# Cracking the Coding Interview C++11

張世明 (simon.zhangsm@gmail.com)

https://github.com/simonzhangsm/acm-cheat-sheet

2014-12-30

## 内容简介

本书包含了 LeetCode Online Judge(http://leetcode.com/onlinejudge) 所有题目及答案，部分包含 Careerup(http://www.careerup.com) 面试题目，所有代码经过精心编写，使用 C++ + STL 风格，编码规范良好，适合读者反复揣摩模仿，甚至在纸上默写。

- 经常使用全局变量。比如用几个全局变量，定义某个递归函数需要的数据，减少递归函数的参数个数，就减少了递归时栈内存的消耗，可以说这几个全局变量是这个递归函数的环境。
- 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。如不需要检查 malloc/new 返回的指针是否为 NULL；不检查内部函数入口参数的有效性；基于 C++11 对象编程时，调用对象的成员方法，不需要检查对象自身是否为 NULL；不使用 Try/Catch/Throw 异常处理机制。

## Github 地址

开源项目地址：https://github.com/simonzhangsm/acm-cheat-sheet

## 网络资源

- Leetcode：https://www.leetcode.com
- LintCode：http://lintcode.com/zh-cn/daily
- CodeEval：https://www.codeeval.com
- TopCoder：https://www.topcoder.com
- HackerRank：https://www.hackerrank.com
- ACM Home：http://www.acmerblog.com
- 结构之法算法之道博客：http://blog.csdn.net/v_JULY_v
- Princeton Algorithm：http://algs4.cs.princeton.edu/home

# 目录

# 第 1 章
# 编程技巧

较大的数组放在 main 函数外，作为全局变量，这样可以防止栈溢出，因为栈的大小是有限制的。如果能够预估栈，队列的上限，则不要用 `stack, queue`，使用数组来模拟，这样速度最快。输入数据一般放在全局变量，且在运行过程中不要修改这些变量。

在判断两个浮点数 (i.e., float/double) a 和 b 是否相等时，不应该用 `a==b`，而是判断二者之差的绝对值 `fabs(a-b)` 是否小于某个阀值 $\epsilon$=`1e-9`。

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

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

以下是关于 STL 基于《Effective STL》的使用技巧。

### 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**

**用 empty 来代替检查 size() 是否为 0**

**确保 new|delete 和 malloc|free 的成对出现, 使用智能指针 shared_ptr 和 unique_ptr，替代 auto_ptr 容器**

**用 distance 和 advance 把 const_iterator 转化成 iterator**

# 第 2 章
# 线性表

线性表 (Linear List) 包含：

- 顺序存储：数组
- 链式存储：单链表，双向链表，循环单链表，循环双向链表
- 二者结合：静态链表

## 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

二指针问题, 一前一后扫描。

#### 代码 1

```
// LeetCode, Remove Duplicates from Sorted Array
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    int removeDuplicates(int A[], int n) {
        if (n==0 || A==nullptr) return 0;
        int index = 0;
        for (int i = 1; i < n; i++) {
            if (A[index] != A[i])
                A[++index] = A[i];
```

```
            }
            return index + 1;
        }
    };
```

### 代码 2

```
    // LeetCode, Remove Duplicates from Sorted Array
    // 使用 STL，时间复杂度 O(n)，空间复杂度 O(1)
    class Solution {
        int removeDuplicates(int A[], int n) {
            return distance(A, unique(A, A + n));
        }
    };
```

### 代码 3

```
    // LeetCode, Remove Duplicates from Sorted Array
    // 使用 STL，时间复杂度 O(n)，空间复杂度 O(1)
    class Solution {
        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 = upper_bound(first, last, *first);
            }
            return output;
        }
    };
```

### 相关题目

- Remove Duplicates from Sorted Array II，见 §2.1.2

## 2.1.2 Remove Duplicates from Sorted Array II

### 描述

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice? For example, Given sorted array `A = [1,1,1,2,2,3]`, Your function should return length = 5, and A is now `[1,1,2,2,3]`

### 分析

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

### 代码 1

```
// LeetCode, Remove Duplicates from Sorted Array II
// 时间复杂度 O(n)，空间复杂度 O(1)
// @author hex108 (https://github.com/hex108)
class Solution {
    public:
    int removeDuplicates(int A[], int n) {
        if (n <= 2) return n;

        int index = 2;
        for (int i = 2; i < n; i++){
            if (A[i] != A[index - 2])
            A[index++] = A[i];
        }

        return index;
    }
};
```

### 代码 2

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

```
// LeetCode, Remove Duplicates from Sorted Array II
// @author 虞航仲 (http://weibo.com/u/1666779725)
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int removeDuplicates(int A[], int n) {
        int index = 0;
        for (int i = 0; i < n; ++i) {
            if (i > 0 && i < n - 1 && A[i] == A[i - 1] && A[i] == A[i + 1])
            continue;

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

### 相关题目

- Remove Duplicates from Sorted Array，见 §2.1.1

## 2.1.3 Search in Rotated Sorted Array

### 描述

Suppose a sorted array is rotated at some pivot unknown to you beforehand. (i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`). You are given a target value to search. If found in the array return its index, otherwise return -1. You may assume no duplicate exists in the array.

### 分析

二分查找，难度主要在于左右边界的确定。

### 代码

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

### 相关题目

- Search in Rotated Sorted Array II，见 §2.1.4

### 2.1.4 Search in Rotated Sorted Array II

#### 描述

Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

#### 分析

允许重复元素，则上一题中如果 `A[m]>=A[l]`，那么 `[l,m]` 为递增序列的假设就不能成立了，比如 `[1,3,1,1,1]`。

如果 `A[m]>=A[l]` 不能确定递增，那就把它拆分成两个条件：

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

#### 代码

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

### 相关题目

- Search in Rotated Sorted Array，见 §2.1.3

## 2.1.5 Median of Two Sorted Arrays

### 描述

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

### 分析

这是一道非常经典的题。这题更通用的形式是，给定两个已经排序好的数组，找到两者所有元素中第 $k$ 大的元素。

$O(m+n)$ 的解法比较直观，直接 merge 两个数组，然后求第 $k$ 大的元素。

不过我们仅仅需要第 $k$ 大的元素，是不需要“排序”这么复杂的操作的。可以用一个计数器，记录当前已经找到第 $m$ 大的元素了。同时我们使用两个指针 `pA` 和 `pB`，分别指向 A 和 B 数组的第一个元素，使用类似于 merge

sort 的原理，如果数组 A 当前元素小，那么 `pA++`，同时 `m++`；如果数组 B 当前元素小，那么 `pB++`，同时 `m++`。最终当 $m$ 等于 $k$ 的时候，就得到了我们的答案，$O(k)$ 时间，$O(1)$ 空间。但是，当 $k$ 很接近 $m+n$ 的时候，这个方法还是 $O(m+n)$ 的。

有没有更好的方案呢？我们可以考虑从 $k$ 入手。如果我们每次都能够删除一个一定在第 $k$ 大元素之前的元素，那么我们需要进行 $k$ 次。但是如果每次我们都删除一半呢？由于 A 和 B 都是有序的，我们应该充分利用这里面的信息，类似于二分查找，也是充分利用了“有序”。

假设 A 和 B 的元素个数都大于 $k/2$，我们将 A 的第 $k/2$ 个元素（即 `A[k/2-1]`）和 B 的第 $k/2$ 个元素（即 `B[k/2-1]`）进行比较，有以下三种情况（为了简化这里先假设 $k$ 为偶数，所得到的结论对于 $k$ 是奇数也是成立的）：

- `A[k/2-1] == B[k/2-1]`
- `A[k/2-1] > B[k/2-1]`
- `A[k/2-1] < B[k/2-1]`

如果 `A[k/2-1] < B[k/2-1]`，意味着 `A[0]` 到 `A[k/2-1` 的肯定在 $A \cup B$ 的 top k 元素的范围内，换句话说，`A[k/2-1` 不可能大于 $A \cup B$ 的第 $k$ 大元素。留给读者证明。

因此，我们可以放心的删除 A 数组的这 $k/2$ 个元素。同理，当 `A[k/2-1] > B[k/2-1]` 时，可以删除 B 数组的 $k/2$ 个元素。

当 `A[k/2-1] == B[k/2-1]` 时，说明找到了第 $k$ 大的元素，直接返回 `A[k/2-1]` 或 `B[k/2-1]` 即可。

因此，我们可以写一个递归函数。那么函数什么时候应该终止呢？

- 当 A 或 B 是空时，直接返回 `B[k-1]` 或 `A[k-1]`；

- 当 k=1 是，返回 min(A[0], B[0])；
- 当 A[k/2-1] == B[k/2-1] 时，返回 A[k/2-1] 或 B[k/2-1]

### 代码

```
// LeetCode, Median of Two Sorted Arrays
// 时间复杂度 O(log(m+n))，空间复杂度 O(log(m+n))
class Solution {
    public:
    double findMedianSortedArrays(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)) / 2.0;
    }
    private:
    static int 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 ia = min(k / 2, m), ib = k - ia;
        if (A[ia - 1] < B[ib - 1])
        return find_kth(A + ia, m - ia, B, n, k - ia);
        else if (A[ia - 1] > B[ib - 1])
        return find_kth(A, m, B + ib, n - ib, k - ib);
        else
        return A[ia - 1];
    }
};
```

### 相关题目

- 无

## 2.1.6 Longest Consecutive Sequence

### 描述

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

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

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

## 分析

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

由于序列里的元素是无序的，又要求 $O(n)$，首先要想到用哈希表。

用一个哈希表 `unordered_map<int, bool> used` 记录每个元素是否使用，对每个元素，以该元素为中心，往左右扩张，直到不连续为止，记录下最长的长度。

## 代码

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

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

        int longest = 0;

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

            int length = 1;

            used[i] = true;

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

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

            longest = max(longest, length);
        }

        return longest;
    }
};
```

### 分析 2

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

### 代码

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

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

### 相关题目

- 无

### 2.1.7 Two Sum

#### 描述

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

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

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

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

Output: `index1=1, index2=2`

#### 分析

方法 1：暴力，复杂度 $O(n^2)$，会超时

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

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

#### 代码

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

### 相关题目

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

## 2.1.8 3Sum

### 描述

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

Note:

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

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

A solution set is:

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

### 分析

先排序，然后左右夹逼，复杂度 $O(n^2)$。

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

### 代码

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

        auto last = num.end();
        for (auto a = num.begin(); a < prev(last, 2); ++a) {
            auto b = next(a);
            auto c = prev(last);
            while (b < c) {
```

```
                    if (*a + *b + *c < target) {
                        ++b;
                    } else if (*a + *b + *c > target) {
                    --c;
                } else {
                result.push_back({ *a, *b, *c });
                ++b;
                --c;
            }
        }
    }
    sort(result.begin(), result.end());
    result.erase(unique(result.begin(), result.end()), result.end());
    return result;
}
};
```

### 相关题目

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

## 2.1.9 3Sum Closest

### 描述

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

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

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

### 分析

先排序，然后左右夹逼，复杂度 $O(n^2)$。

### 代码

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

```
            int min_gap = INT_MAX;

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

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

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

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

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

            return result;
        }
    };
```

### 相关题目

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

## 2.1.10 4Sum

### 描述

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

Note:

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

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

A solution set is:

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

### 分析

先排序，然后左右夹逼，复杂度 $O(n^3)$，会超时。

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

### 左右夹逼

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

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

### map 做缓存

```
// LeetCode, 4Sum
// 用一个 hashmap 先缓存两个数的和
// 时间复杂度，平均 O(n^2)，最坏 O(n^4)，空间复杂度 O(n^2)
class Solution {
    public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {
```

```
        vector<vector<int>> result;
        if (num.size() < 4) return result;
        sort(num.begin(), num.end());

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

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

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

                    result.push_back( { num[vec[k].first],
                        num[vec[k].second], num[c], num[d] });
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

### multimap

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

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

```
            cache.insert(make_pair(num[i] + num[j], make_pair(i, j)));

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

### 方法 4

```
    // LeetCode, 4Sum
    // 先排序，然后左右夹逼，时间复杂度 O(n^3logn)，空间复杂度 O(1)，会超时
    // 跟方法 1 相比，表面上优化了，实际上更慢了，切记!
    class Solution {
        public:
        vector<vector<int>> fourSum(vector<int>& num, int target) {
            vector<vector<int>> result;
            if (num.size() < 4) return result;
            sort(num.begin(), num.end());

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

```
                    result.push_back({ *a, *b, *c, *d });
                    c = upper_bound(c, d, *c);
                    d = prev(lower_bound(c, d, *d));
                }
            }
        }
    }
    return result;
}
};
```

### 相关题目

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

## 2.1.11 Remove Element

### 描述

Given an array and a value, remove all instances of that value in place and return the new length. The order of elements can be changed. It doesn't matter what you leave beyond the new length.

### 分析

无

### 代码 1

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

### 代码 2

```
// LeetCode, Remove Element
// 使用 remove(), 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    int removeElement(int A[], int n, int elem) {
        return distance(A, remove(A, A+n, elem));
    }
};
```

### 相关题目

- 无

## 2.1.12 Next Permutation

### 描述

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

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

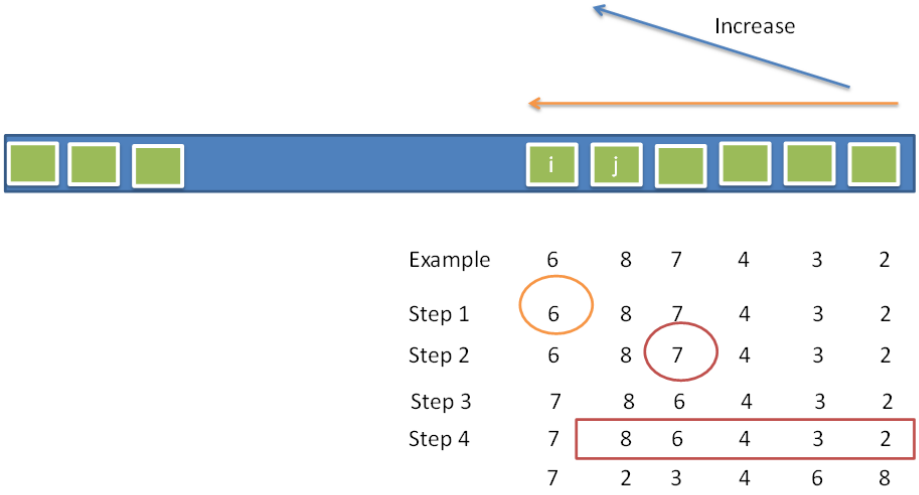
The replacement must be in-place, do not allocate extra memory.

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

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

### 分析

算法过程如图 2-1所示（来自 http://fisherlei.blogspot.com/2012/12/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 2-1 下一个排列算法流程

## 代码

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

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

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

        // Find `pivot`, which is the first element that is no less than its
```

```
            // successor.  `Prev` is used since `pivort` is a
            `reversed_iterator`.
            while (pivot != rlast && *pivot >= *prev(pivot))
            ++pivot;

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

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

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

            return true;
        }
    };
```

### 相关题目

- Permutation Sequence, 见 §2.1.13
- Permutations, 见 §??
- Permutations II, 见 §??
- Combinations, 见 §??

## 2.1.13 Permutation Sequence

### 描述

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

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

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

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

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

### 分析

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

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

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

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

同理，$a_2, a_3, ..., a_n$ 的值推导如下:

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

### 使用 next_permutation()

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

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

### 康托编码

```
// LeetCode, Permutation Sequence
// 康托编码，时间复杂度 O(n)，空间复杂度 O(1)
```

```
    class Solution {
        public:
        string getPermutation(int n, int k) {
            string s(n, '0');
            string result;
            for (int i = 0; i < n; ++i)
            s[i] += i + 1;

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

        // seq 已排好序，是第一个排列
        template<typename Sequence>
        Sequence kth_permutation(const Sequence &seq, int k) {
            const int n = seq.size();
            Sequence S(seq);
            Sequence result;

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

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

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

## 相关题目

- Next Permutation, 见 §2.1.12
- Permutations, 见 §??
- Permutations II, 见 §??
- Combinations, 见 §??

### 2.1.14 Valid Sudoku

#### 描述

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

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

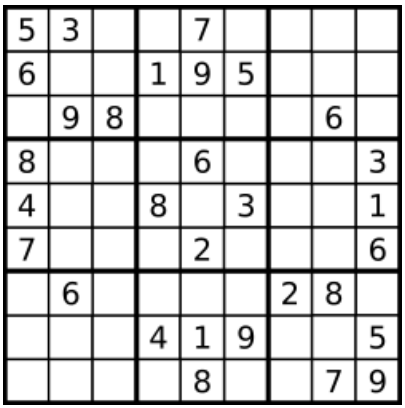



图 2-2 A partially filled sudoku which is valid

#### 分析

细节实现题。

#### 代码

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

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

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

            fill(used, used + 9, false);

            for (int j = 0; j < 9; ++j) // 检查列
```

```
                if (!check(board[j][i], used))
                return false;
            }

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

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

            return true;
        }

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

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

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

### 相关题目

- Sudoku Solver, 见 §??

## 2.1.15 Trapping Rain Water

### 描述

Given $n$ non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return 6.


图 2-3 Trapping Rain Water

## 分析

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

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

也可以，

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

## 代码 1

```
// LeetCode, Trapping Rain Water
// 思路 1，时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    int trap(int A[], int n) {
        int *max_left = new int[n]();
        int *max_right = new int[n]();

        for (int i = 1; i < n; i++) {
            max_left[i] = max(max_left[i - 1], A[i - 1]);
            max_right[n - 1 - i] = max(max_right[n - i], A[n - i]);

        }

        int sum = 0;
```

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

            delete[] max_left;
            delete[] max_right;
            return sum;
        }
    };
```

### 代码 2

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

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

### 代码 3

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

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

```
            int water = 0;

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

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

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

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

            return water;
        }
    };
```

### 相关题目

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

## 2.1.16 Rotate Image

### 描述

You are given an $n \times n$ 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up: Could you do this in-place?

### 分析

首先想到，纯模拟，从外到内一圈一圈的转，但这个方法太慢。

如下图，首先沿着副对角线翻转一次，然后沿着水平中线翻转一次。

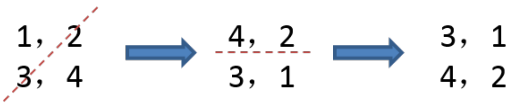



图 2-4 Rotate Image

或者，首先沿着水平中线翻转一次，然后沿着主对角线翻转一次。

### 代码 1

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

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

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

### 代码 2

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

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

        for (int i = 0; i < n; ++i)  // 沿着主对角线反转
        for (int j = i + 1; j < n; ++j)
        swap(matrix[i][j], matrix[j][i]);
    }
};
```

### 相关题目

- 无

## 2.1.17 Plus One

### 描述

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

### 分析

高精度加法。

### 代码 1

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

        for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
            *it += c;
            c = *it / 10;
            *it %= 10;
        }

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

### 代码 2

```
// LeetCode, Plus One
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
```

```
        return digits;
    }
    private:
    // 0 <= digit <= 9
    void add(vector<int> &digits, int digit) {
        int c = digit;  // carry, 进位

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

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

### 相关题目

- 无

## 2.1.18 Climbing Stairs

### 描述

You are climbing a stair case. It takes $n$ steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

### 分析

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

- 从第 $n-1$ 阶前进 1 步；
- 从第 $n-1$ 阶前进 2 步；

因此，有 $f(n)=f(n-1)+f(n-2)$。

这是一个斐波那契数列。

方法 1，递归，太慢；方法 2，迭代。

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

### 迭代

```
// LeetCode, Climbing Stairs
// 迭代，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
```

```
    public:
    int climbStairs(int n) {
        int prev = 0;
        int cur = 1;
        for(int i = 1; i <= n ; ++i){
            int tmp = cur;
            cur += prev;
            prev = tmp;
        }
        return cur;
    }
};
```

### 数学公式

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

### 相关题目

- Decode Ways, 见 §??

## 2.1.19 Gray Code

### 描述

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

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

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

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

Note:

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

## 分析

格雷码 (Gray Code) 的定义请参考 http://en.wikipedia.org/wiki/Gray_code

**自然二进制码转换为格雷码**：$g_0 = b_0, g_i = b_i \oplus b_{i-1}$

保留自然二进制码的最高位作为格雷码的最高位，格雷码次高位为二进制码的高位与次高位异或，其余各位与次高位的求法类似。例如，将自然二进制码 1001，转换为格雷码的过程是：保留最高位；然后将第 1 位的 1 和第 2 位的 0 异或，得到 1，作为格雷码的第 2 位；将第 2 位的 0 和第 3 位的 0 异或，得到 0，作为格雷码的第 3 位；将第 3 位的 0 和第 4 位的 1 异或，得到 1，作为格雷码的第 4 位，最终，格雷码为 1101。

**格雷码转换为自然二进制码**：$b_0 = g_0, b_i = g_i \oplus b_{i-1}$

保留格雷码的最高位作为自然二进制码的最高位，次高位为自然二进制高位与格雷码次高位异或，其余各位与次高位的求法类似。例如，将格雷码 1000 转换为自然二进制码的过程是：保留最高位 1，作为自然二进制码的最高位；然后将自然二进制码的第 1 位 1 和格雷码的第 2 位 0 异或，得到 1，作为自然二进制码的第 2 位；将自然二进制码的第 2 位 1 和格雷码的第 3 位 0 异或，得到 1，作为自然二进制码的第 3 位；将自然二进制码的第 3 位 1 和格雷码的第 4 位 0 异或，得到 1，作为自然二进制码的第 4 位，最终，自然二进制码为 1111。

格雷码有**数学公式**，整数 $n$ 的格雷码是 $n \oplus (n/2)$。

这题要求生成 $n$ 比特的所有格雷码。

方法 1，最简单的方法，利用数学公式，对从 $0 \sim 2^n - 1$ 的所有整数，转化为格雷码。

方法 2，$n$ 比特的格雷码，可以递归地从 $n-1$ 比特的格雷码生成。如图 §2-5所示。

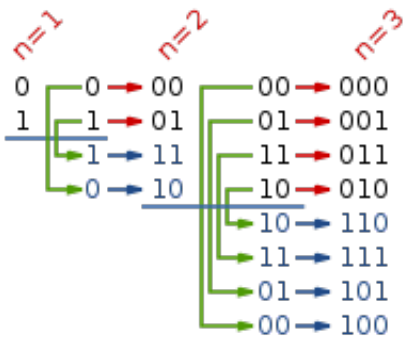



图 2-5 The first few steps of the reflect-and-prefix method.

## 数学公式

```
// LeetCode, Gray Code
// 数学公式，时间复杂度 O(2^n)，空间复杂度 O(1)
class Solution {
    public:
    vector<int> grayCode(int n) {
        vector<int> result;
```

```
        const size_t size = 1 << n;  // 2^n
        result.reserve(size);
        for (size_t i = 0; i < size; ++i)
        result.push_back(binary_to_gray(i));
        return result;
    }
    private:
    static unsigned int binary_to_gray(unsigned int n) {
        return n ^ (n >> 1);
    }
};
```

### Reflect-and-prefix method

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

### 相关题目

- 无

## 2.1.20 Set Matrix Zeroes

### 描述

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

**Follow up**: Did you use extra space?

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

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

Could you devise a constant space solution?

### 分析

$O(m+n)$ 空间的方法很简单，设置两个 bool 数组，记录每行和每列是否存在 0。
想要常数空间，可以复用第一行和第一列。

### 代码 1

```
// LeetCode, Set Matrix Zeroes
// 时间复杂度 O(m*n)，空间复杂度 O(m+n)
class Solution {
    public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        const size_t n = matrix[0].size();
        vector<bool> row(m, false); // 标记该行是否存在 0
        vector<bool> col(n, false); // 标记该列是否存在 0

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

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

### 代码 2

```
// LeetCode, Set Matrix Zeroes
// 时间复杂度 O(m*n)，空间复杂度 O(1)
class Solution {
    public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        const size_t n = matrix[0].size();
```

```
            bool row_has_zero = false; // 第一行是否存在 0
            bool col_has_zero = false; // 第一列是否存在 0

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

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

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

### 相关题目

- 无

## 2.1.21 Gas Station

### 描述

There are $N$ gas stations along a circular route, where the amount of gas at station $i$ is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station $i$ to its next station ($i$+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note: The solution is guaranteed to be unique.

### 分析

首先想到的是 $O(N^2)$ 的解法，对每个点进行模拟。

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

### 代码

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

### 相关题目

- 无

## 2.1.22 Candy

### 描述

There are $N$ children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

### 分析

无

### 迭代版

```
// LeetCode, Candy
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    int candy(vector<int> &ratings) {
        const int n = ratings.size();
        vector<int> increment(n);

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

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

### 递归版

```
// LeetCode, Candy
// 备忘录法，时间复杂度 O(n)，空间复杂度 O(n)
// @author fancymouse (http://weibo.com/u/1928162822)
class Solution {
    public:
    int candy(const vector<int>& ratings) {
        vector<int> f(ratings.size());
        int sum = 0;
        for (int i = 0; i < ratings.size(); ++i)
        sum += solve(ratings, f, i);
        return sum;
    }
    int solve(const vector<int>& ratings, vector<int>& f, int i) {
```

```
            if (f[i] == 0) {
                f[i] = 1;
                if (i > 0 && ratings[i] > ratings[i - 1])
                f[i] = max(f[i], solve(ratings, f, i - 1) + 1);
                if (i < ratings.size() - 1 && ratings[i] > ratings[i + 1])
                f[i] = max(f[i], solve(ratings, f, i + 1) + 1);
            }
            return f[i];
        }
    };
```

### 相关题目

- 无

## 2.1.23 Single Number

### 描述

Given an array of integers, every element appears twice except for one. Find that single one.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

### 分析

异或，不仅能处理两次的情况，只要出现偶数次，都可以清零。

### 代码 1

```
// LeetCode, Single Number
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int singleNumber(int A[], int n) {
        int x = 0;
        for (size_t i = 0; i < n; ++i)
        x ^= A[i];
        return x;
    }
};
```

### 代码 2

```
// LeetCode, Single Number
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
```

```
    public:
    int singleNumber(int A[], int n) {
        return accumulate(A, A + n, 0, bit_xor<int>());
    }
};
```

### 相关题目

- Single Number II, 见 §2.1.24

## 2.1.24 Single Number II

### 描述

Given an array of integers, every element appears three times except for one. Find that single one.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

### 分析

本题和上一题 Single Number，考察的是位运算。

方法 1：创建一个长度为 `sizeof(int)` 的数组 `count[sizeof(int)]`，`count[i]` 表示在在 $i$ 位出现的 1 的次数。如果 `count[i]` 是 3 的整数倍，则忽略；否则就把该位取出来组成答案。

方法 2：用 `one` 记录到当前处理的元素为止，二进制 1 出现“1 次”（mod 3 之后的 1）的有哪些二进制位；用 `two` 记录到当前计算的变量为止，二进制 1 出现“2 次”（mod 3 之后的 2）的有哪些二进制位。当 `one` 和 `two` 中的某一位同时为 1 时表示该二进制位上 1 出现了 3 次，此时需要清零。即**用二进制模拟三进制运算**。最终 `one` 记录的是最终结果。

### 代码 1

```
// LeetCode, Single Number II
// 方法 1，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int singleNumber(int A[], int n) {
        const int W = sizeof(int) * 8; // 一个整数的 bit 数，即整数字长
        int count[W];  // count[i] 表示在在 i 位出现的 1 的次数
        fill_n(&count[0], W, 0);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < W; j++) {
                count[j] += (A[i] >> j) & 1;
                count[j] %= 3;
            }
        }
```

```
            int result = 0;
            for (int i = 0; i < W; i++) {
                result += (count[i] << i);
            }
            return result;
        }
    };
```

### 代码 2

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

            return one;
        }
    };
```

### 相关题目

- Single Number, 见 §2.1.23

## 2.1.25 Vector Class

### 描述

Implement a vector-like data structure from scratch.

This question was to be done in C or C++.

Discussion topics: 1. Dealing with out of bounds accesses. 2. What happens when you need to increase the vector's size? 3. How many copies does the structure perform to insert n elements? That is, n calls to vector.push_back

### 2.1.26 N Parking Slots for N-1 Cars Sorting

#### 描述

There are N parking slots and N-1 cars. Everytime you can move one car. How to move these cars into one given order. BTW: I got this question from internet but i could not figure it out partially because the description is kind of incomplete to me. Anyone knowing this question or the solution?

#### 分析

Sorting with O(1) space, e.g., insertsorting, selectsorting

## 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 §??）很类似

#### 代码

```
// LeetCode, Add Two Numbers
// 跟 Add Binary 很类似
// 时间复杂度 O(m+n), 空间复杂度 O(1)
class Solution {
    public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
```

```
        ListNode dummy(-1); // 头节点
        int carry = 0;
        ListNode *prev = &dummy;
        for (ListNode *pa = l1, *pb = l2;
        pa != nullptr || pb != nullptr;
        pa = pa == nullptr ? nullptr : pa->next,
        pb = pb == nullptr ? nullptr : pb->next,
        prev = prev->next) {
            const int ai = pa == nullptr ? 0 : pa->val;
            const int bi = pb == nullptr ? 0 : pb->val;
            const int value = (ai + bi + carry) % 10;
            carry = (ai + bi + carry) / 10;
            prev->next = new ListNode(value); // 尾插法
        }
        if (carry > 0)
        prev->next = new ListNode(carry);
        return dummy.next;
    }
};
```

### 相关题目

- Add Binary, 见 §??

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

```
        ListNode *prev = &dummy;
        for (int i = 0; i < m-1; ++i)
        prev = prev->next;
        ListNode* const head2 = prev;

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

        return dummy.next;
    }
};
```

### 相关题目

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

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

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

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

            return left_dummy.next;
        }
    };
```

### 相关题目

- 无

## 2.2.4 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

### 递归版

```
// LeetCode, Remove Duplicates from Sorted List
// 递归版，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (!head) return head;
        ListNode dummy(head->val + 1); // 值只要跟 head 不同即可
```

```
            dummy.next = head;

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

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

### 迭代版

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

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

### 相关题目

- Remove Duplicates from Sorted List II，见 §2.2.5

### 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

#### 递归版

```
// LeetCode, Remove Duplicates from Sorted List II
// 递归版，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (!head || !head->next) return head;

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

#### 迭代版

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

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

### 相关题目

- Remove Duplicates from Sorted List，见 §2.2.4

## 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
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    ListNode *rotateRight(ListNode *head, int k) {
        if (head == nullptr || k == 0) return head;

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

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

### 相关题目

- 无

## 2.2.7 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
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        ListNode dummy{-1, head};
        ListNode *p = &dummy, *q = &dummy;

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

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

### 相关题目

- 无

## 2.2.8 Swap Nodes in Pairs

### 描述

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

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

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

### 分析

无

### 代码

```
// LeetCode, Swap Nodes in Pairs
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
```

```
    public:
    ListNode *swapPairs(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return head;
        ListNode dummy(-1);
        dummy.next = head;

        for(ListNode *prev = &dummy, *cur = prev->next, *next = cur->next;
        next;
        prev = cur, cur = cur->next, next = cur ? cur->next: nullptr) {
            prev->next = next;
            cur->next = next->next;
            next->next = cur;
        }
        return dummy.next;
    }
};
```

下面这种写法更简洁，但题目规定了不准这样做。

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

        while (p && p->next) {
            swap(p->val, p->next->val);
            p = p->next->next;
        }

        return head;
    }
};
```

### 相关题目

- Reverse Nodes in k-Group, 见 §2.2.9

## 2.2.9 Reverse Nodes in k-Group

### 描述

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

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

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

Only constant memory is allowed.

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

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

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

## 分析

无

## 递归版

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

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

## 迭代版

```
// LeetCode, Reverse Nodes in k-Group
// 迭代版，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    ListNode *reverseKGroup(ListNode *head, int k) {
        if (head == nullptr || head->next == nullptr || k < 2) return head;
```

```
        ListNode dummy(-1);
        dummy.next = head;

        for(ListNode *prev = &dummy, *end = head; end; end = prev->next) {
            for (int i = 1; i < k && end; i++)
            end = end->next;
            if (end  == nullptr) break;  // 不足 k 个

            prev = reverse(prev, prev->next, end);
        }

        return dummy.next;
    }

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

### 相关题目

- Swap Nodes in Pairs, 见 §2.2.8

## 2.2.10 Copy List with Random Pointer

### 描述

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

Return a deep copy of the list.

### 分析

无

### 代码

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

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

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

### 相关题目

- 无

## 2.2.11 Linked List Cycle

### 描述

Given a linked list, determine if it has a cycle in it. Follow up: Can you solve it without using extra space?

### 分析

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

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

### 代码

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

### 相关题目

- Linked List Cycle II, 见 §2.2.12

## 2.2.12 Linked List Cycle II

### 描述

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`. Follow up: Can you solve it without using extra space?

### 分析

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

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

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

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

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

### 代码

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

                while (slow2 != slow) {
                    slow2 = slow2->next;
                    slow = slow->next;
                }
                return slow2;
            }
        }
        return nullptr;
    }
};
```

### 相关题目

- Linked List Cycle, 见 §2.2.11

## 2.2.13 Reorder List

### 描述

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

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

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

## 分析

题目规定要 in-place，也就是说只能使用 $O(1)$ 的空间。

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

## 代码

```
// LeetCode, Reorder List
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    void reorderList(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return;

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

        slow = reverse(slow);

        // merge two lists
        ListNode *curr = head;
        while (curr->next) {
            ListNode *tmp = curr->next;
            curr->next = slow;
            slow = slow->next;
            curr->next->next = tmp;
            curr = tmp;
        }
        curr->next = slow;
    }

    ListNode* reverse(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return head;

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

### 相关题目

- 无

## 2.2.14 LRU Cache

### 描述

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

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

`set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

### 分析

为了使查找、插入和删除都有较高的性能，我们使用一个双向链表 (`std::list`) 和一个哈希表 (`std::unordered_map`)，因为：

- 哈希表保存每个节点的地址，可以基本保证在 $O(1)$ 时间内查找节点
- 双向链表插入和删除效率高，单向链表插入和删除时，还要查找节点的前驱节点

具体实现细节：

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

### 代码

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

```
        }

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

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

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

## 相关题目

- 无

# 第 3 章
# 字符串

## 3.1 字符串 API

面试中经常会出现，现场编写 strcpy, strlen, strstr, atoi 等库函数的题目。这类题目看起来简单，实则难度很大，区分都很高，很容易考察出你的编程功底，是面试官的最爱。

### 3.1.1 strlen

#### 描述

实现 strlen，获取字符串长度，函数原型如下：

```
size_t strlen(const char *str);
```

#### 分析

#### 代码

```
size_t strlen(const char *str) {
    const char *s;
    for (s = str; *s; ++s) {}
    return(s - str);
}
```

### 3.1.2 strcpy

#### 描述

实现 strcpy，字符串拷贝函数，函数原型如下：

```
char* strcpy(char *to, const char *from);
```

### 分析

### 代码

```
char* strcpy(char *to, const char *from) {
    assert(to != NULL && from != NULL);
    char *p = to;
    while ((*p++ = *from++) != '\0')
        ;
    return to;
}
```

## 3.1.3 strstr

### 描述

实现 strstr，子串查找函数，函数原型如下：

```
char * strstr(const char *haystack, const char *needle);
```

### 分析

暴力算法的复杂度是 $O(m*n)$，代码如下。其他算法见第 §3.5节“子串查找”。

### 代码

```
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

与本题相同的题目：

- LeetCode Implement strStr(), http://leetcode.com/oldoj#question_28

与本题相似的题目：

- 无

## 3.1.4 atoi

### 描述

实现 `atoi`，将一个字符串转化为整数，函数原型如下：

```
int atoi(const char *str);
```

### 分析

注意，这题是故意给很少的信息，让你来考虑所有可能的输入。

来看一下 `atoi` 的官方文档 (http://www.cplusplus.com/reference/cstdlib/atoi/)，看看它有什么特性：

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.

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

```
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;
}
```

## 3.1.5 Minimal Phases Covering

We have words and there positions in a paragraph in sorted order. Write an algorithm to find the least distance for a given 3 words. eg. for 3 words job: 5, 9 , 17 in: 4, 13, 18 google: 8, 19, 21 ... ... Answer: 17, 18, 19 Can you extend it to ”n” words?

Context: In Google search results, the search terms are highlighted in the short paragraph that shows up. We need to find the shortest sentence that has all the words if we have word positions as mentioned above.

### 分析

Two solutions:

- Coverte a graph to solve the shortest path
- Coverte a pair array to compress distice and cover all words, e.g.,
  word1: 1, 2, 8, 9

word2: 2 4 5 8

word2: 12 45 75

....

coverte them to a ordered pair array: (1, word1) (2, word1) (2, word2) (4, word2) (5, word2) (5, word2) (8, word1) (8, word2) (9, word1) (12, word3) (45, word3) (75, word3). now our goal is to find a minimum intersect that have all words, and have min(maxNumber-minNumber). we use two pointers to be left and right. When we have a full word list, we will check if Max-Min is smaller, as we have sorted ,so we have Max=right and Min=left. so we have a O(nlgn) time complexity and O(n) space complexity as for save the type.

```
int minPhrases(vector<int> &words){
    if(words.empty()) return 0;
    vector<int> indexs(word.size(),0);
    int mini = 0;
    make_heap();
    int minV, maxV, minDist=INT_MAX;
    for(int &i:indexs){

    }
}
```

### 相关题目

与本题相同的题目：

- LeetCode String to Integer (atoi), http://leetcode.com/oldoj#question_8

与本题相似的题目：

- 无

## 3.2 字符串排序

## 3.3 单词查找树

## 3.4 Makeup Palindrome String

Given a string S, you are allowed to convert it to a palindrome by adding 0 or more characters in front of it. Find the length of the shortest palindrome that you can create from S by applying the above transformation.

Note: O(n) time complexity and O(1) additinoal space.

```
int FindPalindromeSize(string s){
    // n is the number of matching letters
    int n = 0;

    // r is the reverse string navigator
    for (int r = s.length() - 1; r >= 0; --r)
    {
        while(n > 0 && (s[r] != s[n]))
        {
            n--; // Keep subtracting n until we match again or reach the
            beginning again
        }
        // If we have a match, move to the next letter
        if (s[r] == s[n]) { n++; }
    }

    // original string, plus the difference of non-palindrome-y letters
    // s.length() + s.length() - n;
    return (s.length()*2) - n;
}
```

## 3.5 子串查找

字符串的一种基本操作就是**子串查找** (substring search)：给定一个长度为 $N$ 的文本和一个长度为 $M$ 的模式串 (pattern string)，在文本中找到一个与该模式相符的子字符串。

最简单的算法是暴力查找，时间复杂度是 $O(MN)$。下面介绍两个更高效的算法。

### 3.5.1 KMP 算法

KMP 算法是 Knuth、Morris 和 Pratt 在 1976 年发表的。它的基本思想是，当出现不匹配时，就能知晓一部分文本的内容（因为在匹配失败之前它们已经和模式相匹配）。我们可以利用这些信息避免将指针回退到所有这些已知的字符之前。这样，当出现不匹配时，可以提前判断如何重新开始查找，而这种判断只取决于模式本身。

详细解释请参考《算法》[①]第 5.3.3 节。这本书讲的是确定有限状态自动机 (DFA) 的方法。

推荐网上的几篇比较好的博客，讲的是部分匹配表 (partial match table) 的方法（即 next 数组），“字符串匹配的 KMP 算法”http://t.cn/zTOPfdh，图文并茂，非常通俗易懂，作者是阮一峰；“KMP 算法详解”http://www.matrix67.com/blog/archives/115，作者是顾森 Matrix67；”Knuth-Morris-Pratt string matching” http://www.ics.uci.edu/ eppstein/161/960227.html。

使用 next 数组的 KMP 算法的 C 语言实现如下。

kmp.c

```
#include <stdio.h>
#include <stdlib.h>
```

[①]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

```
#include <string.h>

/*
 * @brief 计算部分匹配表，即 next 数组.
 *
 * @param[in] pattern 模式串
 * @param[out] next next 数组
 * @return 无
 */
void compute_prefix(const char *pattern, int next[]) {
    int i;
    int j = -1;
    const int m = strlen(pattern);

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

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

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

    compute_prefix(pattern, next);

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

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

```
        }
    }

    free(next);
    return -1;
}


int main(int argc, char *argv[]) {
    char text[] = "ABC ABCDAB ABCDABCDABDE";
    char pattern[] = "ABCDABD";
    char *ch = text;
    int i = kmp(text, pattern);

    if (i >= 0) printf("matched @: %s\n", ch + i);
    return 0;
}
```

kmp.c

### 3.5.2 Boyer-Moore 算法

详细解释请参考《算法》[①]第 5.3.4 节。

推荐网上的几篇比较好的博客,“字符串匹配的 Boyer-Moore 算法” http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html，图文并茂，非常通俗易懂，作者是阮一峰；Boyer-Moore algorithm, http://www-igm.univ-mlv.fr/ lecroq/string/node14.html。

有兴趣的读者还可以看原始论文[②]。

Boyer-Moore 算法的 C 语言实现如下。

boyer_moore.c

```
/**
 * 本代码参考了 http://www-igm.univ-mlv.fr/~lecroq/string/node14.html
 * 精力有限的话，可以只计算坏字符的后移，好后缀的位移是可选的，因此可以删除
 * suffixes(), pre_gs() 函数
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ASIZE 256  /* ASCII 字母的种类 */

/*
 * @brief 预处理，计算每个字母最靠右的位置.
 *
 * @param[in] pattern 模式串
 * @param[out] right 每个字母最靠右的位置
```

[①] 《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

[②] BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.

```
 * @return 无
 */
static void pre_right(const char *pattern, int right[]) {
    int i;
    const int m = strlen(pattern);

    for (i = 0; i < ASIZE; ++i) right[i] = -1;
    for (i = 0; i < m; ++i) right[(unsigned char)pattern[i]] = i;
}


static void suffixes(const char *pattern, int suff[]) {
    int f, g, i;
    const int m = strlen(pattern);

    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i) {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && pattern[g] == pattern[g + m - 1 - f])
                --g;
            suff[i] = f - g;
        }
    }
}

/*
 * @brief 预处理，计算好后缀的后移位置.
 *
 * @param[in] pattern 模式串
 * @param[out] gs 好后缀的后移位置
 * @return 无
 */
static void pre_gs(const char pattern[], int gs[]) {
    int i, j;
    const int m = strlen(pattern);
    int *suff = (int*)malloc(sizeof(int) * (m + 1));

    suffixes(pattern, suff);

    for (i = 0; i < m; ++i) gs[i] = m;

    j = 0;
```

```
    for (i = m - 1; i >= 0; --i) if (suff[i] == i + 1)
        for (; j < m - 1 - i; ++j) if (gs[j] == m)
            gs[j] = m - 1 - i;
    for (i = 0; i <= m - 2; ++i)
        gs[m - 1 - suff[i]] = m - 1 - i;
    free(suff);
}

/**
 * @brief Boyer-Moore 算法.
 *
 * @param[in] text 文本
 * @param[in] pattern 模式串
 * @return 成功则返回第一次匹配的位置, 失败则返回-1
 */
int boyer_moore(const char *text, const char *pattern) {
    int i, j;
    int right[ASIZE];  /* bad-character shift */
    const int n = strlen(text);
    const int m = strlen(pattern);
    int *gs = (int*)malloc(sizeof(int) * (m + 1));  /* good-suffix shift */

    /* Preprocessing */
    pre_right(pattern, right);
    pre_gs(pattern, gs);

    /* Searching */
    j = 0;
    while (j <= n - m) {
        for (i = m - 1; i >= 0 && pattern[i] == text[i + j]; --i);

        if (i < 0) { /* 找到一个匹配 */
            /* printf("%d ", j);
            j += bmGs[0]; */
            free(gs);
            return j;
        } else {
            const int max = gs[i] > right[(unsigned char)text[i + j]] -
                    m + 1 + i ? gs[i] : i - right[(unsigned char)text[i + j]];
            j += max;
        }
    }
    free(gs);
    return -1;
}


int main() {
```

```
    const char *text="HERE IS A SIMPLE EXAMPLE";
    const char *pattern = "EXAMPLE";
    const int pos = boyer_moore(text, pattern);
    printf("%d\n", pos); /* 17 */
    return 0;
}
```

boyer_moore.c

### 3.5.3 Rabin-Karp 算法

详细解释请参考《算法》[1]第 5.3.5 节。

Rabin-Karp 算法的 C 语言实现如下。

rabin_karp.c

```
#include <stdio.h>
#include <string.h>

const int R = 256;  /** ASCII 字母表的大小，R 进制 */
/** 哈希表的大小，选用一个大素数，这里用 16 位整数范围内最大的素数 */
const long Q = 0xfff1;

/*
 * @brief 哈希函数.
 *
 * @param[in] key 待计算的字符串
 * @param[int] M 字符串的长度
 * @return 长度为 M 的子字符串的哈希值
 */
static long hash(const char key[], const int M) {
    int j;
    long h = 0;
    for (j = 0; j < M; ++j) h = (h * R + key[j]) % Q;
    return h;
}

/*
 * @brief 计算新的 hash.
 *
 * @param[int] h 该段子字符串所对应的哈希值
 * @param[in] first 长度为 M 的子串的第一个字符
 * @param[in] next 长度为 M 的子串的下一个字符
 * @param[int] RM R^(M-1) % Q
 * @return 起始于位置 i+1 的 M 个字符的子字符串所对应的哈希值
 */
static long rehash(const long h, const char first, const char next,
                   const long RM) {
    long newh = (h + Q - RM * first % Q) % Q;
```

[1]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

```
    newh = (newh * R + next) % Q;
    return newh;
}

/*
 * @brief 用蒙特卡洛算法，判断两个字符串是否相等.
 *
 * @param[in] pattern 模式串
 * @param[in] substring 原始文本长度为 M 的子串
 * @return 两个字符串相同，返回 1，否则返回 0
 */
static int check(const char *pattern, const char substring[]) {
    return 1;
}

/**
 * @brief Rabin-Karp 算法.
 *
 * @param[in] text 文本
 * @param[in] n 文本的长度
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
 * @return 成功则返回第一次匹配的位置，失败则返回-1
 */
int rabin_karp(const char *text, const char *pattern) {
    int i;
    const int n = strlen(text);
    const int m = strlen(pattern);
    const long pattern_hash = hash(pattern, m);
    long text_hash = hash(text, m);
    int RM = 1;
    for (i = 0; i < m - 1; ++i) RM = (RM * R) % Q;

    for (i = 0; i <= n - m; ++i) {
        if (text_hash == pattern_hash) {
            if (check(pattern, &text[i])) return i;
        }
        text_hash = rehash(text_hash, text[i], text[i + m], RM);
    }
    return -1;
}


int main() {
    const char *text = "HERE IS A SIMPLE EXAMPLE";
    const char *pattern = "EXAMPLE";
    const int pos = rabin_karp(text, pattern);
    printf("%d\n", pos); /* 17 */
```

```
    return 0;
}
```

rabin_karp.c

### 3.5.4 总结

| 算法 | 版本 | 复杂度 | | 在文本中回退 | 正确性 | 辅助空间 |
|---|---|---|---|---|---|---|
| | | 最坏情况 | 平均情况 | | | |
| KMP 算法 | 完整的 DFA | 2N | 1.1N | 否 | 是 | MR |
| | 部分匹配表 | 3N | 1.1N | 否 | 是 | M |
| | 完整版本 | 3N | N/M | 是 | 是 | R |
| Boyer-Moore 算法 | 坏字符向后位移 | MN | N/M | 是 | 是 | R |
| Rabin-Karp 算法 * | 蒙特卡洛算法 | 7N | 7N | 否 | 是 * | 1 |
| | 拉斯维加斯算法 | $7N^*$ | 7N | 是 | 是 | 1 |

* 概率保证，需要使用均匀和独立的散列函数

## 3.6 正则表达式

### 3.6.1 Same Pattern Match

Given a pattern and a string input - find if the string follows the same pattern and return 0 or 1. Examples: 1) Pattern : ”abba”, input: ”redbluebluered” should return 1. 2) Pattern: ”aaaa”, input: ”asdasdasdasd” should return 1. 3) Pattern: ”aabb”, input: ”xyzabcxzyabc” should return 0.

# 第 4 章
# 栈和队列

栈 (stack) 只能在表的一端插入和删除，先进后出（LIFO, Last In, First Out）。

队列 (queue) 只能在表的一端（队尾 rear）插入，另一端（队头 front）删除，先进先出（FIFO, First In, First Out）。

## 4.1 栈

### 4.1.1 汉诺塔问题

#### 描述

$n$ **阶汉诺塔问题 (Hanoi Tower)** 假设有三个分别命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 $n$ 个直径大小各不相同、从小到大编号为 1，2，...，n 的圆盘，如图 4-1所示。

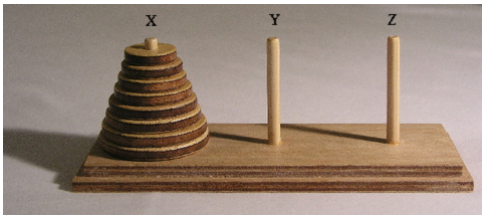



图 4-1　Hanoi 塔问题

现要求将 X 塔上的 n 个圆盘移动到 Z 上并仍按同样的顺序叠放，圆盘移动时必须遵循下列规则：

- 每次只能移动一个圆盘；
- 圆盘可以插在 X、Y 和 Z 中的任一塔座上；
- 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

给出一个数 $n$，求出最少步数的移动序列。

#### 输入

一个整数 $n, n \leq 10$

### 输出

第一行一个整数 $k$，表示最少的移动步数。

接下来 $k$ 行，每行一句话，N from X to Y，表示把 N 号盘从 X 柱移动到 Y 柱。X,Y 属于 `{'A','B','C'}`

### 样例输入

```
3
```

### 样例输出

```
7
1 from A to C
2 from A to B
1 from C to B
3 from A to C
1 from B to A
2 from B to C
1 from A to C
```

### 分析

用递归。

### 代码

hanoi.c

```
#include <stdio.h>

/*
 * @brief 将塔座 x 上按直径有小到大且自上而下编号
 * 为 1 至 n 的 n 个圆盘按规则搬到塔座 z 上，y 可用做辅助塔座.
 * @param[in] n 圆盘个数
 * @param[in] x 源塔座
 * @param[in] y 辅助塔座
 * @param[in] z 目标塔座
 * @return 无
 */
void hanoi(int n, char x, char y, char z) {
    if(n ==  1) {
        /* 将编号为 n 的圆盘从 x 移到 z */
        printf("%d from %c to %c\n", n, x, z);
        return;
    } else {
        /* 将 x 上编号 1 至 n-1 的圆盘移到 y, z 作辅助塔 */
        hanoi(n-1, x, z, y);
```

```
        /* 将编号为 n 的圆盘从 x 移到 z */
        printf("%d from %c to %c\n", n, x, z);
        /* 将 y 上编号 1 至 n-1 的圆盘移到 z, x 作辅助塔 */
        hanoi(n-1, y, x, z);
    }
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%d\n", (1 << n) - 1); /* 总次数 */
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

hanoi.c

### 相关的题目

与本题相同的题目：

- wikioi 3145 汉诺塔游戏, http://www.wikioi.com/problem/3145/

与本题相似的题目：

- 无

## 4.1.2 进制转换

convert_base.cpp

```
#include <stack>
#include <cstdio>

 /**
  * @brief 进制转换，将一个 10 进制整数转化为 d 进制，d<=16.
  * @param[in] n 整数 n
  * @param[in] d d 进制
  * @return 无
  */
void convert_base(int n, const int d) {
    stack<int> s;
    int e;

    while(n != 0) {
        e = n % d;
        s.push(e);
        n /= d;
    }
    while(!s.empty()) {
        e = s.top();
```

```
        s.pop();
        printf("%X", e);
    }
    return;
}

const int MAXN = 64; // 栈的最大长度
int stack[MAXN];
int top = -1;
/**
 * @brief 进制转换，将一个 10 进制整数转化为 d 进制，d<=16，更优化的版本.
 *
 * 如果可以预估栈的最大空间，则用数组来模拟栈，这时常用的一个技巧。
 * 这里，栈的最大长度是多少？假设 CPU 是 64 位，最大的整数则是 2^64，由于
 * 数制最小为 2，在这个进制下，数的位数最长，这就是栈的最大长度，最长为 64。
 *
 * @param[in] n 整数 n
 * @param[in] d d 进制
 * @return 无
 */
void convert_base2(int n, const int d) {
    int e;

    while(n != 0) {
        e = n % d;
        stack[++top] = e; // push
        n /= d;
    }
    while(top >= 0) {
        e = stack[top--]; // pop
        printf("%X", e);
    }
    return;
}


/**
 * @brief 进制转换，将一个 d 进制整数转化为 10 进制，d<=16.
 * @param[in] s d 进制整数
 * @param[in] d d 进制
 * @return 10 进制整数
 */
int restore(const char s[MAXN], const int d) {
    int result = 0;
    int one;

    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] >= '0' && s[i] <= '9') one = s[i] - '0';
```

```
        else if (s[i] >= 'A' && s[i] <= 'F') one = s[i] - 'A' + 10;
        else one = s[i] - 'a' + 10; /* (s[i] >= 'a' && s[i] <= 'f') */

        result = result * d + one;
    }
    return result;
}
```

convert_base.cpp

### 4.1.3 Design Queue by Stack

Design a queue (FIFO) structure using only stacks (LIFO).

Code is not necessary.

Follow-up: provide a complexity analysis of push and remove operations.

## 4.2 队列

### 4.2.1 打印杨辉三角

yanghui_triangle.cpp

```
#include <queue>
/**
 * @brief 打印杨辉三角系数.
 *
 * 分行打印二项式 (a+b)^n 展开式的系数。在输出上一行
 * 系数的同时，将其下一行的系数预先计算好，放入队列中。
 * 在各行系数之间插入一个 0。
 *
 * @param[in] n (a+b)^n
 * @return 无
 */
void yanghui_triangle(const int n) {
    queue<int> q;
    /* 预先放入第一行的 1 */
    q.push(1);

    for(int i = 0; i <= n; i++) {     /* 逐行处理 */
        int s = 0;
        q.push(s);      /* 在各行间插入一个 0*/

        /* 处理第 i 行的 i+2 个系数（包括一个 0）*/
        for(int j = 0; j < i+2; j++) {
            int t;
            int tmp;
            t = q.front();  /* 读取一个系数，放入 t*/
            q.pop();
```

```
            tmp = s + t;       /* 计算下一行系数，并进队列 */
            q.push(tmp);
            s = t;             /* 打印一个系数，第 i+2 个是 0*/
            if(j != i+1) {
                printf("%d ",s);
            }
        }
        printf("\n");
    }
}
```

yanghui_triangle.cpp

# 第 5 章
# 树

## 5.1 BST

Given a BST and a number x, check whether exists two nodes in the BST whose sum equals to x. You can not use one extra array to serialize the BST and do a 2sum solver on it.

### 相关题目

- 2Sum, 见 §2.1.7
- 3Sum, 见 §2.1.8
- 3Sum Closest, 见 §2.1.9
- 4Sum, 见 §2.1.10

## 5.2 二叉树的遍历

在中序遍历中，一个节点的前驱，是其左子树的最右下角结点，后继，是其右子树的最左下角结点。

在后序遍历中，

- 若结点是根结点，则其后继为空；
- 若结点是双亲的右子树，或是左子树但双亲无右子树，则其后继为双亲结点；
- 若结点是双亲的左子树且双亲有右子树，则其后继为右子树按后序遍历的第一个结点

binary_tree.cpp

```
#include <iostream>
#include <stack>
#include <queue>

/** 结点的数据 */
typedef int tree_node_elem_t;
 /*
  *@struct
  *@brief 二叉树结点
```

```
  */
struct binary_tree_node_t {
    binary_tree_node_t *left;   /* 左孩子 */
    binary_tree_node_t *right;   /* 右孩子 */
    tree_node_elem_t elem; /* 结点的数据 */
};

/**
  * @brief 先序遍历, 递归.
  * @param[in] root 根结点
  * @param[in] visit 访问数据元素的函数指针
  * @return 无
  */
void pre_order_r(const binary_tree_node_t *root,
                 int (*visit)(const binary_tree_node_t*)) {
    if (root == nullptr) return;

    visit(root);
    pre_order_r(root->left, visit);
    pre_order_r(root->right, visit);
}

/**
  * @brief 中序遍历, 递归.
  */
void in_order_r(const binary_tree_node_t *root,
                int (*visit)(const binary_tree_node_t*)) {
    if(root == nullptr) return;

    in_order_r(root->left, visit);
    visit(root);
    in_order_r(root->right, visit);
}

/**
  * @brief 后序遍历, 递归.
  */
void post_order_r(const binary_tree_node_t *root,
                  int (*visit)(const binary_tree_node_t*)) {
    if(root == nullptr) return;

    post_order_r(root->left, visit);
    post_order_r(root->right, visit);
    visit(root);
}

/**
 * @brief 先序遍历, 非递归.
```

```
 */
void pre_order(const binary_tree_node_t *root,
               int (*visit)(const binary_tree_node_t*)) {
    const binary_tree_node_t *p;
    stack<const binary_tree_node_t *> s;

    p = root;

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

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

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

/**
 * @brief 中序遍历, 非递归.
 */
void in_order(const binary_tree_node_t *root,
              int (*visit)(const binary_tree_node_t*)) {
    const binary_tree_node_t *p;
    stack<const binary_tree_node_t *> s;

    p = root;

    while(!s.empty() || p!=nullptr) {
        if(p != nullptr) {
            s.push(p);
            p = p->left;
        } else {
            p = s.top();
            s.pop();
            visit(p);
            p = p->right;
        }
    }
}

/**
 * @brief 后序遍历, 非递归.
 */
void post_order(const binary_tree_node_t *root,
                int (*visit)(const binary_tree_node_t*)) {
```

```
    /* p，正在访问的结点，q，刚刚访问过的结点 */
    const binary_tree_node_t *p, *q;
    stack<const binary_tree_node_t *> s;

    p = root;

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

/**
 * @brief 层次遍历，也即 BFS.
 *
 * 跟先序遍历一模一样，唯一的不同是栈换成了队列
 */
void level_order(const binary_tree_node_t *root,
                 int (*visit)(const binary_tree_node_t*)) {
    const binary_tree_node_t *p;
    queue<const binary_tree_node_t *> q;

    p = root;

    if(p != nullptr) q.push(p);

    while(!q.empty()) {
        p = q.front();
        q.pop();
        visit(p);
```

```
        /* 先左后右或先右后左无所谓 */
        if(p->left != nullptr) q.push(p->left);
        if(p->right != nullptr) q.push(p->right);
    }
}
```

binary_tree.cpp

## 5.3 线索二叉树

二叉树中存在很多空指针，可以利用这些空指针，指向其前驱或者后继。这种利用起来的空指针称为线索，这种改进后的二叉树称为线索二叉树 (threaded binary tree)。

一棵 n 个结点的二叉树含有 n+1 个空指针。这是因为，假设叶子节点数为 $n_0$，度为 1 的节点数为 $n_1$，度为 2 的节点数为 $n_2$，每个叶子节点有 2 个空指针，每个度为 1 的节点有 1 个空指针，则空指针的总数为 $2n_0 + n_1$，又有 $n_0 = n_2 + 1$（留给读者证明），因此空指针总数为 $2n_0 + n_1 = n_0 + n_2 + 1 + n_1 = n_0 + n_1 + n_2 + 1 = n + 1$。

在二叉树线索化过程中，通常规定，若无左子树，令 left 指向前驱，若无右子树，令 rchild 指向后继。还需要增加两个标志域表示当前指针是不是线索，例如 ltag=1，表示 left 指向的是前驱，ltag=0，表示 left 指向的是左孩子，rtag 类似。

二叉树的线索化，实质上就是遍历一棵树，只是在遍历的过程中，检查当前节点的左右指针是否为空，若为空，将它们改为指向前驱或后继的线索。

以中序线索二叉树为例，指针 pre 表示前驱，succ 表示后继，如图 5-1所示。

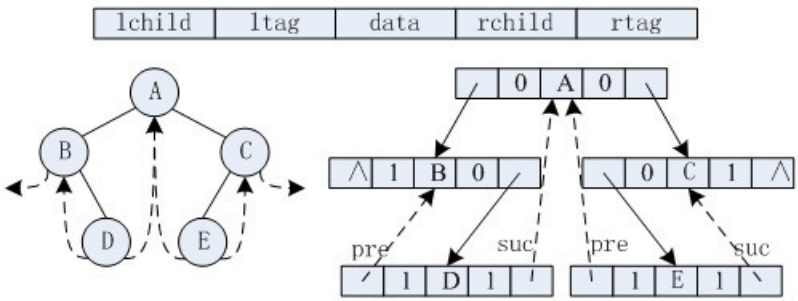



图 5-1 中序线索二叉树

在中序线索二叉树中，一个节点的前驱，是其左子树的最右下角结点，后继，是其右子树的最左下角结点。

中序线索二叉树的 C 语言实现如下。

theaded_binary_tree.c

```
/** @file threaded_binary_tree.c
  * @brief 线索二叉树.
  */
```

```
#include <stddef.h>    /* for NULL */
#include <stdio.h>

/* 结点数据的类型. */
typedef int elem_t;

 /**
  *@struct
  *@brief 线索二叉树结点.
  */
typedef struct tbt_node_t {
    int ltag; /** 1 表示是线索, 0 表示是孩子 */
    int rtag; /** 1 表示是线索, 0 表示是孩子 */
    struct tbt_node_t *left; /** 左孩子 */
    struct tbt_node_t *right; /** 右孩子 */
    elem_t elem; /** 结点所存放的数据 */
}tbt_node_t;

/* 内部函数 */
static void in_thread(tbt_node_t *p, tbt_node_t **pre);
static tbt_node_t *first(tbt_node_t *p);
static tbt_node_t *next(const tbt_node_t *p);

 /**
  * @brief 建立中序线索二叉树.
  * @param[in] root 树根
  * @return 无
  */
void create_in_thread(tbt_node_t *root) {
    /* 前驱结点指针 */
    tbt_node_t *pre=NULL;
    if(root != NULL) { /* 非空二叉树, 线索化 */
        /* 中序遍历线索化二叉树 */
        in_thread(root, &pre);
        /* 处理中序最后一个结点 */
        pre->right = NULL;
        pre->rtag = 1;
    }
}


/**
  * @brief 在中序线索二叉树上执行中序遍历.
  * @param[in] root 树根
  * @param[in] visit 访问结点的数据的函数
  * @return 无
  */
void in_order(tbt_node_t *root, int(*visit)(tbt_node_t*)) {
```

```
    tbt_node_t *p;
    for(p = first(root); p != NULL; p = next(p)) {
        visit(p);
    }
}


 /*
  * @brief 中序线索化二叉树的主过程.
  * @param[in] p 当前要处理的结点
  * @param[inout] pre 当前结点的前驱结点
  * @return 无
  */
static void in_thread(tbt_node_t *p, tbt_node_t **pre) {
    if(p != NULL) {
        in_thread(p->left, pre); /* 线索化左子树 */
        if(p->left == NULL) {  /* 左子树为空, 建立前驱 */
            p->left = *pre;
            p->ltag = 1;
        }
        /* 建立前驱结点的后继线索 */
        if((*pre) != NULL &&
            (*pre)->right == NULL) {
            (*pre)->right = p;
            (*pre)->rtag = 1;
        }
        *pre = p; /* 更新前驱 */
        in_thread(p->right, pre); /* 线索化右子树 */
    }
}

 /*
  * @brief 寻找线索二叉树的中序下的第一个结点.
  * @param[in] p 线索二叉树中的任意一个结点
  * @return 此线索二叉树的第一个结点
  */
static tbt_node_t *first(tbt_node_t *p) {
    if(p == NULL)  return NULL;

    while(p->ltag == 0) {
        p = p->left;  /* 最左下结点, 不一定是叶结点 */
    }
    return p;
}

 /*
  * @brief 求中序线索二叉树中某结点的后继.
  * @param[in] p 某结点
```

```
   * @return p 的后继
   */
 static tbt_node_t *next(const tbt_node_t *p) {
     if(p->rtag == 0) {
         return first(p->right);
     } else {
         return p->right;
     }
 }
```

theaded_binary_tree.c

中序线索二叉树最简单，在中序线索的基础上稍加修改就可以实现先序，后续就要再费点心思了。

## 5.4 Morris Traversal

通过前面第 §5.2节，我们知道，实现二叉树的前序（preorder）、中序（inorder）、后序（postorder）遍历有两个常用的方法，一是递归 (recursive)，二是栈 (stack+iterative)。这两种方法都是 O(n) 的空间复杂度。

而 Morris Traversal 只需要 O(1) 的空间复杂度。这种算法跟线索二叉树很像，不过 Morris Traversal 一边建线索，一边访问数据，访问完后销毁线索，保持二叉树不变。

### 5.4.1 Morris 中序遍历

Morris 中序遍历的步骤如下：

1. 初始化当前节点 cur 为 root 节点
2. 如果 cur 没有左孩子，则输出当前节点并将其右孩子作为当前节点，即 cur = cur->rchild。
3. 如果 cur 有左孩子，则寻找 cur 的前驱，即 cur 的左子树的最右下角结点。
   a) 如果前驱节点的右孩子为空，将它的右孩子指向当前节点，当前节点更新为当前节点的左孩子。
   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状），输出当前节点，当前节点更新为当前节点的右孩子。
4. 重复 2、3 步骤，直到 cur 为空。

如图 5-2所示，cur 表示当前节点，深色节点表示该节点已输出。

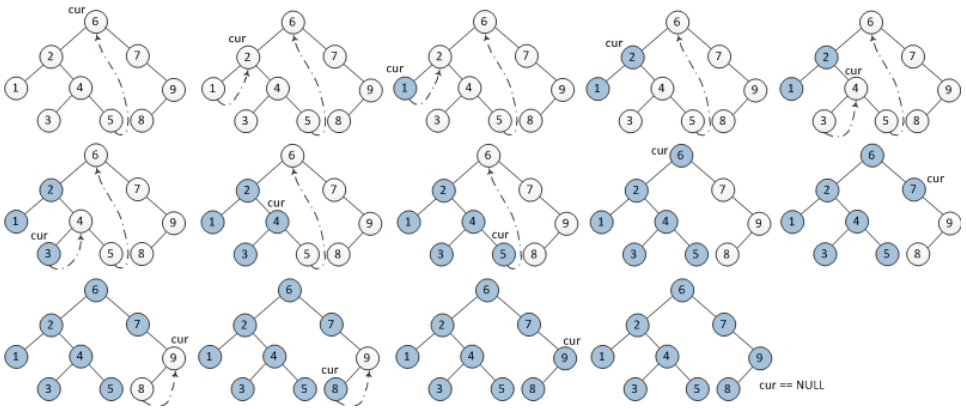



图 5-2 Morris 中序遍历

C 语言实现见第 §5.4.4节。

### 相关的题目

- Leet Code - Binary Tree Inorder Traversal, http://leetcode.com/onlinejudge#question_94

## 5.4.2 Morris 先序遍历

Morris 先序遍历的步骤如下：

1. 初始化当前节点 cur 为 root 节点
2. 如果 cur 没有左孩子，则输出当前节点并将其右孩子作为当前节点，即 cur = cur->rchild。
3. 如果 cur 有左孩子，则寻找 cur 的前驱，即 cur 的左子树的最右下角结点。
   a) 如果前驱节点的右孩子为空，将它的右孩子指向当前节点，**输出当前节点（在这里输出，这是与中序遍历唯一的不同点）**当前节点更新为当前节点的左孩子。
   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状），~~输出当前节点，~~当前节点更新为当前节点的右孩子。
4. 重复 2、3 步骤，直到 cur 为空。

如图 5-3所示。

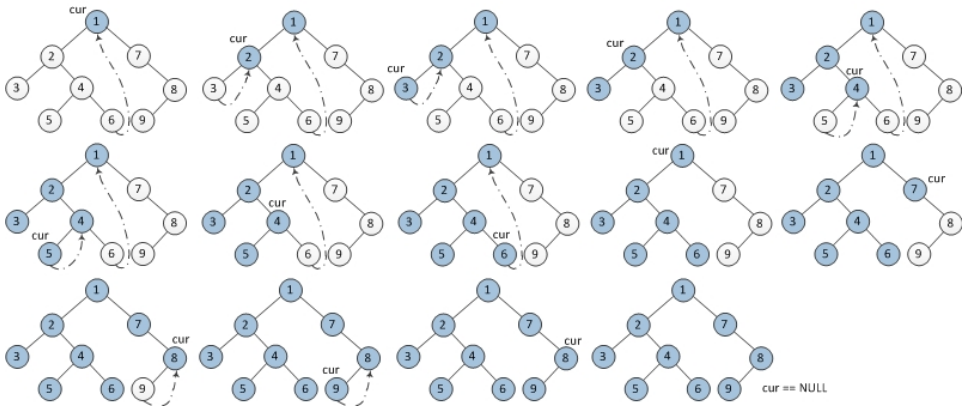



图 5-3 Morris 先序遍历

C 语言实现见第 §5.4.4节。

### 5.4.3 Morris 后序遍历

Morris 后续遍历稍微复杂，需要建立一个临时节点 dump，令其左孩子是 root，并且还需要一个子过程，就是倒序输出某两个节点之间路径上的所有节点。

Morris 后序遍历的步骤如下：

1. 初始化当前节点 cur 为 root 节点
2. 如果 cur 没有左孩子，则~~输出当前节点并~~将其右孩子作为当前节点，即 cur = cur->rchild。
3. 如果 cur 有左孩子，则寻找 cur 的前驱，即 cur 的左子树的最右下角结点。
   a) 如果前驱节点的右孩子为空，将它的右孩子指向当前节点，当前节点更新为当前节点的左孩子。
   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状），~~输出当前节点，~~**倒序输出从当前节点的左孩子到该前驱节点这条路径上的所有节点。**当前节点更新为当前节点的右孩子。
4. 重复 2、3 步骤，直到 cur 为空。

如图 5-4所示。

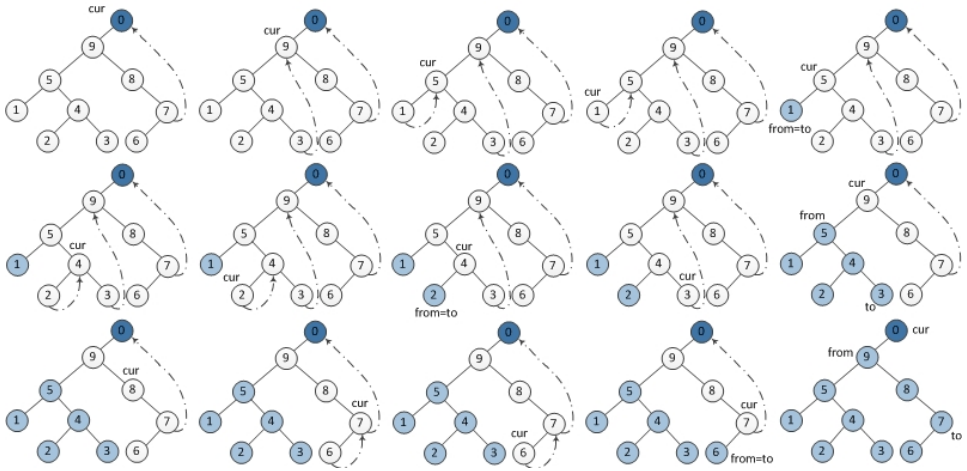



图 5-4 Morris 后序遍历

C 语言实现见第 §5.4.4节。

### 5.4.4 C 语言实现

morris_traversal.c

```
/** @file morris_traversal.c
 * @brief Morris 遍历算法.
 */
#include<stdio.h>
#include<stdlib.h>

/* 结点数据的类型. */
typedef int elem_t;

/**
 *@struct
 *@brief 二叉树结点.
 */
typedef struct bt_node_t {
    elem_t elem; /* 节点的数据 */
    struct bt_node_t *left; /* 左孩子 */
    struct bt_node_t *right; /* 右孩子 */
} bt_node_t;

/**
 * @brief 中序遍历, Morris 算法.
 * @param[in] root 根节点
 * @param[in] visit 访问函数
 * @return 无
```

```
 */
void in_order_morris(bt_node_t *root, int(*visit)(bt_node_t*)) {
    bt_node_t *cur, *prev;

    cur = root;
    while (cur != NULL ) {
        if (cur->left == NULL ) {
            visit(cur);
            prev = cur;
            cur = cur->right;
        } else {
            /* 查找前驱 */
            bt_node_t *node = cur->left;
            while (node->right != NULL && node->right != cur)
                node = node->right;

            if (node->right == NULL ) { /* 还没线索化，则建立线索 */
                node->right = cur;
                /* prev = cur; 不能有这句，cur 还没有被访问 */
                cur = cur->left;
            } else {    /* 已经线索化，则访问节点，并删除线索  */
                visit(cur);
                node->right = NULL;
                prev = cur;
                cur = cur->right;
            }
        }
    }
}

/**
 * @brief 先序遍历，Morris 算法.
 * @param[in] root 根节点
 * @param[in] visit 访问函数
 * @return 无
 */
void pre_order_morris(bt_node_t *root, int (*visit)(bt_node_t*)) {
    bt_node_t *cur, *prev;

    cur = root;
    while (cur != NULL ) {
        if (cur->left == NULL ) {
            visit(cur);
            prev = cur; /* cur 刚刚被访问过 */
            cur = cur->right;
        } else {
            /* 查找前驱 */
            bt_node_t *node = cur->left;
```

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

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


static void reverse(bt_node_t *from, bt_node_t *to);
static void visit_reverse(bt_node_t* from, bt_node_t *to,
        int (*visit)(bt_node_t*));
/**
 * @brief 后序遍历，Morris 算法.
 * @param[in] root 根节点
 * @param[in] visit 访问函数
 * @return 无
 */
void post_order_morris(bt_node_t *root, int (*visit)(bt_node_t*)) {
    bt_node_t dummy = { 0, NULL, NULL };
    bt_node_t *cur, *prev = NULL;

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

            if (node->right == NULL ) { /* 还没线索化，则建立线索 */
                node->right = cur;
                prev = cur; /* 必须要有 */
                cur = cur->left;
            } else { /* 已经线索化，则访问节点，并删除线索  */
                visit_reverse(cur->left, prev, visit);  // call print
```

```
                prev->right = NULL;
                prev = cur; /* 必须要有 */
                cur = cur->right;
            }
        }
    }
}

/*
 * @brief 逆转路径.
 * @param[in] from from
 * @param[to] to to
 * @return 无
 */
static void reverse(bt_node_t *from, bt_node_t *to) {
    bt_node_t *x = from, *y = from->right, *z;
    if (from == to) return;

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

/*
 * @brief  访问逆转后的路径上的所有结点.
 * @param[in] from from
 * @param[to] to to
 * @return 无
 */
static void visit_reverse(bt_node_t* from, bt_node_t *to,
        int (*visit)(bt_node_t*)) {
    bt_node_t *p = to;
    reverse(from, to);

    while (1) {
        visit(p);
        if (p == from)
            break;
        p = p->right;
    }

    reverse(to, from);
}

/*
```

```
 * @brief 分配一个新节点.
 * @param[in] e 新节点的数据
 * @return 新节点
 */
bt_node_t* new_node(int e) {
    bt_node_t* node = (bt_node_t*) malloc(sizeof(bt_node_t));
    node->elem = e;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

static int print(bt_node_t *node) {
    printf(" %d ", node->elem);
    return 0;
}

/* test */
int main() {
    /* 构造的二叉树如下
       1
     /   \
    2      3
  /  \
4     5
     */
    bt_node_t *root = new_node(1);
    root->left = new_node(2);
    root->right = new_node(3);
    root->left->left = new_node(4);
    root->left->right = new_node(5);

    in_order_morris(root, print);
    printf("\n");
    pre_order_morris(root, print);
    printf("\n");
    post_order_morris(root, print);
    printf("\n");

    return 0;
}
```

morris_traversal.c

## 5.5 重建二叉树

binary_tree_rebuild.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
/**
 * @brief 给定前序遍历和中序遍历，输出后序遍历.
 *
 * @param[in] pre 前序遍历的序列
 * @param[in] in 中序遍历的序列
 * @param[in] n 序列的长度
 * @param[out] post 后续遍历的序列
 * @return 无
 */
void build_post(const char * pre, const char *in, const int n, char *post) {
    int left_len = strchr(in, pre[0]) - in;
    if(n <= 0) return;

    build_post(pre + 1, in, left_len, post);
    build_post(pre + left_len + 1, in + left_len + 1,
            n - left_len - 1, post + left_len);
    post[n - 1] = pre[0];
}

#define MAX  64
// 测试
// BCAD CBAD，输出 CDAB
// DBACEGF ABCDEFG，输出 ACBFGED
void build_post_test() {
    char pre[MAX] = {0};
    char in[MAX] = {0};
    char post[MAX] = {0};
    int n;

    scanf("%s%s", pre, in);
    n = strlen(pre);

    build_post(pre, in, n, post);
    printf("%s\n", post);
}

/* 结点数据的类型. */
typedef char elem_t;

/**
 *@struct
 *@brief 二叉树结点.
 */
typedef struct bt_node_t {
```

```
    elem_t elem; /* 节点的数据 */
    struct bt_node_t *left; /* 左孩子 */
    struct bt_node_t *right; /* 右孩子 */
} bt_node_t;

/**
 * @brief 给定前序遍历和中序遍历，重建二叉树.
 *
 * @param[in] pre 前序遍历的序列
 * @param[in] in 中序遍历的序列
 * @param[in] n 序列的长度
 * @param[out] root 根节点
 * @return 无
 */
void rebuild(const char *pre, const char *in, int n, bt_node_t **root) {
    int left_len;
    // 检查终止条件
    if (n <= 0 || pre == NULL || in == NULL)
        return;
    //获得前序遍历的第一个结点
    *root = (bt_node_t*) malloc(sizeof(bt_node_t));
    (*root)->elem = *pre;
    (*root)->left = NULL;
    (*root)->right = NULL;

    left_len = strchr(in, pre[0]) - in;
    //重建左子树
    rebuild(pre + 1, in, left_len, &((*root)->left));
    //重建右子树
    rebuild(pre + left_len + 1, in + left_len + 1, n - left_len - 1,
            &((*root)->right));
}

void print_post_order(const bt_node_t *root) {
    if(root != NULL) {
        print_post_order(root->left);
        print_post_order(root->right);
        printf("%c", root->elem);
    }
}

void rebuild_test() {
    char pre[MAX] = { 0 };
    char in[MAX] = { 0 };
    int n;
    bt_node_t *root;
    scanf("%s%s", pre, in);
    n = strlen(pre);
```

```
    rebuild(pre, in, n, &root);
    print_post_order(root);
}

int main() {
    build_post_test();
    rebuild_test();
    return 0;
}
```

binary_tree_rebuild.c

## 5.6 堆

### 5.6.1 原理和实现

C++ 可以直接使用 `priority_queue`。

heap.c

```
/** @file heap.c
 * @brief 堆, 默认为小根堆, 即堆顶为最小.
 * @author soulmachine@gmail.com
 */
#include <stdlib.h>  /* for malloc() */
#include <string.h>  /* for memcpy() */

typedef int heap_elem_t; // 元素的类型

/**
 * @struct
 * @brief 堆的结构体
 */
typedef struct heap_t {
    int     size;   /** 实际元素个数 */
    int     capacity; /** 容量, 以元素为单位 */
    heap_elem_t  *elems;   /** 堆的数组 */
    int (*cmp)(const heap_elem_t*, const heap_elem_t*);   /** 元素的比较函数 */
}heap_t;


/** 基本类型 (如 int, long, float, double) 的比较函数 */
int cmp_int(const int *x, const int *y) {
    const int sub = *x - *y;
    if(sub > 0) {
        return 1;
    } else if(sub < 0) {
        return -1;
    } else {
```

```
        return 0;
    }
}

/**
 * @brief 创建堆.
 * @param[out] capacity 初始容量
 * @param[in] cmp cmp 比较函数, 小于返回-1, 等于返回 0
 *            大于返回 1, 反过来则是大根堆
 * @return 成功返回堆对象的指针, 失败返回 NULL
 */
heap_t* heap_create(const int capacity,
        int (*cmp)(const heap_elem_t*, const heap_elem_t*)) {
    heap_t *h = (heap_t*)malloc(sizeof(heap_t));
    h->size = 0;
    h->capacity = capacity;
    h->elems = (heap_elem_t*)malloc(capacity * sizeof(heap_elem_t));
    h->cmp = cmp;

    return h;
}

/**
 * @brief 销毁堆.
 * @param[inout] h 堆对象的指针
 * @return 无
 */
void heap_destroy(heap_t *h) {
    free(h->elems);
    free(h);
}


/**
 * @brief 判断堆是否为空.
 * @param[in] h 堆对象的指针
 * @return 是空, 返回 1, 否则返回 0
 */
int heap_empty(const heap_t *h) {
    return h->size == 0;
}

/**
 * @brief 获取元素个数.
 * @param[in] s 堆对象的指针
 * @return 元素个数
 */
int heap_size(const heap_t *h) {
```

```
    return h->size;
}

/*
 * @brief 小根堆的自上向下筛选算法.
 * @param[in] h 堆对象的指针
 * @param[in] start 开始结点
 * @return 无
 */
void heap_sift_down(const heap_t *h, const int start) {
    int i = start;
    int j;
    const heap_elem_t tmp = h->elems[start];

    for(j = 2 * i + 1; j < h->size; j = 2 * j + 1) {
        if(j < (h->size - 1) &&
            // h->elems[j] > h->elems[j + 1]
            h->cmp(&(h->elems[j]), &(h->elems[j + 1])) > 0) {
                j++; /* j 指向两子女中小者 */
        }
        // tmp <= h->data[j]
        if(h->cmp(&tmp, &(h->elems[j])) <= 0) {
            break;
        } else {
            h->elems[i] = h->elems[j];
            i = j;
        }
    }
    h->elems[i] = tmp;
}

/*
 * @brief 小根堆的自下向上筛选算法.
 * @param[in] h 堆对象的指针
 * @param[in] start 开始结点
 * @return 无
 */
void heap_sift_up(const heap_t *h, const int start) {
    int j = start;
    int i= (j - 1) / 2;
    const heap_elem_t tmp = h->elems[start];

    while(j > 0) {
        // h->data[i] <= tmp
        if(h->cmp(&(h->elems[i]), &tmp) <= 0) {
            break;
        } else {
            h->elems[j] = h->elems[i];
```

```
            j = i;
            i = (i - 1) / 2;
        }
    }
    h->elems[j] = tmp;
}

/**
 * @brief 添加一个元素.
 * @param[in] h 堆对象的指针
 * @param[in] x 要添加的元素
 * @return 无
 */
void heap_push(heap_t *h, const heap_elem_t x) {
    if(h->size == h->capacity) { /* 已满, 重新分配内存 */
        heap_elem_t* tmp =
            (heap_elem_t*)realloc(h->elems, h->capacity * 2 * sizeof(heap_elem_t));
        h->elems = tmp;
        h->capacity *= 2;
    }

    h->elems[h->size] = x;
    h->size++;

    heap_sift_up(h, h->size - 1);
}

/**
 * @brief 弹出堆顶元素.
 * @param[in] h 堆对象的指针
 * @return 无
 */
void heap_pop(heap_t *h) {
    h->elems[0] = h->elems[h->size - 1];
    h->size --;
    heap_sift_down(h, 0);
}

/**
 * @brief 获取堆顶元素.
 * @param[in] h 堆对象的指针
 * @return 堆顶元素
 */
heap_elem_t heap_top(const heap_t *h) {
    return h->elems[0];
}
```

heap.c

### 5.6.2 最小的 N 个和

#### 描述

有两个长度为 $N$ 的序列 A 和 B，在 A 和 B 中各任取一个数可以得到 $N^2$ 个和，求这 $N^2$ 个和中最小的 $N$ 个。

#### 输入

第一行输入一个正整数 $N$；第二行 N 个整数 $A_i$ 且 $A_i \leq 10^9$；第三行 $N$ 个整数 $B_i$，且 $Bi \leq 10^9$。

#### 输出

输出仅一行，包含 $N$ 个整数，从小到大输出这 $N$ 个最小的和，相邻数字之间用空格隔开。

#### 样例输入

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

#### 样例输出

```
2 3 4 4 5
```

#### 分析

由于数据太大，有 $N^2$ 个和，不能通过先求和再排序的方式来求解，这个时候就要用到堆了。

首先将 A，B 两数组排序，我们可以建立这样一个有序表：

$$\begin{aligned} A_1+B_1 < A_1+B_2 < A_1+B_3 < \quad ... \quad < A_1+B_N \\ A_2+B_1 < A_2+B_2 < A_2+B_3 < \quad ... \quad < A_2+B_N \\ \quad ... \\ A_N+B_1 < A_N+B_2 < A_N+B_3 < \quad ... \quad < A_N+B_N \end{aligned}$$

首先将 `A[i] + B[0]` 压入堆中，设每次出堆的元素为 `sum=A[a]+B[b]`，则将 `A[a]+B[b+1]` 入堆，这样可以保证前 $N$ 个出堆的元素为最小的前 $N$ 项。在实现的时候，可以不用保存 B 数组的下标，通过 `sum-B[b]+B[b+1]` 来替换 `A[a]+B[b+1]` 来节省空间。

#### 代码

sequence.cpp

```
/* wikioi 1245 最小的 N 个和，http://www.wikioi.com/problem/1245/  */
#include <cstdio>
```

```
#include <queue>
#include <algorithm>

const int MAXN = 100000;

int N;
int a[MAXN], b[MAXN];

typedef struct node_t {
    int sum;
    int b; /* sum=a[i]+b[b] */
    bool operator>(const node_t &other) const {
        return sum > other.sum;
    }
} node_t;


void k_merge() {
    sort(a, a+N);
    sort(b, b+N);
    priority_queue<node_t, vector<node_t>,
                                greater<node_t> > q;

    for (int i = 0; i < N; i++) {
        node_t tmp;
        tmp.sum = a[i]+b[0];
        tmp.b = 0;
        q.push(tmp);
    }

    for (int i = 0; i < N; i++) {
        node_t tmp = q.top(); q.pop();
        printf("%d ", tmp.sum);
        tmp.sum = tmp.sum - b[tmp.b] + b[tmp.b + 1];
        tmp.b++;
        q.push(tmp);
    }

    return;
}

int main() {
    scanf("%d", &N);
    for (int i = 0; i < N; i++) {
        scanf("%d", &a[i]);
    }
    for (int i = 0; i < N; i++) {
        scanf("%d", &b[i]);
```

```
    }

    k_merge();
    return 0;
}
```

sequence.cpp

### 相关的题目

与本题相同的题目：

- wikioi 1245 最小的 N 个和, http://www.wikioi.com/problem/1245/

与本题相似的题目：

- POJ 2442 Sequence, http://poj.org/problem?id=2442

## 5.7 并查集

### 5.7.1 原理和实现

通常用树双亲表示作为并查集的存储结构。每个集合以一棵树表示，数组元素的下标代表元素名，根结点的双亲指针为一个负数，表示集合的元素的个数。如图 5-5、图 5-6和图 5-7所示。

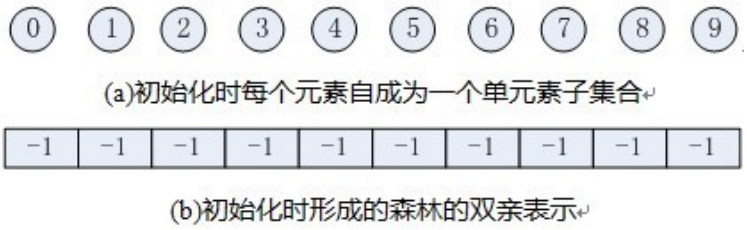



图 5-5 并查集的初始化

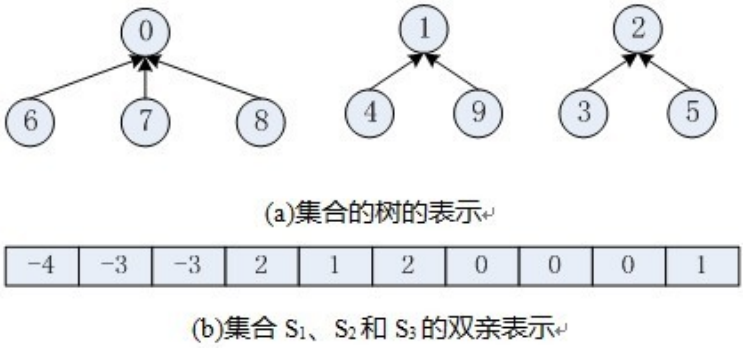



图 5-6 用树表示并查集

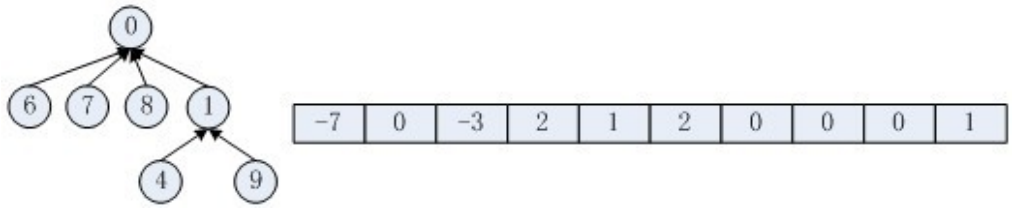



图 5-7 两个集合的并

并查集的 C 语言实现如下。

ufs.c

```
#include <stdlib.h>

/** 并查集. */
typedef struct ufs_t {
    int *p;     /** 树的双亲表示法 */
    int size;   /** 大小. */
} ufs_t;

/**
 * @brief 创建并查集.
 * @param[in] n 数组的容量
 * @return 并查集
 */
ufs_t* ufs_create(int n) {
    ufs_t *ufs = (ufs_t*)malloc(sizeof(ufs_t));
    int i;
    ufs->p = (int*)malloc(n * sizeof(int));
    for(i = 0; i < n; i++)
        ufs->p[i] = -1;
    return ufs;
}

/**
 * @brief 销毁并查集.
 * @param[in] ufs 并查集
 * @return 无
 */
void ufs_destroy(ufs_t *ufs) {
    free(ufs->p);
    free(ufs);
}

/**
 * @brief Find 操作, 带路径压缩, 递归版.
 * @param[in] s 并查集
 * @param[in] x 要查找的元素
```

```
 * @return 包含元素 x 的树的根
 */
int ufs_find(ufs_t *ufs, int x) {
    if (ufs->p[x] < 0) return x; // 终止条件

    return ufs->p[x] = ufs_find(ufs, ufs->p[x]); /* 回溯时的压缩路径 */
}

/** Find 操作，朴素版，deprecated. */
static int ufs_find_naive(ufs_t *ufs, int x) {
    while (ufs->p[x] >= 0) {
        x = ufs->p[x];
    }
    return x;
}

/** Find 操作，带路径压缩，迭代版. */
static int ufs_find_iterative(ufs_t *ufs, int x) {
    int oldx = x; /* 记录原始 x */
    while (ufs->p[x] >= 0) {
        x = ufs->p[x];
    }
    while (oldx != x) {
        int temp = ufs->p[oldx];
        ufs->p[oldx] = x;
        oldx = temp;
    }
    return x;
}

/**
 * @brief Union 操作，将 y 并入到 x 所在的集合.
 * @param[in] s 并查集
 * @param[in] x 一个元素
 * @param[in] y 另一个元素
 * @return 如果二者已经在同一集合，并失败，返回-1，否则返回 0
 */
int ufs_union(ufs_t *ufs, int x, int y) {
    const int rx = ufs_find(ufs, x);
    const int ry = ufs_find(ufs, y);
    if(rx == ry) return -1;

    ufs->p[rx] += ufs->p[ry];
    ufs->p[ry] = rx;
    return 0;
}

/**
```

```
 * @brief 获取元素所在的集合的大小
 * @param[in] ufs 并查集
 * @param[in] x 元素
 * @return 元素所在的集合的大小
 */
int ufs_set_size(ufs_t *ufs, int x) {
    const int rx = ufs_find(ufs, x);
    return -ufs->p[rx];
}
```

ufs.c

### 5.7.2 病毒感染者

#### 描述

一个学校有 $n$ 个社团，一个学生能同时加入不同的社团。由于社团内的同学们交往频繁，如果一个学生感染了病毒，该社团的所有学生都会感染病毒。现在 0 号学生感染了病毒，问一共有多少个人感染了病毒。

#### 输入

输入包含多组测试用例。每个测试用例，第一行包含两个整数 $n$,$m$，$n$ 表示学生个数，$m$ 表示社团个数。假设 $0 < n \leq 30000, 0 \leq m \leq 500$。每个学生从 0 到 $n-1$ 编号。接下来是 $m$ 行，每行开头是一个整数 k，表示该社团的学生个数，接着是 $k$ 个整数表示该社团的学生编号。最后一个测试用例，$n = 0, m = 0$，表示输入结束。

#### 输出

对每个测试用例，输出感染了病毒的学生数目。

#### 样例输入

```
100 4
2 1 2
5 10 13 11 12 14
2 0 1
2 99 2
200 2
1 5
5 1 2 3 4 5
1 0
0 0
```

### 样例输出

```
4
1
1
```

### 分析

非常简单的并查集题目。

### 代码

suspects.c

```
/* POJ 1611 The Suspects, http://poj.org/problem?id=1611 */
#include <stdio.h>

#define MAXN 30000

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "ufs.c"  /* 见“树-> 并查集”这节 */

int main() {
    int n, m, k;
    while (scanf("%d%d", &n, &m) && n > 0) {
        ufs_t *ufs = ufs_create(MAXN);
        while (m--) {
            int x, y; /* 两个学生 */
            int rx, ry; /* x, y 所属的集合的根 */
            scanf("%d", &k);

            k--;
            scanf("%d", &x);
            rx = ufs_find(ufs, x);
            while (k--) {
                scanf("%d", &y);
                ry = ufs_find(ufs, y);
                ufs_union(ufs, rx, ry);  /* 只要是跟 x 同一个集合的都并进去 */
            }
        }
        /* 最后搜索 0 属于哪个集合，这个集合有多少人 */
        printf("%d\n", ufs_set_size(ufs, 0));
        ufs_destroy(ufs);
    }
    return 0;
}
```

suspects.c

### 相关的题目

与本题相同的题目：

- POJ 1611 The Suspects, http://poj.org/problem?id=1611

与本题相似的题目：

- None

## 5.7.3 两个黑帮

### 描述

Tadu 城市有两个黑帮帮派，已知有 $N$ 黑帮分子，从 1 到 $N$ 编号，每个人至少属于一个帮派。每个帮派至少有一个人。给你 $M$ 条信息，有两类信息：

- D a b，明确告诉你，a 和 b 属于不同的帮派
- A a b，问你，a 和 b 是否属于不同的帮派

### 输入

第一行是一个整数 $T$，表示有 $T$ 组测试用例。每组测试用例的第一行是两个整数 $N$ 和 $M$，接下来是 $M$ 行，每行包含一条消息。

### 输出

对每条消息"A a b"，基于当前获得的信息，输出判断。答案是"In the same gang.", "In different gangs." 和"Not sure yet." 中的一个。

### 样例输入

```
1
5 5
A 1 2
D 1 2
A 1 2
D 2 4
A 1 4
```

### 样例输出

```
Not sure yet.
In different gangs.
In the same gang.
```

### 分析

把不在一个集合的节点直接用并查集合并在一起。这样的话，如果询问的 2 个节点在同一个并查集里面，那么它们之间的关系是确定的，否则无法确定它们的关系。

现在还有一个问题是，在同一个集合里面的 2 个节点是敌对关系还是朋友关系？可以给每个节点另外附加个信息，记录其距离集合根节点的距离。如果，询问的 2 个节点距离其根节点的距离都是奇数或者都是偶数，那么这 2 个节点是朋友关系，否则是敌对关系。

### 代码

two_gangs.c

```
/* POJ 1703 Find them, Catch them, http://poj.org/problem?id=1703 */
#include <stdio.h>
#include <stdlib.h>

#define MAXN 1000001

/** 并查集. */
typedef struct ufs_t {
    int *p;     /** 树的双亲表示法 */
    int *dist;  /** 到根节点的距离的奇偶性 */
    int size;   /** 大小. */
} ufs_t;

/**
 * @brief 创建并查集.
 * @param[in] ufs 并查集
 * @param[in] ufs 并查集
 * @param[in] n 数组的容量
 * @return 并查集
 */
ufs_t* ufs_create(int n) {
    int i;
    ufs_t *ufs = (ufs_t*)malloc(sizeof(ufs_t));
    ufs->p = (int*)malloc(n * sizeof(int));
    ufs->dist = (int*)malloc(n * sizeof(int));
    for(i = 0; i < n; i++) {
        ufs->p[i] = -1;
        ufs->dist[i] = 0;
    }
    return ufs;
}

/**
 * @brief 销毁并查集.
 * @param[in] ufs 并查集
 * @return 无
```

```
 */
void ufs_destroy(ufs_t *ufs) {
    free(ufs->p);
    free(ufs->dist);
    free(ufs);
}

/**
 * @brief Find 操作, 带路径压缩, 递归版.
 * @param[in] s 并查集
 * @param[in] x 要查找的元素
 * @return 包含元素 x 的树的根
 */
int ufs_find(ufs_t *ufs, int x) {
    if (ufs->p[x] < 0) return x; // 终止条件

    const int parent = ufs->p[x];
    ufs->p[x] = ufs_find(ufs, ufs->p[x]); /* 回溯时的压缩路径 */
    ufs->dist[x] = (ufs->dist[x] + ufs->dist[parent]) % 2;
    return ufs->p[x];
}

/**
 * @brief Union 操作, 将 root2 并入到 root1.
 * @param[in] s 并查集
 * @param[in] root1 一棵树的根
 * @param[in] root2 另一棵树的根
 * @return 如果二者已经在同一集合, 并失败, 返回-1, 否则返回 0
 */
int ufs_union(ufs_t *ufs, int root1, int root2) {
    if(root1 == root2) return -1;
    ufs->p[root1] += ufs->p[root2];
    ufs->p[root2] = root1;
    return 0;
}

/**
 * @brief 添加一对敌人.
 * @param[inout] s 并查集
 * @param[in] x 一对敌人的一个
 * @param[in] y 一对敌人的另一个
 * @return 无
 */
void ufs_add_opponent(ufs_t *ufs, int x, int y) {
    const int rx = ufs_find(ufs, x);
    const int ry = ufs_find(ufs, y);
    ufs_union(ufs, rx, ry);
    /* ry 与 y 关系 + y 与 x 的关系 + x 与 rx 的关系 = ry 与 rx 的关系 */
```

```
    ufs->dist[ry] = (ufs->dist[y] + 1 + ufs->dist[x]) % 2;
}

int main() {
    int T;

    scanf("%d", &T);
    while (T--) {
        ufs_t *ufs = ufs_create(MAXN);
        int n, m;
        char c;
        int x, y, rx, ry;
        scanf("%d%d%*c", &n, &m);

        while (m--) {
            scanf("%c%d%d%*c", &c, &x, &y); //注意输入
            rx = ufs_find(ufs, x);
            ry = ufs_find(ufs, y);

            if (c == 'A') {
                if (rx == ry) { //如果根节点相同，则表示能判断关系
                    if (ufs->dist[x] != ufs->dist[y])
                        printf("In different gangs.\n");
                    else
                        printf("In the same gang.\n");
                } else
                    printf("Not sure yet.\n");
            } else if (c == 'D') {
                ufs_add_opponent(ufs, x, y);
            }
        }
        ufs_destroy(ufs);
    }
    return 0;
}
```

two_gangs.c

## 相关的题目

与本题相同的题目：

- POJ 1703 Find them, Catch them, http://poj.org/problem?id=1703

与本题相似的题目：

- None

### 5.7.4 食物链

#### 描述

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。现有 $N$ 个动物，从 1 到 $N$ 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是"1 X Y"，表示 X 和 Y 是同类。
- 第二种说法是"2 X Y"，表示 X 吃 Y。

此人对 $N$ 个动物，用上述两种说法，一句接一句地说出 $K$ 句话，这 $K$ 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 $N(1 \leq N \leq 50,000)$ 和 $K$ 句话 $(0 \leq K \leq 100,000)$，输出假话的总数。

#### 输入

第一行是两个整数 $N$ 和 $K$，以一个空格分隔。

以下 $K$ 行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中 D 表示说法的种类。

- 若 D=1，则表示 X 和 Y 是同类。
- 若 D=2，则表示 X 吃 Y。

#### 输出

只有一个整数，表示假话的数目。

#### 样例输入

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

#### 样例输出

```
3
```

### 分析

### 代码

food_chain.c

```
/* POJ 1182 食物链, Catch them, http://poj.org/problem?id=1182 */
#include <stdio.h>
#include <stdlib.h>

/** 并查集. */
typedef struct ufs_t {
    int *p;     /** 树的双亲表示法 */
    int *dist;  /** 表示 x 与父节点 p[x] 的关系, 0 表示 x 与 p[x] 是同类,
                    1 表示 x 吃 p[x], 2 表示 p[x] 吃 x */
    int size;   /** 大小. */
} ufs_t;

/**
 * @brief 创建并查集.
 * @param[in] ufs 并查集
 * @param[in] n 数组的容量
 * @return 并查集
 */
ufs_t* ufs_create(int n) {
    int i;
    ufs_t *ufs = (ufs_t*)malloc(sizeof(ufs_t));
    ufs->p = (int*)malloc(n * sizeof(int));
    ufs->dist = (int*)malloc(n * sizeof(int));
    for(i = 0; i < n; i++) {
        ufs->p[i] = -1;
        ufs->dist[i] = 0; // 自己与自己是同类
    }
    return ufs;
}

/**
 * @brief 销毁并查集.
 * @param[in] ufs 并查集
 * @return 无
 */
void ufs_destroy(ufs_t *ufs) {
    free(ufs->p);
    free(ufs->dist);
    free(ufs);
}

/**
 * @brief Find 操作, 带路径压缩, 递归版.
 * @param[in] s 并查集
```

```
 * @param[in] x 要查找的元素
 * @return 包含元素 x 的树的根
 */
int ufs_find(ufs_t *ufs, int x) {
    if (ufs->p[x] < 0) return x; // 终止条件

    const int parent = ufs->p[x];
    ufs->p[x] = ufs_find(ufs, ufs->p[x]); /* 回溯时的压缩路径 */
    /* 更新关系 */
    ufs->dist[x] = (ufs->dist[x] + ufs->dist[parent]) % 3;
    return ufs->p[x];
}

/**
 * @brief Union 操作，将 root2 并入到 root1.
 * @param[in] s 并查集
 * @param[in] root1 一棵树的根
 * @param[in] root2 另一棵树的根
 * @return 如果二者已经在同一集合，并失败，返回-1，否则返回 0
 */
int ufs_union(ufs_t *ufs, int root1, int root2) {
    if(root1 == root2) return -1;
    ufs->p[root1] += ufs->p[root2];
    ufs->p[root2] = root1;
    return 0;
}

/**
 * @brief 添加一对关系.
 * @param[inout] s 并查集
 * @param[in] x 一个
 * @param[in] y 另一个
 * @param[in] len
 * @return 无
 */
void ufs_add_relation(ufs_t *ufs, int x, int y, int relation) {
    const int rx = ufs_find(ufs, x);
    const int ry = ufs_find(ufs, y);
    ufs_union(ufs, ry, rx); /* 注意顺序! */
    /* rx 与 x 关系 + x 与 y 的关系 + y 与 ry 的关系 = rx 与 ry 的关系 */
    ufs->dist[rx] = (ufs->dist[y] - ufs->dist[x] + 3 + relation) % 3;
}

int main() {
    int n, k;
    int result = 0; /* 假话的数目 */
    ufs_t *ufs;
```

```
    scanf("%d%d", &n, &k);
    ufs = ufs_create(n + 1);

    while(k--) {
        int d, x, y;
        scanf("%d%d%d", &d, &x, &y);

        if (x > n || y > n || (d == 2 && x == y)) {
            result++;
        } else {
            const int rx = ufs_find(ufs, x);
            const int ry = ufs_find(ufs, y);

            if (rx == ry) { /* 若在同一个集合则可确定 x 和 y 的关系 */
                if((ufs->dist[x] - ufs->dist[y] + 3) % 3 != d - 1)
                    result++;
            } else {
                ufs_add_relation(ufs, x, y, d-1);
            }
        }
    }

    printf("%d\n", result);

    ufs_destroy(ufs);
    return 0;
}
```

food_chain.c

### 相关的题目

与本题相同的题目：

- POJ 1182 食物链, http://poj.org/problem?id=1182
- wikioi 1074 食物链, http://www.wikioi.com/problem/1074/

与本题相似的题目：

- None

## 5.8 线段树

### 5.8.1 原理和实现

**线段树**，也叫区间树 (interval tree)，它在各个节点保存一条线段（即子数组）。设数列 $A$ 包含 $N$ 个元素，则线段树的根节点表示整个区间 $A[1,N]$，左孩子表示区间 $A[1,(1+N)/2]$，右孩子表示区间 $A[(1+N)/2+1,N]$，不断递归，直到叶子节点，叶子节点只包含一个元素。

线段树有如下特征：

- 线段树是一棵完全二叉树
- 线段树的深度不超过 $\log L$, $L$ 是区间的长度
- 线段树把一个长度为 L 的区间分成不超过 $2\log L$ 条线段

线段树的基本操作有构造线段树、区间查询和区间修改。

线段树通常用于解决和区间统计有关的问题。比如某些数据可以按区间进行划分，按区间动态进行修改，而且还需要按区间多次进行查询，那么使用线段树可以达到较快的查询速度。

用线段树解题，关键是要想清楚每个节点要存哪些信息（当然区间起点和终点，以及左右孩子指针是必须的），以及这些信息如何高效查询，更新。不要一更新就更新到叶子节点，那样更新操作的效率最坏有可能 $O(N)$ 的。

### 5.8.2 Balanced Lineup

#### 描述

给定 $N(1 \leq N \leq 50,000)$ 个数, $A_1, A_2, ..., A_N$，求任意区间中最大数和最小数的差。

#### 输入

第一行包含两个整数，$N$ 和 $Q$。$Q$ 表示查询次数。

第 2 到 N+1 行，每行包含一个整数 $A_i$。

第 N+2 到 N+Q+1 行，每行包含两个整数 $a$ 和 $b(1 \leq a \leq b \leq N)$，表示区间 $A[a, b]$。

#### 输出

对每个查询进行回应，输出该区间内最大数和最小数的差

#### 样例输入

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

#### 样例输出

```
6
3
0
```

## 分析

本题是“区间求和”，只需要“线段树构造”和“区间查询”两个操作。

## 代码

balanced_lineup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define MAXN 50001
#define INF INT_MAX
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))
#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)

typedef struct node_t {
    int left, right;  /* 区间  */
    int max, min;  /* 本区间里的最大值和最小值 */
} node_t;

int A[MAXN]; /* 输入数据，0 位置未用 */

/* 完全二叉树，结点编号从 1 开始，层次从 1 开始.
 * 用一维数组存储完全二叉树，空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

int minx, maxx; /* 存放查询的结果 */

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点，为区间 A[l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l, node[t].right = r;
    if (l == r) {
        node[t].max = node[t].min = A[l];
        return;
    }
    const int mid = (l + r) / 2;
    build(L(t), l, mid);
    build(R(t), mid + 1, r);
    node[t].max = max(node[L(t)].max,node[R(t)].max);
```

```
    node[t].min = min(node[L(t)].min,node[R(t)].min);
}

/* 查询根结点为 t，区间为 A[l,r] 的最大值和最小值 */
void query(int t, int l, int r) {
    if (node[t].left == l && node[t].right == r) {
        if (maxx < node[t].max)
            maxx = node[t].max;
        if (minx > node[t].min)
            minx = node[t].min;
        return;
    }
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid) {
        query(R(t), l, r);
    } else if (r <= mid) {
        query(L(t), l, r);
    } else {
        query(L(t), l, mid);
        query(R(t), mid + 1, r);
    }
}

int main() {
    int n, q, i;

    scanf("%d%d", &n, &q);
    for (i = 1; i <= n; i++) scanf("%d", &A[i]);

    init();
    /* 建立以 tree[1] 为根结点，区间为 A[1,n] 的线段树 */
    build(1, 1, n);

    while (q--) {
        int a, b;
        scanf("%d%d", &a, &b);
        maxx = 0;
        minx = INF;
        query(1, a, b); /* 查询区间 A[a,b] 的最大值和最小值 */
        printf("%d\n", maxx - minx);
    }
    return 0;
}
```

balanced_lineup.c

## 相关的题目

与本题相同的题目：

- POJ 3264 Balanced Lineup, http://poj.org/problem?id=3264

与本题相似的题目：

- None

### 5.8.3 线段树练习 1

#### 描述

一行 $N(1 \leq N < 100000)$ 个方格，开始每个格子里都有一个整数。现在动态地提出一些命令请求，有两种命令，查询和增加：求某一个特定的子区间 $[a, b]$ 中所有元素的和；指定某一个格子 $x$，加上一个特定的值 A。现在要求你能对每个请求作出正确的回答。

#### 输入

输入文件第一行为一个整数 $N$，接下来是 $n$ 行每行 1 个整数，表示格子中原来的整数。接下来是一个正整数 $Q$，再接下来有 $Q$ 行，表示 $Q$ 个询问，第一个整数表示命令代号，命令代号 1 表示增加，后面的两个数 $a$ 和 $x$ 表示给位置 $a$ 上的数值增加 $x$，命令代号 2 表示区间求和，后面两个整数 a 和 b，表示要求 [a,b] 之间的区间和。

#### 输出

共 $Q$ 行，每个整数

#### 样例输入

```
6
4
5
6
2
1
3
4
1 3 5
2 1 4
1 1 9
2 2 6
```

#### 样例输出

```
22
22
```

## 分析

单点更新 + 区间求和

## 代码

interval_tree1.c

```
/* wikioi 1080 线段树练习 , http://www.wikioi.com/problem/1080/ */
#include <stdio.h>
#include <string.h>

#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)
#define MAXN 100001

typedef long long int64_t;

typedef struct node_t {
    int left, right;
    int64_t sum;
} node_t;

int A[MAXN]; /* 输入数据, 0 位置未用 */

/* 完全二叉树, 结点编号从 1 开始, 层次从 1 开始.
 * 用一维数组存储完全二叉树, 空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点, 为区间 A[l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l;
    node[t].right = r;
    if (l == r) {
        node[t].sum = A[l];
        return;
    }
    const int mid = (l + r) / 2;
    build(L(t), l, mid);
    build(R(t), mid + 1, r);
    node[t].sum = node[L(t)].sum + node[R(t)].sum;
}

/* 给区间 A[l,r] 里的 pos 位置加 delta */
```

```
void update(int t, int l, int r, int pos, int64_t delta) {
    if (node[t].left > pos || node[t].right < pos) return;
    if (node[t].left == node[t].right) {
        node[t].sum += delta;
        return;
    }

    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid) update(R(t), l, r, pos, delta);
    else if (r <= mid) update(L(t), l, r, pos, delta);
    else {
        update(L(t), l, mid, pos, delta);
        update(R(t), mid + 1, r, pos, delta);
    }
    node[t].sum = node[L(t)].sum + node[R(t)].sum;
}

/* 查询根结点为 t, 区间为 A[l,r] 的和 */
int64_t query(int t, int l, int r) {
    if (node[t].left == l && node[t].right == r)
        return node[t].sum;
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid) return query(R(t), l, r);
    else if (r <= mid) return query(L(t), l, r);
    else return query(L(t), l, mid) + query(R(t), mid + 1, r);
}

int main() {
    int i, n, q;
    scanf("%d", &n);
    for (i = 1; i <= n; i++) scanf("%d", &A[i]);

    init();
    /* 建立以 tree[1] 为根结点, 区间为 A[1,n] 的线段树 */
    build(1, 1, n);

    scanf("%d", &q);
    while (q--) {
        int cmd;
        scanf("%d", &cmd);
        if (cmd == 2) {
            int a, b;
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(1, a, b)); /* 查询区间 A[a,b] 的和 */
        } else {
            int a;
            int64_t x;
            scanf("%d%lld", &a, &x);
```

```
            if (x != 0) update(1, 1, n, a, x);
        }
    }
    return 0;
}
```

interval_tree1.c

### 相关的题目

与本题相同的题目：

- wikioi 1080 线段树练习 1, http://www.wikioi.com/problem/1080/

与本题相似的题目：

- wikioi 1081 线段树练习 2, http://www.wikioi.com/problem/1081/ 。本题是“区间更新 + 单点查询”，可以转化为线段树练习 1。设原数组为 $A[N]$，将其转化为差分数列，然后在数组上维护一棵线段树。“区间更新”操作转化为两个“单点更新”操作：将 $A[a]$ 加上 $x$，并将 $A[b+1]$ 减去 $x$（也就是加上 $-x$）。“单点查询”操作转化为“区间求和”操作：求 $A$ 数组 $[1..i]$ 范围内所有数的和。这样就转化成与线段树练习 1 完全相同了。标程 https://gist.github.com/soulmachine/6449609

## 5.8.4 A Simple Problem with Integers

### 描述

You have $N$ integers, $A_1, A_2, ..., A_N$. You need to deal with two kinds of operations. One type of operation is to add some given number to each number in a given interval. The other is to ask for the sum of numbers in a given interval.

### 输入

The first line contains two numbers $N$ and $Q$. $1 \leq N, Q \leq 100000$.

The second line contains $N$ numbers, the initial values of $A_1, A_2, ..., A_N$. $-1000000000 \leq A_i \leq 1000000000$.

Each of the next $Q$ lines represents an operation. "C a b c" means adding $c$ to each of $A_a, A_{a+1}, ..., A_b$. $-10000 \leq c \leq 10000$. "Q a b" means querying the sum of $A_a, A_{a+1}, ..., A_b$.

### 输出

You need to answer all $Q$ commands in order. One answer in a line.

### 样例输入

```
10 5
1 2 3 4 5 6 7 8 9 10
Q 4 4
Q 1 10
Q 2 4
C 3 6 3
Q 2 4
```

### 样例输出

```
4
55
9
15
```

### 提示

The sums may exceed the range of 32-bit integers.

### 分析

区间更新 + 区间求和。

树节点要存哪些信息？只存该区间的和，行不行？只存和，会导致每次加数的时候都要更新到叶子节点，速度太慢。本题节点的结构如下：

```
typedef struct node_t {
    int left, right;
    int64_t sum;  /* 本区间的和实际上是 sum+inc*[right-left+1] */
    int64_t inc;  /* 增量 c 的累加 */
} node_t;
```

### 代码

poj3468.c

```
#include <stdio.h>
#include <string.h>

#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)
#define MAXN 100001

typedef long long int64_t;

typedef struct node_t {
    int left, right;
    int64_t sum;  /* 本区间的和实际上是 sum+inc*[right-left+1] */
```

```
    int64_t inc;  /* 增量 c 的累加 */
} node_t;

int A[MAXN]; /* 输入数据, 0 位置未用 */

/* 完全二叉树, 结点编号从 1 开始, 层次从 1 开始.
 * 用一维数组存储完全二叉树, 空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点, 为区间 A[l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l;
    node[t].right = r;
    if (l == r) {
        node[t].sum = A[l];
        return;
    }
    const int mid = (l + r) / 2;
    build(L(t), l, mid);
    build(R(t), mid+1, r);
    node[t].sum = node[L(t)].sum + node[R(t)].sum;
}

/* 给区间 A[l,r] 里的每个元素都加 c */
void update(int t, int l, int r, int64_t c) {
    if (node[t].left == l && node[t].right == r) {
        node[t].inc += c;
        node[t].sum += c * (r - l + 1);
        return;
    }
    if (node[t].inc) {
        node[R(t)].inc += node[t].inc;
        node[L(t)].inc += node[t].inc;
        node[R(t)].sum += node[t].inc * (node[R(t)].right - node[R(t)].left + 1);
        node[L(t)].sum += node[t].inc * (node[L(t)].right - node[L(t)].left + 1);
        node[t].inc = 0;
    }
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid)
        update(R(t), l, r, c);
    else if (r <= mid)
        update(L(t), l, r, c);
```

```
    else {
        update(L(t), l, mid, c);
        update(R(t), mid + 1, r, c);
    }
    node[t].sum = node[L(t)].sum + node[R(t)].sum;
}

/* 查询根结点为 t, 区间为 A[l,r] 的和 */
int64_t query(int t, int l, int r) {
    if (node[t].left == l && node[t].right == r)
        return node[t].sum;
    if (node[t].inc) {
        node[R(t)].inc += node[t].inc;
        node[L(t)].inc += node[t].inc;
        node[R(t)].sum += node[t].inc * (node[R(t)].right - node[R(t)].left + 1);
        node[L(t)].sum += node[t].inc * (node[L(t)].right - node[L(t)].left + 1);
        node[t].inc = 0;
    }
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid)
        return query(R(t), l, r);
    else if (r <= mid)
        return query(L(t), l, r);
    else
        return query(L(t), l, mid) + query(R(t), mid + 1, r);
}

int main() {
    int i, n, q;
    char s[5];
    scanf("%d%d", &n, &q);
    for (i = 1; i <= n; i++) scanf("%d", &A[i]);

    init();
    /* 建立以 tree[1] 为根结点, 区间为 A[1,n] 的线段树 */
    build(1, 1, n);

    while (q--) {
        int a, b;
        int64_t c;
        scanf("%s", s);
        if (s[0] == 'Q') {
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(1, a, b)); /* 查询区间 A[a,b] 的和 */
        } else {
            scanf("%d%d%lld", &a, &b, &c);
            if (c != 0) update(1, a, b, c);
        }
```

```
    }
    return 0;
}
```

poj3468.c

### 相关的题目

与本题相同的题目：

- POJ 3468 A Simple Problem with Integers, http://poj.org/problem?id=3468

与本题相似的题目：

- None

## 5.8.5 约瑟夫问题

### 描述

有编号从 1 到 $N$ 的 $N$ 个小朋友在玩一种出圈的游戏。开始时 $N$ 个小朋友围成一圈，编号为 $i+1$ 的小朋友站在编号为 $i$ 小朋友左边。编号为 1 的小朋友站在编号为 $N$ 的小朋友左边。首先编号为 1 的小朋友开始报数，接着站在左边的小朋友顺序报数，直到数到某个数字 $M$ 时就出圈。直到只剩下 1 个小朋友，则游戏完毕。

现在给定 $N, M$，求 $N$ 个小朋友的出圈顺序。

### 输入

唯一的一行包含两个整数 $N, M (1 \leq N, M \leq 30000)$。

### 输出

唯一的一行包含 $N$ 个整数，每两个整数中间用空格隔开，第 $i$ 个整数表示第 $i$ 个出圈的小朋友的编号。

### 样例输入

```
5 3
```

### 样例输出

```
3 1 5 2 4
```

## 分析

约瑟夫问题的难点在于，每一轮都不能通过简单的运算得出下一轮谁淘汰，因为中间有人已经退出了。因此一般只能模拟，效率很低。

现在考虑，每一轮都令所有剩下的人从左到右重新编号，例如 3 退出后，场上还剩下 1、2、4、5，则给 1 新编号 1，2 新编号 2，4 新编号 3，5 新编号 4。不妨称这个编号为“剩余队列编号”。如下所示，括号内为原始编号：

```
1(1) 2(2) 3(3) 4(4) 5(5) --> 剩余队列编号 3 淘汰，对应原编号 3
1(1) 2(2) 3(4) 4(5) --> 剩余队列编号 1 淘汰，对应原编号 1
1(2) 2(4) 3(5) --> 剩余队列编号 3 淘汰，对应原编号 5
1(2) 2(4) --> 剩余队列编号 1 淘汰，对应原编号 2
1(4) --> 剩余队列编号 1 滔天，对应原编号 4
```

一个人在当前剩余队列中编号为 $i$，则说明他是从左到右数第 $i$ 个人，这启发我们可以用线段树来解决问题。用线段树维护原编号 $[i..j]$ 内还有多少人没有被淘汰，这样每次选出被淘汰者后，在当前线段树中查找位置就可以了。

例如我们有 5 个原编号，当前淘汰者在剩余队列中编号为 3，先看左子树，即原编号 [1..3] 区间内，如果剩下的人不足 3 个，则说明当前剩余编号为 3 的这个人原编号只能是在 [4..5] 区间内，继续在 [4..5] 上搜索；如果 [1..3] 内剩下的人大于等于 3 个，则说明就在 [1..3] 内，也继续缩小范围查找，这样即可在 $O(\log N)$ 时间内完成对应。问题得到圆满的解决。

## 代码

josephus_problem.c

```
/* wikioi 1282 约瑟夫问题, http://www.wikioi.com/problem/1282/ */
#include <stdio.h>
#include <string.h>

#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)
#define MAXN 30001

typedef struct node_t {
    int left, right;
    int count; /* 区间内的元素个数 */
} node_t;

/* 完全二叉树，结点编号从 1 开始，层次从 1 开始.
 * 用一维数组存储完全二叉树，空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

void init() {
    memset(node, 0, sizeof(node));
}
```

```
/* 以 t 为根结点，为区间 [l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l;
    node[t].right = r;
    node[t].count = r - l + 1;
    if (l == r) return;

    const int mid = (r + l) / 2;
    build(L(t), l, mid);
    build(R(t), mid + 1, r);
}

/**
 * @brief 输出 i
 * @param[in] t 根节点
 * @param[in] i 剩余队列编号
 * @return 被删除的实际数字
 */
int delete(int t, int i) {
    node[t].count--;
    if (node[t].left == node[t].right) {
        printf("%d ", node[t].left);
        return node[t].left;
    }
    if (node[L(t)].count >= i) return delete(L(t), i);
    else return delete(R(t), i - node[L(t)].count); /* 左子树人数不足，则在右子树查找 */
}

/**
 * @brief 返回 1 到 i 内的活人数
 * @param[in] t 根节点
 * @param[in] i 原始队列的数字
 * @return 1 到 i 内的活人数
 */
int get_count(int t, int i) {
    if (node[t].right <= i) return node[t].count;

    const int mid = (node[t].left + node[t].right) / 2;
    int s = 0;
    if (i > mid) {
        s += node[L(t)].count;
        s += get_count(R(t), i);
    } else
        s += get_count(L(t), i);
    return s;
}
```

```
int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    init();
    build(1, 1, n);

    int i;
    int j = 0; /* 剩余队列的虚拟编号 */
    for (i = 1; i <= n; i++) {
        j += m;
        if (j > node[1].count)
            j %= node[1].count;
        if (j == 0) j = node[1].count;
        const int k = delete(1, j);
        j = get_count(1, k);
    }
    return 0;
}
```

josephus_problem.c

### 相关的题目

与本题相同的题目：

- wikioi 1282 约瑟夫问题, http://www.wikioi.com/problem/1282/

与本题相似的题目：

- None

## 5.9 Trie 树

### 5.9.1 原理和实现

trie_tree.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


#define MAXN 10000   /** 输入的编码的最大个数. */
#define CHAR_COUNT  10 /** 字符的种类, 也即单个节点的子树的最大个数 */
#define MAX_CODE_LEN 10 /** 编码的最大长度. */
#define MAX_NODE_COUNT  (MAXN * MAX_CODE_LEN + 1)  /** 字典树的最大节点个数. */
                    /* 如果没有指定 MAXN, 则是 CHAR_COUNT^(MAX_CODE_LEN+1)-1 */

/** 字典树的节点 */
```

```
typedef struct trie_node_t {
    struct trie_node_t* next[CHAR_COUNT];
    bool is_tail; /** 标记当前字符是否位于某个串的尾部 */
} trie_node_t;

/** 字典树. */
typedef struct trie_tree_t {
    trie_node_t *root; /** 树的根节点 */
    int size; /** 树中实际出现的节点数 */

    trie_node_t nodes[MAX_NODE_COUNT]; /* 开一个大数组, 加快速度 */
} trie_tree_t;

/** 创建. */
trie_tree_t* trie_tree_create(void) {
    trie_tree_t *tree = (trie_tree_t*)malloc(sizeof(trie_tree_t));
    tree->root = &(tree->nodes[0]);
    memset(tree->nodes, 0, sizeof(tree->nodes));
    tree->size = 1;
    return tree;
}

/** 销毁. */
void trie_tree_destroy(trie_tree_t *tree) {
    free(tree);
    tree = NULL;
}

/** 将当前字典树中的所有节点信息清空 */
void trie_tree_clear(trie_tree_t *tree) {
    memset(tree->nodes, 0, sizeof(tree->nodes));
    tree->size = 1; // 清空时一定要注意这一步!
}

/** 在当前树中插入 word 字符串, 若出现非法, 返回 false */
bool trie_tree_insert(trie_tree_t *tree, char *word) {
    int i;
    trie_node_t *p = tree->root;
    while (*word) {
        int curword = *word - '0';
        if (p->next[curword] == NULL) {
            p->next[curword] = &(tree->nodes[tree->size++]);
        }
        p = p->next[curword];
        if (p->is_tail) return false; // 某串是当前串的前缀

        word++; // 指针下移
    }
```

```
    p->is_tail = true; // 标记当前串已是结尾

    // 判断当前串是否是某个串的前缀
    for (i = 0; i < CHAR_COUNT; i++)
        if (p->next[i] != NULL)
            return false;
    return true;
}
```

trie_tree.c

### 5.9.2 Immediate Decodebility

#### 描述

An encoding of a set of symbols is said to be immediately decodable if no code for one symbol is the prefix of a code for another symbol. We will assume for this problem that all codes are in binary, that no two codes within a set of codes are the same, that each code has at least one bit and no more than ten bits, and that each set has at least two codes and no more than eight.

Examples: Assume an alphabet that has symbols `{A, B, C, D}`.

The following code is immediately decodable:

```
A:01 B:10 C:0010 D:0000
```

but this one is not:

```
A:01 B:10 C:010 D:0000 (Note that A is a prefix of C)
```

#### 输入

Write a program that accepts as input a series of groups of records from standard input. Each record in a group contains a collection of zeroes and ones representing a binary code for a different symbol. Each group is followed by a single separator record containing a single 9; the separator records are not part of the group. Each group is independent of other groups; the codes in one group are not related to codes in any other group (that is, each group is to be processed independently).

#### 输出

For each group, your program should determine whether the codes in that group are immediately decodable, and should print a single output line giving the group number and stating whether the group is, or is not, immediately decodable.

#### 样例输入

```
01
10
```

```
0010
0000
9
01
10
010
0000
9
```

### 样例输出

```
Set 1 is immediately decodable
Set 2 is not immediately decodable
```

### 分析

判断一个串是否是另一个串的前缀，这正是 Trie 树（即字典树）的用武之地。

### 代码

immediate_decodebility.c

```
/* POJ 1056 IMMEDIATE DECODABILITY, http://poj.org/problem?id=1056 */

#define CHAR_COUNT  2
#define MAX_CODE_LEN 10
/** 字典树的最大节点个数.
 * 本题中每个 code 不超过 10bit，即树的高度不超过 11，因此最大节点个数为 2^11-1
 */
#define MAX_NODE_COUNT  ((1<<(MAX_CODE_LEN+1))-1)

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "trie_tree.c"  /* 见“树->Trie 树”这节 */

int main() {
    int T = 0;  // 测试用例编号
    char line[MAX_NODE_COUNT]; // 输入的一行
    trie_tree_t *trie_tree = trie_tree_create();
    bool islegal = true;

    while (scanf("%s", line) != EOF) {
        if (strcmp(line, "9") == 0) {
            if (islegal)
                printf("Set %d is immediately decodable\n", ++T);
            else
                printf("Set %d is not immediately decodable\n", ++T);
            trie_tree_clear(trie_tree);
            islegal = true;
        } else {
```

```
            if (islegal)
                islegal = trie_tree_insert(trie_tree, line);
        }
    }
    trie_tree_destroy(trie_tree);
    return 0;
}
```

immediate_decodebility.c

### 相关的题目

与本题相同的题目：

- POJ 1056 IMMEDIATE DECODABILITY, http://poj.org/problem?id=1056

与本题相似的题目：

- POJ 3630 Phone List, http://poj.org/problem?id=3630
  参考代码 https://gist.github.com/soulmachine/6609332

## 5.9.3 Hardwood Species

### 描述

现在通过卫星扫描，扫描了很多区域的树，并获知了每棵树的种类，求每个种类的百分比。

### 输入

一行一棵树，表示该树的种类。每个名字不超过 30 字符，树的种类不超过 10,000，不超过 1,000,000 棵树。

### 输出

按字母顺序，打印每个种类的百分比，精确到小数点后 4 位。

### 样例输入

```
Red Alder
Ash
Aspen
Basswood
Ash
Beech
Yellow Birch
Ash
Cherry
Cottonwood
```

```
Ash
Cypress
Red Elm
Gum
Hackberry
White Oak
Hickory
Pecan
Hard Maple
White Oak
Soft Maple
Red Oak
Red Oak
White Oak
Poplan
Sassafras
Sycamore
Black Walnut
Will
```

## 样例输出

```
Ash 13.7931
Aspen 3.4483
Basswood 3.4483
Beech 3.4483
Black Walnut 3.4483
Cherry 3.4483
Cottonwood 3.4483
Cypress 3.4483
Gum 3.4483
Hackberry 3.4483
Hard Maple 3.4483
Hickory 3.4483
Pecan 3.4483
Poplan 3.4483
Red Alder 3.4483
Red Elm 3.4483
Red Oak 6.8966
Sassafras 3.4483
Soft Maple 3.4483
Sycamore 3.4483
White Oak 10.3448
Willow 3.4483
Yellow Birch 3.4483
```

## 分析

无

## 代码

hardwood_species.c

```
/* POJ 2418 Hardwood Species, http://poj.org/problem?id=2418 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


#define MAXN 1000   /**  no more than 10,000 species, 会 MLE, 因此减一个 0 */
#define CHAR_COUNT  128 /** ASCII 编码范围 */
#define MAX_WORD_LEN 30 /** 编码的最大长度. */
#define MAX_NODE_COUNT  (MAXN * MAX_WORD_LEN + 1)  /** 字典树的最大节点个数. */


/** 字典树的节点 */
typedef struct trie_node_t {
    struct trie_node_t* next[CHAR_COUNT];
    int count;  /** 该单词出现的次数 */
} trie_node_t;

/** 字典树. */
typedef struct trie_tree_t {
    trie_node_t *root; /** 树的根节点 */
    int size; /** 树中实际出现的节点数 */

    trie_node_t nodes[MAX_NODE_COUNT]; /* 开一个大数组, 加快速度 */
} trie_tree_t;

/** 创建. */
trie_tree_t* trie_tree_create(void) {
    trie_tree_t *tree = (trie_tree_t*)malloc(sizeof(trie_tree_t));
    tree->root = &(tree->nodes[0]);
    memset(tree->nodes, 0, sizeof(tree->nodes));
    tree->size = 1;
    return tree;
}

/** 销毁. */
void trie_tree_destroy(trie_tree_t *tree) {
    free(tree);
    tree = NULL;
}

/** 将当前字典树中的所有节点信息清空 */
```

```
void trie_tree_clear(trie_tree_t *tree) {
    memset(tree->nodes, 0, sizeof(tree->nodes));
    tree->size = 1; // 清空时一定要注意这一步!
}

/** 在当前树中插入 word 字符串 */
void trie_tree_insert(trie_tree_t *tree, char *word) {
    trie_node_t *p = tree->root;
    while (*word) {
        if (p->next[*word] == NULL) {
            p->next[*word] = &(tree->nodes[tree->size++]);
        }
        p = p->next[*word];

        word++; // 指针下移
    }
    p->count++;
    return;
}


int n = 0;  // 输入的行数

/** 深度优先遍历. */
void dfs_travel(trie_node_t *root) {
    static char word[MAX_WORD_LEN + 1]; /* 中间结果 */
    static int pos;  /* 当前位置 */
    int i;

    if (root->count) { /* 如果 count 不为 0, 则肯定找到了一个单词 */
        word[pos] = '\0';
        printf("%s %0.4f\n", word, ((float)root->count * 100) / n);
    }
    for (i = 0; i < CHAR_COUNT; i++) {  /* 扩展 */
        if (root->next[i]) {
            word[pos++] = i;
            dfs_travel(root->next[i]);
            pos--; /* 返回上一层时恢复位置 */
        }
    }
}

int main() {
    char line[MAX_WORD_LEN + 1];
    trie_tree_t *trie_tree = trie_tree_create();

    while (gets(line)) {
        trie_tree_insert(trie_tree, line);
```

```
        n++;
    }
    dfs_travel(trie_tree->root);

    trie_tree_destroy(trie_tree);
    return 0;
}
```

hardwood_species.c

### 相关的题目

与本题相同的题目：

- POJ 2418 Hardwood Species, http://poj.org/problem?id=2418

与本题相似的题目：

- 无

## 5.10 Expression Tree

```
#include <iostream>
#include <conio.h>
using namespace std;
struct EXTree{
    char data;
    EXTree *l, *r;
}*root = NULL, *p = NULL, *t = NULL, *y = NULL;
struct EXTreeNode{
    EXTree *pt;
    EXTreeNode *next;
}*top = NULL, *q = NULL, *np = NULL;
void push(EXTree *ptr)
{
    np = new node;
    np->pt = ptr;
    np->next = NULL;
    if (top == NULL)
    {
        top = np;
    }
    else
    {
        q = top;
        top = np;
        np->next = q;
    }
}
```

```
EXTree *pop()
{
    if (top == NULL)
    {
        cout<<"underflow\n";
    }
    else
    {
        q = top;
        top = top->next;
        return(q->pt);
        delete(q);
    }
}
void oprnd_str(char val)
{
    if (val >= 48 && val <= 57)
    {
        t = new tree;
        t->data = val;
        t->l = NULL;
        t->r = NULL;
        push(t);
    }
    else if (val >= 42 && val <= 47)
    {
        p = new tree;
        p->data = val;
        p->l = pop();
        p->r = pop();
        push(p);
    }
}
char pstorder(EXTree *w)
{
    if (w != NULL)
    {
        pstorder(w->l);
        pstorder(w->r);
        cout<<w->data;
    }
}
int main()
{
    char a[15];
    int i;
    int j = -1;
    cout<<"enter the value of character string\n";
```

```
    cin>>a;
    i = strlen(a);
    while (i >= 0)
    {
        i--;
        oprnd_str(a[i]);
    }
    cout<<"displaying in postorder\n";
    pstorder(pop());
    getch();
}


Output:
enter the value of character string
-+-5/763*48
displaying in postorder
576/-3+48*-
```

# 第 6 章
# 查找

## 6.1 折半查找

binary_search.c

```
/** 数组元素的类型 */
typedef int elem_t;
/**
  * @brief 有序顺序表的折半查找算法.
  *
  * @param[in] a 存放数据元素的数组，已排好序
  * @param[in] n 数组的元素个数
  * @param[in] x 要查找的元素
  * @return 如果找到 x，则返回其下标。 如果找
  * 不到 x 且 x 小于 array 中的一个或多个元素，则为一个负数，该负数是大
  * 于 x 的第一个元素的索引的按位求补。 如果找不到 x 且 x 大于 array 中的
  * 任何元素，则为一个负数，该负数是（最后一个元素的索引加 1）的按位求补。
  */
int binary_search(const elem_t a[], const int n, const elem_t x) {
    int left = 0, right = n -1, mid;
    while(left <= right) {
        mid = left + (right - left) / 2;
        if(x > a[mid]) {
            left = mid + 1;
        } else if(x < a[mid]) {
            right = mid - 1;
        } else {
            return mid;
        }
    }
    return -(left+1);
}
```

binary_search.c

# 6.2 哈希表

## 6.2.1 原理和实现

哈希表处理冲突有两种方式，开地址法 (Open Addressing) 和闭地址法 (Closed Addressing)。

闭地址法也即拉链法 (Chaining)，每个哈希地址里不再是一个元素，而是链表的首地址。

开地址法有很多方案，有线性探测法 (Linear Probing)、二次探测法 (Quodratic Probing) 和双散列法 (Double Hashing) 等。

下面是拉链法的 C 语言实现。

### 代码

hash_set.hpp

```
/** 元素的哈希函数  */
template<typename elem_t>
int elem_hash(const elem_t &e);

/** 元素的比较函数  */
template<typename elem_t>
bool operator==(const elem_t &e1, const elem_t &e2);

/** 哈希集合，elem_t 是元素的数据类型. */
template<typename elem_t>
class hash_set {
public:
    hash_set(int prime, int capacity);
    ~hash_set();
    bool find(const elem_t &elem); /** 查找某个元素是否存在. */
    bool insert(const elem_t &elem); /** 添加一个元素，如果已存在则添加失败. */
private:
    int prime; /** 哈希表取模的质数，也即哈希桶的个数，小于 capacity. */
    int capacity; /** 哈希表容量，一定要大于元素最大个数  */

    int *head/*[PRIME]*/; /** 首节点下标 */

    struct node_t {
        elem_t elem;
        int next;
        node_t():next(-1) {}
    } *node/*[HASH_SET_CAPACITY]*/; /** 静态链表 */

    int size; /** 实际元素个数 */
};

template<typename elem_t>
hash_set<elem_t>::hash_set(int prime, int capacity) {
    this->prime = prime;
```

```
    this->capacity = capacity;
    head = new int[prime];
    node = new node_t[capacity];
    fill(head, head + prime, -1);
    fill(node, node + capacity, node_t());
    size = 0;
}

template<typename elem_t>
hash_set<elem_t>::~hash_set() {
    this->prime = 0;
    this->capacity = 0;
    delete[] head;
    delete[] node;
    head = NULL;
    node = NULL;
    size = 0;
}

template<typename elem_t>
bool hash_set<elem_t>::find(const elem_t &elem) {
    for (int i = head[elem_hash(elem)]; i != -1; i = node[i].next)
        if (elem == node[i].elem) return true;

    return false;
}

template<typename elem_t>
bool hash_set<elem_t>::insert(const elem_t &elem) {
    const int hash_code = elem_hash(elem);

    for (int i = head[hash_code]; i != -1; i = node[i].next)
        if (elem == node[i].elem) return false; // 已经存在

    /* 不存在，则插入在首节点之前 */
    node[size].next = head[hash_code];
    node[size].elem = elem;
    head[hash_code] = size++;
    return true;
}
```

hash_set.hpp

### 6.2.2 Babelfish

#### 描述

You have just moved from Waterloo to a big city. The people here speak an incomprehensible dialect of a foreign language. Fortunately, you have a dictionary to help you understand them.

### 输入

Input consists of up to 100,000 dictionary entries, followed by a blank line, followed by a message of up to 100,000 words. Each dictionary entry is a line containing an English word, followed by a space and a foreign language word. No foreign word appears more than once in the dictionary. The message is a sequence of words in the foreign language, one word on each line. Each word in the input is a sequence of at most 10 lowercase letters.

### 输出

Output is the message translated to English, one word per line. Foreign words not in the dictionary should be translated as "eh".

### 样例输入

```
dog ogday
cat atcay
pig igpay
froot ootfray
loops oopslay

atcay
ittenkay
oopslay
```

### 样例输出

```
cat
eh
loops
```

### 分析

最简单的方法是，把输入的单词对，存放在一个数组，排好序，查找的时候每次进行二分查找。

更快的方法是，用 HashMap。C++ 有 `map`，C++ 11 以后有 `unordered_map`，比 `map` 快。也可以自己实现哈希表。

### 代码

使用 `map` 。

babelfish_map.cpp

```
/* POJ 2503 Babelfish , http://poj.org/problem?id=2503 */
#include <cstdio>
#include <map>
```

```
#include <string>
#include <cstring>

using namespace std;

/** 字符串最大长度 */
const int MAX_WORD_LEN = 10;

int main() {
    char line[MAX_WORD_LEN * 2 + 1];
    char s1[MAX_WORD_LEN + 1], s2[MAX_WORD_LEN + 1];
    map<string, string> dict;

    while (gets(line) && line[0] != 0) {
        sscanf(line, "%s %s", s1, s2);
        dict[s2] = s1;
    }

    while (gets(line)) {
        if (dict[line].length() == 0) puts("eh");
        else printf("%s\n", dict[line].c_str());
    }
    return 0;
}
```

babelfish_map.cpp

自己实现哈希表。

babelfish.cpp

```
/* POJ 2503 Babelfish , http://poj.org/problem?id=2503 */
#include <cstring>

/** 单词最大长度. */
#define MAX_WORD_LEN   11

/** 词典中的一对. */
struct dict_pair_t {
    char english[MAX_WORD_LEN];
    char foreign[MAX_WORD_LEN];
};

/** 哈希表容量，一定要大于元素最大个数  */
#define HASH_SET_CAPACITY  100001
/** 哈希表取模的质数，也即哈希桶的个数，小于 HASH_SET_CAPACITY. */
#define PRIME  99997

int elem_hash(const dict_pair_t &e) {
    const char *str = e.foreign;
    unsigned int r = 0;
    int len = strlen(str);
```

```
    int i;

    for (i = 0; i < len; i++) {
        r = (r * 31 + str[i]) % PRIME;
    }
    return r % PRIME;
}

bool operator==(const dict_pair_t &e1, const dict_pair_t &e2) {
    return strcmp(e1.foreign, e2.foreign) == 0;
}

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "hash_set.hpp"  /* 见“查找-> 哈希表”这节 */

/* 跟 find() 略有不同，专为为本题定制 */
template<typename elem_t>
bool hash_set<elem_t>::get(elem_t &e) {
    for (int i = head[elem_hash(e)]; i != -1; i = node[i].next)
        if (e == node[i].elem) {
            // 多了一行代码，获取对应的 english
            strncpy(e.english, node[i].elem.english, MAX_WORD_LEN-1);
            return true;
        }

    return false;
}


int main() {
    char line[MAX_WORD_LEN * 2];
    hash_set<dict_pair_t> set(PRIME, HASH_SET_CAPACITY);
    dict_pair_t e;

    while (gets(line) && line[0] != 0) {
        sscanf(line, "%s %s", e.english, e.foreign);
        set.insert(e);
    }
    while (gets(e.foreign)) {
        if (set.get(e)) printf("%s\n", e.english);
        else printf("eh\n");
    }
    return 0;
}
```

babelfish.cpp

## 相关的题目

与本题相同的题目：

- POJ 2503 Babelfish, http://poj.org/problem?id=2503

与本题相似的题目：

- 无