<string> > findLadders(string start, string end,
            const unordered_set<string> &dict) {
        unordered_set<string> visited; // 判重
        unordered_map<string, vector<string> > father; // 树
        unordered_set<string> current, next;  // 当前层，下一层，用集合是为了去重

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

        current.insert(start);
        while (!current.empty() && !found) {
            ++level;
```

```
            // 先将本层全部置为已访问，防止同层之间互相指向
            for (auto word : current)
                visited.insert(word);
            for (auto word : current) {
                for (size_t i = 0; i < word.size(); ++i) {
                    string new_word = word;
                    for (char c = 'a'; c <= 'z'; ++c) {
                        if (c == new_word[i]) continue;
                        swap(c, new_word[i]);

                        if (new_word == end) found = true; //找到了

                        if (visited.count(new_word) == 0
                                && (dict.count(new_word) > 0 ||
                                        new_word == end)) {
                            next.insert(new_word);
                            father[new_word].push_back(word);
                            // visited.insert(new_word) 移动到最上面了
                        }

                        swap(c, new_word[i]);  // restore
                    }
                }
            }

            current.clear();
            swap(current, next);
        }
        vector<vector<string> > result;
        if (found) {
            vector<string> path;
            buildPath(father, path, start, end, result);
        }
        return result;
    }
private:
    void buildPath(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 (auto f : father[word]) {
                buildPath(father, path, start, f, result);
            }
        }
        path.pop_back();
    }
};
```

相关题目

- Word Ladder，见 §12.1

第 8 章
深度优先搜索

8.1 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，前后两个隔板默认已经使用，因此复杂度为 $O(2^{n-1})$

代码

```
//LeetCode, Palindrome Partitioning
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> output;  // 一个 partition 方案
        DFS(s, 0, 1, output, result);
        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, size_t prev, size_t start, vector<string>& output,
            vector<vector<string>> &result) {
        if (start == s.size()) { // 最后一个隔板
            if (isPalindrome(s, prev, start - 1)) { // 必须使用
```

```
                output.push_back(s.substr(prev, start - prev));
                result.push_back(output);
                output.pop_back();
            }
            return;
        }
        // 不断开
        DFS(s, prev, start + 1, output, result);
        // 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)) {
            // 不断开，if 上一行已经做了
            // 断开
            output.push_back(s.substr(prev, start - prev));
            DFS(s, start, start + 1, output, result);
            output.pop_back();
        }
    }

    bool isPalindrome(string &s, int start, int end) {
        while (start < end) {
            if (s[start++] != s[end--]) return false;
        }
        return true;
    }
};
```

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

```
//LeetCode, Palindrome Partitioning
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> output;  // 一个 partition 方案
        DFS(s, 0, output, result);
        return result;
    }
    // 搜索必须以 s[start] 开头的 partition 方案
    void DFS(string &s, int start, vector<string>& output,
            vector<vector<string>> &result) {
        if (start == s.size()) {
            result.push_back(output);
            return;
        }
        for (int i = start; i < s.size(); i++) {
            if (isPalindrome(s, start, i)) { // 从 i 位置砍一刀
                output.push_back(s.substr(start, i - start + 1));
                DFS(s, i + 1, output, result);  // 继续往下砍
                output.pop_back(); // 撤销上一个 push_back 的砍
            }
        }
    }
    bool isPalindrome(string &s, int start, int end) {
        while (start < end) {
```

```
            if (s[start] != s[end]) return false;
            start++;
            end--;
        }
        return true;
    }
};
```

相关题目

- Palindrome Partitioning II，见 §11.3

8.2 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?

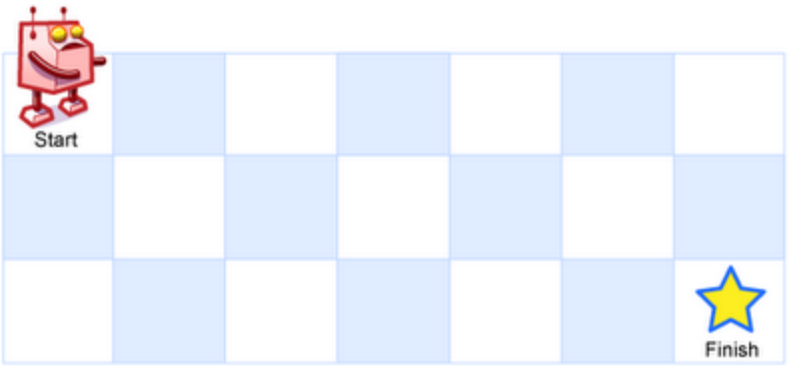


图 8-1 Above is a 3×7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

8.2.1 深搜

深搜，小集合可以过，大集合会超时

代码

```
// LeetCode, Unique Paths
// 深搜，小集合可以过，大集合会超时
```

```
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);
    }
};
```

8.2.2 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

```
// LeetCode, Unique Paths
// 深搜 + 缓存，即备忘录法
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);
    }
};
```

8.2.3 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

```
// LeetCode, Unique Paths
// 动规, 滚动数组
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];
    }
};
```

8.2.4 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

```
// LeetCode, Unique Paths
// 数学公式
class Solution {
public:
    typedef long long int64_t;
    // 求阶乘, n!/(start-1)!, 即 n*(n-1)...start, 要求 n >= 1
    static int64_t factor(int n, int start = 1) {
        int64_t  ret = 1;
        for(int i = start; i <= n; ++i)
            ret *= i;
        return ret;
    }
    // 求组合数 C_n^k
    static int64_t combination(int n, int k) {
        // 常数优化
        if (k == 0) return 1;
        if (k == 1) return n;

        int64_t ret = factor(n, k+1);
        ret /= factor(n - k);
        return ret;
    }
```

```
    int uniquePaths(int m, int n) {
        // max 可以防止 n 和 k 差距过大，从而防止 combination() 溢出
        return combination(m+n-2, max(m-1, n-1));
    }
};
```

相关题目

- Unique Paths II，见 §8.3

8.3 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.

8.3.1 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

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

```
    vector<vector<int> > f;  // 缓存

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

        // (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);
    }
};
```

8.3.2 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

```
// LeetCode, Unique Paths II
// 动规，滚动数组
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);

        // 寻找第一列的第一个障碍在哪一行
        int first_col_obstacle = INT_MAX;
        for (int i = 0; i < m; i++) {
            if (obstacleGrid[i][0]) {
                first_col_obstacle = i;
                break;
            }
        }

        for (int i = 0; i < m; i++) {
            // 第一列如果某一行有障碍物，那么后面的行应该为 0。
```

```
            if(i >= first_col_obstacle) f[0] = 0;
            else f[0] = 1;
            for (int j = 1; j < n; j++) {
                if (!obstacleGrid[i][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];
                } else {
                    f[j] = 0;
                }
            }
        }
        return f[n - 1];
    }
};
```

相关题目

- Unique Paths，见 §8.2

8.4 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.

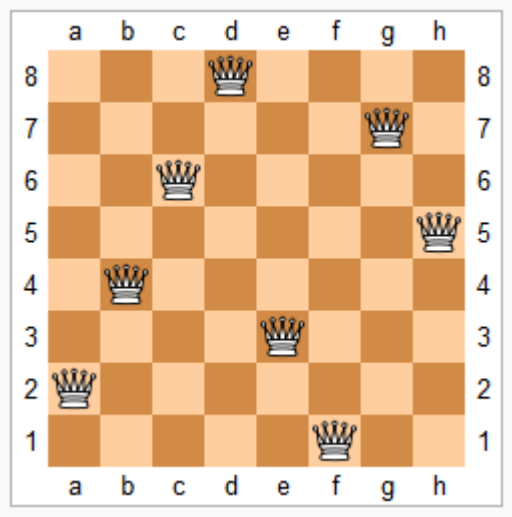


图 8-2 Eight Queens

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

```
// LeetCode, N-Queens
// 深搜 + 剪枝
class Solution {
public:
    vector<vector<string> > solveNQueens(int n) {
        this->columns = vector<int>(n, 0);
        this->principal_diagonals = vector<int>(2 * n, 0);
        this->counter_diagonals = 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(0, C, result);
        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> principal_diagonals;  // 占据了哪些主对角线
    vector<int> counter_diagonals;  // 占据了哪些副对角线

    void dfs(int row, vector<int> &C,
            vector<vector<string> > &result) {
        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 &&
                    principal_diagonals[row + j] == 0
                    && counter_diagonals[row - j + N] == 0;
            if (ok) {  // 剪枝：如果合法，继续递归
                // 执行扩展动作
                C[row] = j;
                columns[j] = principal_diagonals[row + j] =
                        counter_diagonals[row - j + N] = 1;
                dfs(row + 1, C, result);
                // 撤销动作
                // C[row] = 0;
                columns[j] = principal_diagonals[row + j] =
                        counter_diagonals[row - j + N] = 0;
            }
        }
    }
};
```

相关题目

- N-Queens II，见 §8.5

8.5 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

```
// LeetCode, N-Queens II
// 深搜 + 剪枝
class Solution {
```

```
public:
    int totalNQueens(int n) {
        this->count = 0;
        this->columns = vector<int>(n, 0);
        this->principal_diagonals = vector<int>(2 * n, 0);
        this->counter_diagonals = 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(0, C);
        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> principal_diagonals;  // 占据了哪些主对角线
    vector<int> counter_diagonals;  // 占据了哪些副对角线

    void dfs(int row, vector<int> &C) {
        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 &&
                    principal_diagonals[row + j] == 0
                    && counter_diagonals[row - j + N] == 0;
            if (ok) {  // 剪枝：如果合法，继续递归
                // 执行扩展动作
                C[row] = j;
                columns[j] = principal_diagonals[row + j] =
                        counter_diagonals[row - j + N] = 1;
                dfs(row + 1, C);
                // 撤销动作
                // C[row] = 0;
                columns[j] = principal_diagonals[row + j] =
                        counter_diagonals[row - j + N] = 0;
            }
        }
    }
};
```

相关题目

- N-Queens，见 §8.4

8.6 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

```
// LeetCode, Restore IP Addresses
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

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

```
// LeetCode, Combination Sum
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 §8.8

8.8 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

```
// LeetCode, Combination Sum II
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 §8.7

第 9 章
分治法

9.1 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

```
//LeetCode, Pow(x, n)
// 二分法, $x^n = x^{n/2} * x^{n/2} * x^{n\%2}$
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 §9.2

9.2 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

```
// LeetCode, Longest Substring Without Repeating Characters
// 二分查找
class Solution {
public:
    int sqrt(int x) {
        int left = 1, right = x / 2;
        int mid;
        int last_mid;  // 记录最近一次 mid

        if (x < 2) return x;

        while(left <= right) {
            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 §9.1

第 10 章
贪心法

10.1 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

```
// LeetCode, Jump Game
// 思路一
class Solution {
public:
    bool canJump(int A[], int n) {
        int right_most = 0; // 最右能跳到哪里
        for (int start = 0; start <= right_most; start++) {
            if (A[start] + start > right_most)
```

```
                right_most = A[start] + start;

            if (right_most >= n - 1) return true;
        }
        return false;
    }
};
```

```
// LeetCode, Jump Game
// 思路二
class Solution {
public:
    bool canJump (int A[], int n) {
        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;
    }
};
```

```
// LeetCode, Jump Game
// 思路三, 动规
class Solution {
public:
    bool canJump(int A[], int n) {
        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 §10.2

10.2 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

```
// LeetCode, Jump Game II
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;
    }
};
```

相关题目

- Jump Game，见 §10.1

10.3 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

```
// LeetCode, Best Time to Buy and Sell Stock
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 §10.4
- Best Time to Buy and Sell Stock III，见 §11.5

10.4 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

```
// LeetCode, Best Time to Buy and Sell Stock II
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 §10.3
- Best Time to Buy and Sell Stock III，见 §11.5

10.5 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 10-1所示。

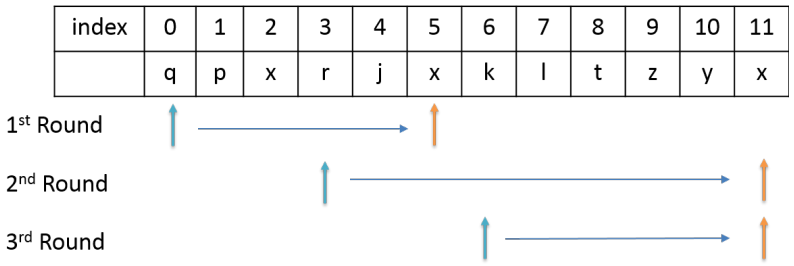


图 10-1 不含重复字符的最长子串

代码

```
// LeetCode, Longest Substring Without Repeating Characters
// 贪心法
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
        fill(last, last + ASCII_MAX, -1); // 0 也是有效位置, 因此初始化为-1
        int len = 0, max_len = 0;
        for (size_t i = 0; i < s.size(); i++, len++) {
            if (last[s[i] - 'a'] >= 0) {
                max_len = max(len, max_len);
                len = 0;
                i = last[s[i] - 'a'] + 1;
                fill(last, last + ASCII_MAX, -1);   // 重新开始
            }
            last[s[i] - 'a'] = i;
        }
        return max(len, max_len);  // 别忘了最后一次, 例如"abcd"
    }
};
```

相关题目

- 无

第 11 章
动态规划

11.1 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

```
// LeetCode, Triangle
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

11.2 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 d[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：

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

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 $d[j-1]+S[j]$。
- 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 1：直接在 i 到 j 之间暴力枚举，复杂度是 $O(n^3)$
- 思路 2：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 3：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 4：把思路 2$O(n^2)$ 的代码稍作处理，得到 $O(n)$ 的算法
- 思路 5：当成 M=1 的最大 M 子段和

代码

```
// LeetCode, Maximum Subarray
class Solution {
public:
    int maxSubArray(int A[], int n) {
        return mcss(A, n);
        //return mcss_dp(A, n);
    }
private:
    /**
     * @brief 最大连续子序列和, 思路四
     * @param[in] S 数列
     * @param[in] n 数组的长度
     * @return 最大连续子序列和
     */
    static int mcss(int S[], int n) {
        int i, result, cur_min;
        int *sum = (int*) malloc((n + 1) * sizeof(int));  // 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] + S[i - 1];
        }
        for (i = 1; i <= n; i++) {
            result = max(result, sum[i] - cur_min);
            cur_min = min(cur_min, sum[i]);
        }
        free(sum);
        return result;
    }

    /**
     * @brief 最大连续子序列和, 动规
     * @param[in] S 数列
     * @param[in] n 数组的长度
     * @return 最大连续子序列和
     */
    static int mcss_dp(int S[], int n) {
        int i, result;
        int *d = (int*) malloc(n * sizeof(int));
        d[0] = S[0];
        result = d[0];
        for (i = 1; i < n; i++) {
            d[i] = max(S[i], d[i - 1] + S[i]);
            if (result < d[i])
                result = d[i];
        }
        free(d);
        return result;
    }
```

```
};
```

相关题目

- Binary Tree Maximum Path Sum，见 §4.5.3

11.3 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

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

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

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

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

```
//LeetCode, Palindrome Partitioning II
class Solution {
public:
    int minCut(string s) {
        const int len = s.size();
        int f[len+1];
        bool p[len][len];
        //the worst case is cutting by each char
        for (int i = 0; i <= len; i++)
            f[i] = len - 1 - i; // 最后一个 f[len]=-1
```

```
        for (int i = 0; i < len; i++)
            for (int j = 0; j < len; j++)
                p[i][j] = false;
        for (int i = len - 1; i >= 0; i--) {
            for (int j = i; j < len; 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 §8.1

11.4 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

```
// LeetCode, Maximal Rectangle
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

11.5 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

```
// LeetCode, Best Time to Buy and Sell Stock III
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 §10.3
- Best Time to Buy and Sell Stock II，见 §10.4

11.6 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

这题用二维动态规划。

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]);
```

代码

```
// LeetCode, Interleaving String
// 动规, 二维数组
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()];
    }
};
```

```
// LeetCode, Interleaving String
// 动规, 滚动数组
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()];
    }
};
```

```
// LeetCode, Interleaving String
// 递归，小集合可以过，大集合会超时
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));
    }
};
```

相关题目

- 无

11.7 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".

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

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

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

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

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

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

分析

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

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

加缓存，可以用数组或 HashMap。本题维数较高，用 HashMap，`map` 和 `unordered_map` 均可。

既然可以用记忆化搜索，这题也一定可以用动规。设状态为 `f[i][j][k]`，表示长度为 i，起点为 `s1[j]` 和起点为 `s2[]` 两个字符串是否互为 scramble，则状态转移方程为

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

代码

```
// LeetCode, Interleaving String
// 递归, 小集合可以过, 大集合会超时
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;
    }
};
```

```
// LeetCode, Interleaving String
// 递归 + 剪枝
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;
    }
};
```

```
// LeetCode, Interleaving String
// 递归 +map 做 cache
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 ((cachedIsScramble(first1, first1 + i, first2)
                    && cachedIsScramble(first1 + i, last1, first2 + i))
                    || (cachedIsScramble(first1, first1 + i, last2 - i)
                            && cachedIsScramble(first1 + i, last1, first2)))
                return true;

        return false;
    }

    bool cachedIsScramble(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));
    }
};
```

```
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;
    }
};
}

// LeetCode, Interleaving String
// 递归 +unordered_map 做 cache, 比 map 快
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 ((cachedIsScramble(first1, first1 + i, first2)
                    && cachedIsScramble(first1 + i, last1, first2 + i))
                    || (cachedIsScramble(first1, first1 + i, last2 - i)
                            && cachedIsScramble(first1 + i, last1, first2)))
                return true;

        return false;
    }

    bool cachedIsScramble(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

第 12 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。

12.1 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 $O(n)$

代码

```
//LeetCode, Two Sum
// 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)，代码如下，
class Solution {
public:
    vector<int> twoSum(vector<int> &numbers, int target) {
        unordered_map<int, int> mapping;
        vector<int> result;
        for (int i = 0; i < numbers.size(); i++) {
            mapping[numbers[i]] = i;
        }
        for (int i = 0; i < numbers.size(); i++) {
            const int gap = target - numbers[i];
            if (mapping.find(gap) != mapping.end()) {
                result.push_back(i + 1);
```

```
                result.push_back(mapping[gap] + 1);
                break;
            }
        }
        return result;
    }
};
```

相关题目

- 无

12.2 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) { }
};

//LeetCode, Insert Interval
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 §12.3

12.3 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) { }
};

//LeetCode, Merge Interval
//复用一下 Insert Intervals 的解法即可
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 §12.2

12.4 Minimum Window Substring

描述

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity $O(n)$.

For example, `S = "ADOBECODEBANC", T = "ABC"`

Minimum window is `"BANC"`.

Note:

- If there is no such window in S that covers all characters in T, return the emtpy string `""`.
- If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

分析

双指针，动态维护一个区间。尾指针不断往后扫，当扫到有一个窗口包含了所有 T 的字符后，然后再收缩头指针，直到不能再收缩为止。最后记录所有可能的情况中窗口最小的

代码

```
// LeetCode, Minimum Window Substring
// 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
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

12.5 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

```
// LeetCode, Multiply Strings
// @author 连城 (http://weibo.com/lianchengzju)
// 一个字符对应一个 int
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 move(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 move(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 move(z);
}

class Solution {
```

```
public:
    string multiply(string num1, string num2) {
        return to_string(make_bigint(num1) * make_bigint(num2));
    }
};
// LeetCode, Multiply Strings
// 9 个字符对应一个 int64_t
/** 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

12.6 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

```
// LeetCode, Substring with Concatenation of All Words
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

12.7 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。

代码

```
// LeetCode, Pascal's Triangle
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;
    }
};
```

相关题目

- Pascal's Triangle II，见 §12.8

12.8 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

```
// LeetCode, Pascal's Triangle II
// 滚动数组
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> result(rowIndex + 2, 0);

        result[1] = 1;
        for (int i = 0; i < rowIndex; i++) {
            for (int j = rowIndex + 1; j > 0; j--) {
                result[j] = result[j - 1] + result[j];
            }
        }
        result.erase(result.begin());
        return result;
    }
};
```

相关题目

- Pascal's Triangle，见 §12.7

12.9 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

无

代码

```
// LeetCode, Spiral Matrix
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int> > &matrix) {
        const int m = matrix.size();
        if (m == 0) return vector<int>();
        const int n = matrix[0].size();
        if (n == 0) return vector<int>();

        vector<int> result(m * n, 0);

        int layer, index;
        for (layer = min(m, n), index = 0; layer > 1; layer -= 2) {
            const int offset = (min(m, n) - layer) / 2;
            // left to right
            for (int i = offset; i < n - offset - 1; i++)
                result[index++] = matrix[offset][i];
            // top to bottom
            for (int i = offset; i < m - offset - 1; i++)
                result[index++] = matrix[i][n - offset - 1];
            // right to left
            for (int i = n - offset - 1; i > offset; i--)
                result[index++] = matrix[m - offset - 1][i];
            // bottom to top
            for (int i = m - offset - 1; i > offset; i--)
                result[index++] = matrix[i][offset];
        }

        if (layer == 1) {  // 最后一行
```

```
            if (m < n)
                for (int i = m / 2; i < n - m / 2; i++)
                    result[index++] = matrix[m / 2][i];
            else
                for (int i = n / 2; i < m - n / 2; i++)
                       result[index++] = matrix[i][n / 2];
        }
        return result;
    }
};
```

相关题目

- Spiral Matrix II，见 §12.10

12.10 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

```
// LeetCode, Spiral Matrix II
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        if (n == 0) return vector<vector<int> >();

        vector<vector<int> > matrix(n, vector<int>(n, 0));

        int num = 1, layer;
        for (layer = n; layer > 1; layer -= 2) {
            const int offset = (n - layer) / 2;
            // left to right
            for (int i = offset; i < n - offset - 1; i++)
                matrix[offset][i] = num++;
```

```
            // top to bottom
            for (int i = offset; i < n - offset - 1; i++)
                matrix[i][n - offset - 1] = num++;
            // right to left
            for (int i = n - offset - 1; i > offset; i--)
                matrix[n - offset - 1][i] = num++;
            // bottom to top
            for (int i = n - offset - 1; i > offset; i--)
                matrix[i][offset] = num++;
        }
        if (layer == 1) matrix[n / 2][n / 2] = num;

        return matrix;
    }
};
```

相关题目

- Spiral Matrix，见 §12.9