# LeetCode 题解

戴方勤 (soulmachine@gmail.com)

https://gitcafe.com/soulmachine/LeetCode

最后更新 2013-9-13

## 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

动态分配定长数组，不要用 `arr = new type[len]`，用 std::vector<type> arr(len)，new 还需要 delete，容易出错。

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`。

# 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]) continue;
            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 = std::find_if(first, last,
                    std::bind1st(std::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) continue;
                else occur++;
            } else {
                occur = 1;
            }
            A[++index] = A[i];
        }
        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 l = 0, r = n - 1;
        while (l <= r) {
            const int m = (l + r) / 2;
            if (A[m] == target)
                return m;
            if (A[m] >= A[l]) {
                if (A[l] <= target && target <= A[m])
                    r = m - 1;
                else
                    l = m + 1;
            } else {
                if (A[m] >= target || target >= A[l])
                    r = m - 1;
                else
```

```
                    l = m + 1;
            }
        }
        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
class Solution {
public:
    bool search(int A[], int n, int target) {
        int l = 0;
        int r = n - 1;
        while (l <= r) {
            int m = (l + r) / 2;
            if (A[m] == target)
                return true;
            if (A[l] < A[m]) {
                if (target >= A[l] && target < A[m])
                    r = m - 1;
                else
                    l = m + 1;
            } else if (A[l] > A[m]) {
                if (target > A[m] && target <= A[r])
                    l = m + 1;
```

```
                else
                    r = m - 1;
            } else
                //skip duplicate one, A[start] == A[mid]
                l++;
        }
        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

## 2.2 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 §3.1）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 = new ListNode(-1); // 头节点
        ListNode* pre = head;
        ListNode *pa = l1, *pb = l2;
        int carry = 0;
        while (pa != nullptr || pb != nullptr) {
            int av = pa == nullptr ? 0 : pa->val;
            int bv = pb == nullptr ? 0 : pb->val;
            ListNode* node = new ListNode((av + bv + carry) % 10);
            carry = (av + bv + carry) / 10;
            pre->next = node; // 尾插法
            pre = pre->next;
            pa = pa == nullptr ? nullptr : pa->next;
            pb = pb == nullptr ? nullptr : pb->next;
        }
        if (carry > 0)
            pre->next = new ListNode(1);
        pre = head->next;
        delete head;
```

```
        return pre;
    }
};
```

### 相关题目

- Add Binary，见 §3.1

## 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;
        while(p) {
            if (count == m-1) pm = p;
            if (count == n) {
                pn = p;
                break;
            }
            count++;
            p = p->next;
        }
        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* temp = cur;
                cur = cur->next;
                prev->next = cur;
                delete temp;
                continue;
```

```
            }
            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

这类题目考察字符串的操作。

## 3.1 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;
        int maxL = a.size() > b.size() ? a.size() : b.size();
        std::reverse(a.begin(), a.end());
        std::reverse(b.begin(), b.end());
        int carry = 0;
        for (int i = 0; i < maxL; i++) {
            int ai = i < a.size() ? a[i] - '0' : 0;
            int bi = i < b.size() ? b[i] - '0' : 0;
            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

# 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*> queToPush, queToPop;
        queToPop.push(root);
        while (!queToPop.empty()) {
            vector<int> level; // elments in level level

            while (!queToPop.empty()) {
                TreeNode* node = queToPop.front();
                queToPop.pop();
                level.push_back(node->val);
                if (node->left != nullptr) queToPush.push(node->left);
                if (node->right != nullptr) queToPush.push(node->right);
            }
            result.push_back(level);
            swap(queToPush, queToPop); //!!! 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> > retTemp;

        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) {
                    retTemp.push_front(level);
```

```
                    level.erase(level.begin(),level.end());
                    q.push(nullptr);
                }
            }
        }

        vector<vector<int> > ret;
        for(list<vector<int> >::iterator it = retTemp.begin();
                it != retTemp.end(); ++it) {
            ret.push_back(*it);
        }
        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> temp;
                    for (int i = level.size() - 1; i >= 0; --i) {
                        temp.push_back(level[i]);
                    }
                    result.push_back(temp);
                }
                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;
        std::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 and node->right != cur)
                    node = node->right;

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

                if (node->right == nullptr) {
                    node->right = cur;
                    prev = cur;
                    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 and prev->val > current->val) {
            if (broken.first == nullptr) {
                broken.first = prev;
            } //不能用 else
```

```
            broken.second = current;
        }
    }
};
```

### 相关题目

- Binary Tree Inorder Traversal，见 §4.1.4

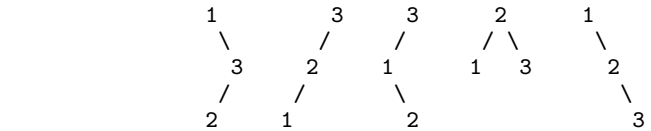
## 4.2 二叉查找树

### 4.2.1 Unique Binary Search Trees

#### 描述

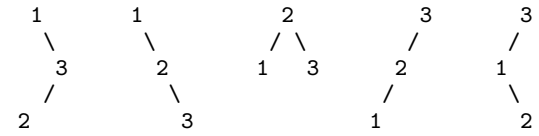
Given $n$, how many structurally unique BST's (binary search trees) that store values $1...n$?

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。



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

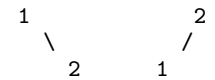
当数组为 $1, 2, 3, ..., n$ 时，基于以下原则的构建的 BST 树具有唯一性：**以 i 为根节点的树，其左子树由 [1, i-1] 构成，其右子树由 [i+1, n] 构成。**

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

如果数组为空，毫无疑问，只有一种 BST，即空树，$f(0) = 1$。

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

如果数组有两个元素 1,2，那么有如下两种可能



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

再看一看 3 个元素的数组，可以发现 BST 的取值方式如下：

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

所以，由此观察，可以得出 $f$ 的递推公式为

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

至此，问题划归为一维动态规划。

### 代码

```
// 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

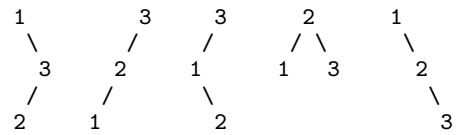
- Unique Binary Search Trees II，见 §4.2.2

## 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) {
        auto subTree = new vector<TreeNode*>();
        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->begin(); i < leftSubs->end(); i++) {
                for (auto j = rightSubs->begin(); j < rightSubs->end(); j++) {
                    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 and root->val < upper
                and isValidBST(root->left, lower, root->val)
                and 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

分治法，二分。

### 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 = std::distance (first, last);

        if (length == 0) return nullptr;
        if (length == 1) return new TreeNode (*first);

        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 = std::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(std::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(std::make_pair(node->left, depth + 1));

            if (node->right && result > depth) // 深度控制，剪枝
                s.push(std::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 std::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(std::begin(preorder), std::end(preorder),
                std::begin(inorder), std::end(inorder));
    }

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

        auto root = new TreeNode(*pre_first);

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

        root->left = buildTree(std::next(pre_first), std::next(pre_first,
                leftSize + 1), in_first, std::next(in_first, leftSize));
        root->right = buildTree(std::next(pre_first, leftSize + 1), pre_last,
                std::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(std::begin(inorder), std::end(inorder),
                std::begin(postorder), std::end(postorder));
    }

    template<typename BidiIt>
    TreeNode* buildTree(BidiIt in_first, BidiIt in_last,
            BidiIt post_first, BidiIt post_last) {
        if (std::distance(in_first, in_last) == 0) return nullptr;
        if (std::distance(post_first, post_last) == 0) return nullptr;

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

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

        root->left = buildTree(in_first, in_root_pos, post_first, post_left_last);
        root->right = buildTree(std::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 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

在父节点时就可以及时进行判断，因此本题适合用先根遍历。

### 代码

```
// LeetCode, Same Tree
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        if (!p and !q) return true;
        if (!p || !q) return false;
        return p->val == q->val
                and isSameTree(p->left, q->left)
                and isSameTree(p->right, q->right);
    }
};
```

### 相关题目

- Symmetric Tree，见 §4.5.2

## 4.5.2 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

在父节点时就可以及时进行判断，因此本题适合用先根遍历。

### 代码

```
// 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 and !right) return true;
        if (!left or !right) return false;
        if (left->val != right->val) return false;
        return isSymmetric(left->left, right->right)
                and isSymmetric(left->right, right->left);
    }
};
```

```
// 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 lhs = s.top ();
            s.pop();

            auto rhs = s.top ();
            s.pop();

            if (!lhs and !rhs) continue;
            if (!lhs or !rhs) return false;
            if (lhs->val != rhs->val) return false;

            s.push(lhs->left);
            s.push(rhs->right);

            s.push(lhs->right);
            s.push(rhs->left);
        }

        return true;
    }
};
```

### 相关题目

- Same Tree，见 §4.5.1

## 4.5.3 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;

        int lhs = balancedHeight (root->left);
        int rhs = balancedHeight (root->right);

        if (lhs < 0 or rhs < 0 or std::abs (lhs - rhs) > 1)
            return -1;

        return std::max(lhs, rhs) + 1;
    }
};
```

### 相关题目

- 无

## 4.5.4 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
class Solution {
public:
    void flatten(TreeNode *root) {
        if (root == nullptr) return;
        //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 ** ptn = & (root->left->right);
        while (*ptn)
            ptn = & ((*ptn)->right);
        //4.2.connect right sub tree after left sub tree
        *ptn = 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 top = s.top();
            s.pop();

            if (top->right)
                s.push(top->right);
            if (top->left)
```

```
                s.push(top->left);

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

### 相关题目

- 无

## 4.5.5 Binary Tree Maximum Path Sum

### 描述

Given a binary tree, find the maximum path sum.

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

```
  1
 / \
2   3
```

Return 6.

### 分析

这题很难，路径可以从任意节点开始，到任意节点结束。

可以利用“最大连续子序列和”问题的思路，见第 §8.1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 = INT_MIN;
        dfs(root);
        return max;
    }
private:
    int max;
    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 = std::max(max, sum);
        return std::max(r, l) > 0 ? std::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 §8.1

# 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 = new ListNode(-1);
        ListNode* p = head;
        while (l1 != nullptr || l2 != nullptr) {
            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;
            }
            p = p->next;
        }
        p = head->next;
        delete head;
        return p;
    }
};
```

### 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 = new ListNode(-1);
        ListNode* p = head;
        while (l1 != nullptr || l2 != nullptr) {
            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;
            }
            p = p->next;
        }
        p = head->next;
        delete head;
        return p;
    }
};
```

### 相关题目

- Merge Sorted Array §5.1
- Merge Two Sorted Lists，见 §5.2

# 第 6 章
# 分治法

## 6.1 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 {
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;
    }
public:
    double pow(double x, int n) {
        if (n < 0) return 1.0 / power(x, -n);
        else return power(x, n);
    }
};
```

### 相关题目

- 无

# 第 7 章
# 贪心法

## 7.1 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

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

### 相关题目

- Best Time to Buy and Sell Stock II，见 §7.2
- Best Time to Buy and Sell Stock III，见 §8.4

## 7.2 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

```
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 §7.1
- Best Time to Buy and Sell Stock III，见 §8.4

# 第 8 章
# 动态规划

## 8.1 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 §4.5.5

## 8.2 Palindrome Partitioning II

### 描述

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

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given `s = "aab"`,

Return 1 since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

### 分析

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

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

这是一个二维函数，实际写代码比较麻烦。

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

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

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

定义状态 `P[i][j] = true if [i,j]` 为回文，那么

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

### 代码

```
//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 §10.1

## 8.3 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

## 8.4 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 = std::min(valley, prices[i]);
            f[i] = std::max(f[i - 1], prices[i] - valley);
        }

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

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

        return maxProfit;
    }
};
```

### 相关题目

- Best Time to Buy and Sell Stock，见 §7.1
- Best Time to Buy and Sell Stock II，见 §7.2

## 8.5 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] += std::min(triangle[i + 1][j],
                        triangle[i + 1][j + 1]);

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

### 相关题目

- 无

## 8.6 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] and f[i - 1][j])
       or (s2[j - 1] == s3 [i + j - 1] and 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] and s1[i - 1] == s3[i - 1];

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

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

```
            for (size_t j = 1; j <= s2.length(); ++j)
                f[j] = (s1[i - 1] == s3[i + j - 1] and f[j])
                        or (s2[j - 1] == s3[i + j - 1] and 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 and first2 == last2;

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

### 相关题目

- 无

## 8.7 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，和动规中自顶向下的记忆化搜索类似。

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，`std::map` 和 `std::unordered_map` 均可。

其次，这题可以用动规。

## 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)
                 and isScramble(first1 + i, last1, first2 + i))
                    or (isScramble(first1, first1 + i, last2 - i)
                            and 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
        std::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)
                 and isScramble(first1 + i, last1, first2 + i))
                    or (isScramble(first1, first1 + i, last2 - i)
                            and 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)
                    and cachedIsScramble(first1 + i, last1, first2 + i))
                    or (cachedIsScramble(first1, first1 + i, last2 - i)
                            and 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;
        std::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)
                    and cachedIsScramble(first1 + i, last1, first2 + i))
                    or (cachedIsScramble(first1, first1 + i, last2 - i)
                            and 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 9 章
# 广度优先搜索

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

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

## 9.1 Word Ladder

### 描述

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

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

For example, Given:

```
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
```

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

Note:

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

### 分析

### 代码

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

```
public:
    int ladderLength(string start, string end, unordered_set<string> &dict) {
        if (start.size() != end.size()) return 0;
        if (start.empty() || end.empty()) return 1; return 0;
        int level = 1;  // 层次
        queue<string> queToPush, queToPop;

        queToPop.push(start);
        while (dict.size() > 0 && !queToPop.empty()) {
            while (!queToPop.empty()) {
                string str(queToPop.front());
                queToPop.pop();
                for (int i = 0; i < str.size(); i++) {
                    for (char j = 'a'; j <= 'z'; j++) {
                        if (j == str[i]) continue;

                        const char temp = str[i];
                        str[i] = j;
                        if (str == end) return level + 1; //找到了

                        if (dict.count(str) > 0) {
                            queToPush.push(str);
                            dict.erase(str); // 删除该单词，防止死循环
                        }
                        str[i] = temp; // 恢复该单词
                    }
                }
            }
            swap(queToPush, queToPop); //!!! 交换两个队列
            level++;
        }
        return 0; // 所有单词已经用光，还是找不到通向目标单词的路径
    }
};
```

### 相关题目

- Word Ladder II，见 §9.2

## 9.2 Word Ladder II

### 描述

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

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

For example, Given:

```
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
```

Return

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

Note:

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

## 分析

跟 Word Ladder 比，这题是求路径本身，不是路径长度，也是 BFS，略微麻烦点

## 代码

```
//LeetCode, Word Ladder II
//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
class Solution {
public:
    std::vector<std::vector<std::string> > findLadders(std::string start,
            std::string end, std::unordered_set<std::string> &dict) {
        result_.clear();
        std::unordered_map<std::string, std::vector<std::string>> prevMap;

        for (auto iter = dict.begin(); iter != dict.end(); ++iter) {
            prevMap[*iter] = std::vector<std::string>();
        }

        std::vector<std::unordered_set<std::string>> candidates(2);

        int current = 0;
        int previous = 1;

        candidates[current].insert(start);

        while (true) {
            current = !current;
            previous = !previous;

            // 从 dict 中删除 previous 中的单词，避免自己指向自己
            for (auto iter = candidates[previous].begin();
                    iter != candidates[previous].end(); ++iter) {
                dict.erase(*iter);
            }

            candidates[current].clear();
```

```
            for (auto iter = candidates[previous].begin();
                    iter != candidates[previous].end(); ++iter) {
                for (size_t pos = 0; pos < iter->size(); ++pos) {
                    std::string word = *iter;
                    for (int i = 'a'; i <= 'z'; ++i) {
                        if (word[pos] == i) continue;

                        word[pos] = i;

                        if (dict.count(word) > 0) {
                            prevMap[word].push_back(*iter);
                            candidates[current].insert(word);
                        }
                    }
                }
            }

            if (candidates[current].size() == 0) return result_;  // 此题无解
            if (candidates[current].count(end)) break; // 没看懂
        }

        std::vector<std::string> path;
        GeneratePath(prevMap, path, end);

        return result_;
    }

private:
    std::vector<std::vector<std::string>> result_;

    void GeneratePath(
            std::unordered_map<std::string, std::vector<std::string>> &prevMap,
            std::vector<std::string>& path, const std::string& word) {
        if (prevMap[word].size() == 0) {
            path.push_back(word);
            std::vector<std::string> curPath = path;
            reverse(curPath.begin(), curPath.end());
            result_.push_back(curPath);
            path.pop_back();
            return;
        }

        path.push_back(word);
        for (auto iter = prevMap[word].begin(); iter != prevMap[word].end();
                ++iter) {
            GeneratePath(prevMap, path, *iter);
        }
        path.pop_back();
    }
};
```

### 相关题目

- Word Ladder，见 §11.1

# 第 10 章
# 深度优先搜索

## 10.1 Palindrome Partitioning

### 描述

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

Return all possible palindrome partitioning of s.

For example, given `s = "aab"`, Return

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

### 分析

在每一步都可以判断中间结果是否为合法结果，用回溯法。

一个长度为 n 的字符串，有 n+1 个地方可以砍断，每个地方可断可不断，因此复杂度为 $O(2^{n+1})$

### 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, output, result);
        return result;
    }
    // 搜索必须以 s[start] 开头的 partition 方案
    // 如果一个字符串长度为 n，则可以插入 n+1 个隔板，复制度为 O(2^{n+1})
    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 §8.2

# 第 11 章
# 基本实现题

这类题目不考特定的算法，纯粹考察写代码的熟练度。

## 11.1 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 1：暴力，O(n^2)
// 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

## 11.2 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 §11.3

## 11.3 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 §11.2