Cracking the Coding Interview C++11

張世明 (simon.zhangsm@gmail.com)

https://github.com/simonzhangsm/crackcoding

2015-1-28

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/crackcoding

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 ϵ=`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

1.1 数据结构与算法汇总

1、常见数据结构

线性：数组，链表，队列，堆栈，块状数组（数组 + 链表），hash 表，双端队列，位图（bitmap）

树：堆（大顶堆、小顶堆），trie 树（字母树 or 字典树），后缀树，后缀树组，二叉排序/查找树，B+/B-，AVL 树，Treap，红黑树，splay 树，线段树，树状数组图：图

其它：并查集

2、常见算法

（1）基本思想：枚举，递归，分治，模拟，贪心，动态规划，剪枝，回溯

（2）图算法：深度优先遍历与广度优先遍历，最短路径，最小生成树，拓扑排序

（3）字符串算法：字符串查找，hash 算法，KMP 算法

（4）排序算法：冒泡，插入，选择，快排，归并排序，堆排序，桶排序

（5）动态规划：背包问题，最长公共子序列，最优二分检索树

（6）数论问题：素数问题，整数问题，进制转换，同余模运算，

（7）排列组合：排列和组合算法

（8）其它：LCA 与 RMQ 问题

第 2 章
线性表

线性表 (Linear List) 包含：

- 顺序存储：数组 (vector/deque/array/set)
- 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 $[1, 2]$.

解题思路

二指针问题, 一前一后扫描。

Algorithm1

```
// 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;
    }
};
```

Algorithm2

```
// 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));
    }
};
```

Algorithm3

```
// 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]`

解题思路

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。

Algorithm1

```
// 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 {
    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;
    }
};
```

Algorithm2

下面是一个更简洁的版本。上面的 Algorithm 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 {
```

```
    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.

解题思路

二分查找，难度主要在于左右边界的确定。

Algorithm

```
// LeetCode, Search in Rotated Sorted Array
// 时间复杂度 O(log n)，空间复杂度 O(1)
class Solution {
    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.

解题思路

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。

Algorithm

```
// LeetCode, Search in Rotated Sorted Array II
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    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 $O(\log(m+n))$.

解题思路

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]`

Algorithm

```
// 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 {
    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);
        return (find_kth(A, m, B, n, total / 2) + find_kth(A, m, B, n, total / 2 + 1)) / 2.0;
    }

    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.

解题思路

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。

Algorithm

```
// Leet Code, Longest Consecutive Sequence
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    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`

Algorithm

```
// Leet Code, Longest Consecutive Sequence
// 时间复杂度 O(n)，空间复杂度 O(n)
// Author: @advancedxy
class Solution {
    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;
    }

    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 kSum

描述

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

Find k elements in set A, where $\sum_{i=0}^{k} A_i = target$

解题思路

For even k, Time: $O(n^{\frac{k}{2}}log(n))$– Compute a sorted list S of all sum of $\frac{k}{2}$ elements in A. Check whether S contains two elements that sum to target.

For odd k, Time: $O(n^{\frac{k+1}{2}})$ – Compute a sorted list S of all sum of $\frac{k-1}{2}$ elements in A. For each input element a in A, check whether S contains s and s', where $a + s + s' = target$

k≥2	Brute Force	Sort Find	Hash	
			Time	Space
2Sum	n^2	$nlog(n)$	n	n
3Sum	n^3	n^2	n^2	n
4Sum	n^4	n^3	$n^2log(n)$	n^2

方法 1：Brute Force: Find all k pairs of numbers and calculate their sum, if it equals the target return their index in increasing order. Running time = $O(n^k)$.

方法 2： Sort + Double Pointer: First, sort the array in $O(nlog(n))$ time. Then use two pointers p and q, p scans from left to right, and q scans from right to left.unning time = $O(n \log n)$

方法 3：Hash Map: We can do even better, by using hash map. First we create a hash map, where $[key, value] = [target - A[i], i]$. This requires $O(n)$. Then, we iterate the array. If $A[j]$ contains in map, which means $A[j] = target - A[i]$, we return i and j. This solution has a running time $O(n)$, and its space complicity is $O(n)$. We used $O(n)$ space to reduce the running time. Note, one number may be used twice, therefore, we need to check $j! = map.get(A[j])$

Note： Handle duplicate elements in the result.

Algorithm

```
//LeetCode, KSum
// 方法 3: 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 {
    vector< vector<int> > KSum(vector<int> &sortednum, int K, int target, int p) {
        vector< vector<int> > vecResults;
        if (K == 2) { // base case
            vector<int> tuple(2, 0);
            int i = p, j = sortednum.size() - 1;
            while (i < j) {
                if (i > p && sortednum[i] == sortednum[i - 1]) {
                    ++i;
```

```
                    continue;
                }
                int sum = sortednum[i] + sortednum[j];
                if (sum == target) {
                    tuple[0] = sortednum[i++];
                    tuple[1] = sortednum[j--];
                    vecResults.push_back(tuple);
                }
                else if (sum > target) {
                    --j;
                }
                else {
                    ++i;
                }
            }
            return vecResults;
        }
        // K > 2
        for (int i = p; i < sortednum.size(); ++i) {
            if (i > p && sortednum[i] == sortednum[i - 1]) continue;
            vector< vector<int> > K1Sum = KSum(sortednum, K - 1, target - sortednum[i], i + 1);
            for (auto it = K1Sum.begin(); it != K1Sum.end(); ++it) {
                vector<int> tuple;
                tuple.push_back(sortednum[i]);
                tuple.insert(tuple.end(), it->begin(), it->end());
                vecResults.push_back(tuple);
            }
        }
        return vecResults;
    }
};
```

相关题目

- KSum, 见 §2.1.8
- 3Sum, 见 §2.1.9
- 3Sum Closest, 见 §2.1.10
- 4Sum, 见 §2.1.11

2.1.8 2Sum

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`

解题思路

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。

Algorithm

```
//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 {
    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

- KSum, 见 §2.1.7
- 3Sum, 见 §2.1.9
- 3Sum Closest, 见 §2.1.10
- 4Sum, 见 §2.1.11

2.1.9 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)
```

解题思路

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 $O(\max\{n \log n, n^{k-1}\})$。

Algorithm

```
// 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 {
    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

- KSum, 见 §2.1.7
- 2Sum, 见 §2.1.8
- 3Sum Closest, 见 §2.1.10
- 4Sum, 见 §2.1.11

2.1.10 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`).

解题思路

先排序，然后左右夹逼，复杂度 $O(n^2)$。

Algorithm

```
// 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 {
    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

- KSum, 见 §2.1.7
- 2Sum, 见 §2.1.8
- 3Sum, 见 §2.1.9
- 4Sum, 见 §2.1.11

2.1.11 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)
```

解题思路

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 {
    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 {
    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 {
    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 {
        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

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

2.1.12 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.

解题思路

无

Algorithm1

```
// LeetCode, Remove Element
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    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;
    }
};
```

Algorithm2

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

相关题目

- 无

2.1.13 Next Permutation

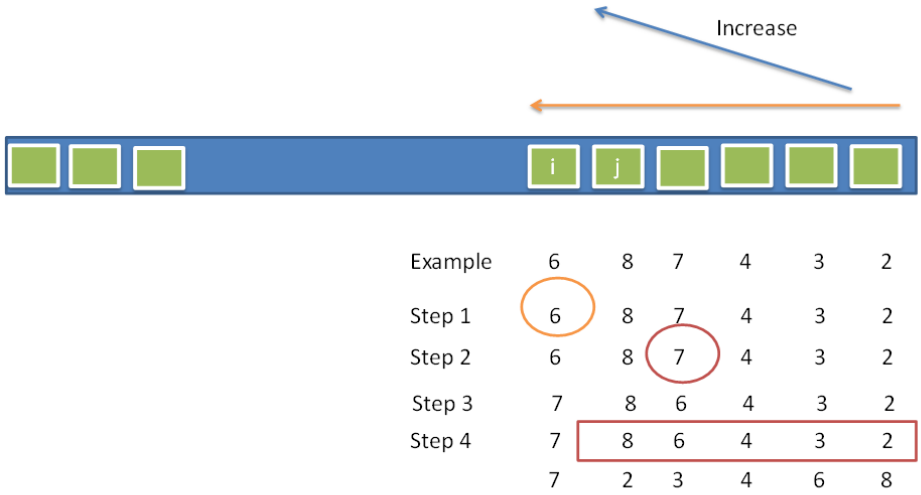
描述

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
```

解题思路

算法过程如图 18-1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

Algorithm

```
// LeetCode, Next Permutation
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    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 §2.1.14
- Permutations, 见 §??
- Permutations II, 见 §??
- Combinations, 见 §??

2.1.14 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.

解题思路

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 {
    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) {
            // Algorithm 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 {
        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);
        }

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

2.1.15 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 `'.'`.

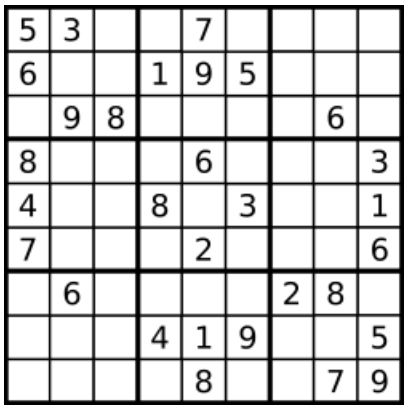


图 2-2 A partially filled sudoku which is valid

解题思路

细节实现题。

Algorithm

```
// LeetCode, Valid Sudoku
// 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
    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 §??

2.1.16 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

解题思路

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。

Algorithm1

```
// LeetCode, Trapping Rain Water
// 思路 1，时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    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;
    }
};
```

Algorithm2

```
// LeetCode, Trapping Rain Water
// 思路 2，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    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;
    }
};
```

Algorithm3

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 {
    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) break; // 碰到了比当前元素高的，跳出循环
                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 §??

2.1.17 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?

解题思路

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。

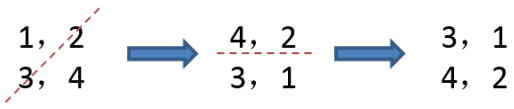


图 2-4 Rotate Image

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。

Algorithm1

```
// LeetCode, Rotate Image
// 思路 1, 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
    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]);
    }
};
```

Algorithm2

```
// LeetCode, Rotate Image
// 思路 2, 时间复杂度 O(n^2), 空间复杂度 O(1)
class Solution {
    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

2.1.18 Plus One

描述

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

解题思路

高精度加法。

Algorithm1

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

    // 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);
    }
};
```

Algorithm2

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

    // 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

2.1.19 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?

解题思路

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 {
    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 {
    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 §??

2.1.20 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.

解题思路

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：$g_0 = b_0, g_i = b_i \oplus b_{i-1}$, here, $\oplus \equiv \wedge$.

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。

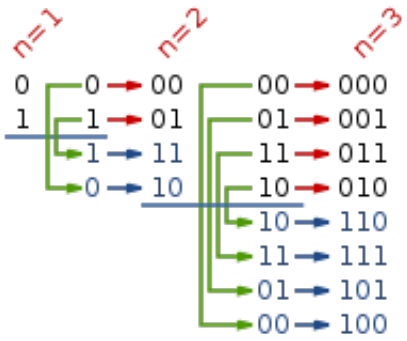


图 2-5 The first few steps of the reflect-and-prefix method.

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 {
    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;
    }
```

```
    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 {
    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;
    }
};
```

```
// LeetCode, Gray Code, By Simon Zhang
// reflect-and-prefix method
// 时间复杂度 O(2^n)，空间复杂度 O(1),
vector<int> grayCode(int n) {
    vector<int> res;
    res.push_back(0);
    for(int i=0;i<n;i++){
        int highbit = 1<<i;
        int curlen = res.size();
        for(int j=curlen-1;j>=0;j--)
        res.push_back(highbit | res[j]);
    }
    return res;
}
```

相关题目

- 无

2.1.21 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?

解题思路

$O(m + n)$ 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。

Algorithm1

```
// LeetCode, Set Matrix Zeroes
// 时间复杂度 O(m*n)，空间复杂度 O(m+n)
class Solution {
    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;
    }
};
```

Algorithm2

```
// LeetCode, Set Matrix Zeroes
// 时间复杂度 O(m*n)，空间复杂度 O(1)
class Solution {
    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

2.1.22 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.

解题思路

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。

Algorithm

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

2.1.23 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?

解题思路

无

迭代版

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

2.1.24 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?

解题思路

异或，不仅能处理两次的情况，只要出现偶数次，都可以清零。

Algorithm1

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

Algorithm2

```
// LeetCode, Single Number
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    int singleNumber(int A[], int n) {
        return accumulate(A, A + n, 0, bit_xor<int>());
    }
};
```

相关题目

- Single Number II, 见 §2.1.25

2.1.25 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?

解题思路

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。

Algorithm1

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

Algorithm2

```
// LeetCode, Single Number II
// 方法 2，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    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 §2.1.24

2.1.26 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

2.1.27 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?

解题思路

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

2.1.28 Word Search

描述

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where ”adjacent” cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example, Given board =

[

[”ABCE”],

[”SFCS”],

[”ADEE”]

]

word = ”ABCCED”, -> returns true,

word = ”SEE”, -> returns true,

word = ”ABCB”, -> returns false. Anyone knowing this question or the solution?

解题思路

The idea of this question is as follows:

1. Find the 1st element of the word in the board.
2. For each position found where the 1st element lies, recursively do:
   i) Search the around cell to see if the next element exists. (4 directions: $(i-1,j),(i+1,j),(i,j-1),(i,j+1)$)
   ii) If the word ends, return true.
3. Return false if no matching found.

Note: A mask matrix is needed to store the positions where have already been visited. Details can be found in code.

Algorithm

```
// LeetCode, Word Search
class Solution {
    bool search(vector<vector<char> > &board, int i, int j, string &word, int idx){
        if(idx == word.size()) return true;
        if(i<board.size() && j<board[i].size() && i >= 0 && j >= 0 && board[i][j] == word[idx]){
            char c = board[i][j];
            board[i][j] = '#';
            if(search(board, i+1, j, word, idx+1)) return true;
            if(search(board, i-1, j, word, idx+1)) return true;
            if(search(board, i, j+1, word, idx+1)) return true;
            if(search(board, i, j-1, word, idx+1)) return true;
            board[i][j] = c;
        }
        return false;
    }

    bool exist(vector<vector<char> > &board, string word) {
        if(board.empty()||board[0].empty()) return false;
        if(word.empty()) return true;
        for(int i=0; i<board.size();i++){
            for(int j=0;j<board[i].size();j++){
                if (board[i][j] == word[0]){
```

```
                    vector<vector<char> > tmp (board);
                    if(search(tmp, i, j, word, 0))
                        return true;
                }
            }
        }
        return false;
    }
};
```

相关题目

- Robot Unique Paths, 见 §2.1.29

2.1.29 Robot Unique Paths

描述

A robot is located at the top-left corner of a mxn grid (marked 'Start' in the diagram below).

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

How many possible unique paths are there?

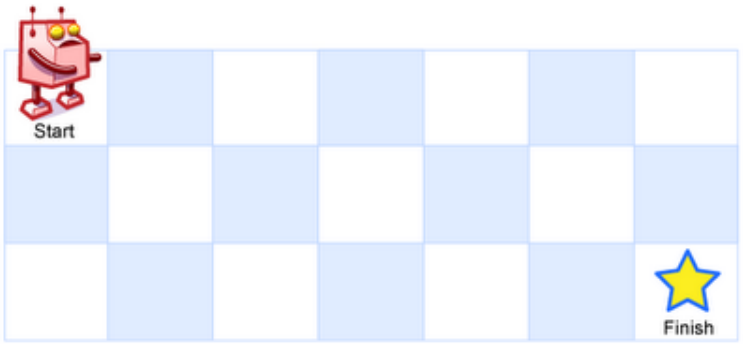


图 2-6 Robot Unique Paths

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

解题思路

一维 DP。Step[i][j] = Step[i-1][j] + Step[i][j-1];

Algorithm

```
// LeetCode, Robot Unique Paths
class Solution {
    int uniquePaths(int m, int n) {
        vector<int> dp(n,0);
        dp[0] = 1;
        for(int i=0;i<m;i++){
            for(int j=1;j<n;j++)
                dp[j] = dp[j-1] + dp[j];
        }
        return dp[n-1];
    }
};
```

相关题目

- Word Search, 见 §2.1.28

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 Fabonacci 数列

Fabonacci 数列为：$a_0 = 1, a_1 = 1, a_n = a_{n-1} + a_{n-2}$ 快速幂优化

2.2.2 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`

解题思路

跟 Add Binary（见 §3.12.5）很类似

Algorithm

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

相关题目

- Add Binary, 见 §3.12.5

2.2.3 Reverse Linked List II

描述

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

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

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

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

解题思路

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!

Algorithm

```
// LeetCode, Reverse Linked List II
// 迭代版，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    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

2.2.4 Partition List

描述

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

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

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

解题思路

无

Algorithm

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

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

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

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

        return left_dummy.next;
    }
};
```

相关题目

- 无

2.2.5 Remove Duplicates from Sorted List

描述

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

For example,

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

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

解题思路

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 {
    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 {
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return nullptr;

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

相关题目

- Remove Duplicates from Sorted List II，见 §2.2.6

2.2.6 Remove Duplicates from Sorted List II

描述

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

For example,

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

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

解题思路

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 {
    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 {
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return head;

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

相关题目

- Remove Duplicates from Sorted List，见 §2.2.5

2.2.7 Rotate List

描述

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

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

解题思路

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。

Algorithm

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

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

        p->next = head; // 首尾相连
        for(int step = 0; step < k; step++)
            p = p->next;  //接着往后跑

        head = p->next; // 新的首节点
        p->next = nullptr; // 断开环
        return head;
    }
};
```

相关题目

- 无

2.2.8 Remove Nth Node From End of List

描述

Given a linked list, remove the n^{th} node from the end of list and return its head.

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

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

Note:

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

解题思路

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。

Algorithm

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

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

        while(q->next) { // 一起走
            p = p->next;
            q = q->next;
        }
```

```
            ListNode *tmp = p->next;
            p->next = p->next->next;
            delete tmp;
            return dummy.next;
        }
    };
```

相关题目

- 无

2.2.9 Swap Nodes in Pairs

描述

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

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

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

解题思路

无

Algorithm

```
// LeetCode, Swap Nodes in Pairs
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    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 {
    ListNode* swapPairs(ListNode* head) {
        ListNode* p = head;
        while (p && p->next) {
            swap(p->val, p->next->val);
            p = p->next->next;
        }
        return head;
    }
};
```

相关题目

- Reverse Nodes in k-Group, 见 §2.2.10

2.2.10 Reverse Nodes in k-Group

描述

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

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

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

Only constant memory is allowed.

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

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

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

解题思路

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 {
    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 {
    ListNode *reverseKGroup(ListNode *head, int k) {
        if (head == nullptr || head->next == nullptr || k < 2) return head;
        ListNode dummy(-1);
        dummy.next = head;

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

```
        }
        return dummy.next;
    }

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

相关题目

- Swap Nodes in Pairs, 见 §2.2.9

2.2.11 Copy List with Random Pointer

描述

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

Return a deep copy of the list.

解题思路

无

Algorithm

```
// 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 {
    RandomListNode *copyRandomList(RandomListNode *head) {
        for (RandomListNode* cur = head; cur != nullptr; ) {
            RandomListNode* node = new RandomListNode(cur->label);
            node->next = cur->next;
            cur->next = node;
            cur = node->next;
        }

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

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

```
            return dummy.next;
        }
    };
```

相关题目

- 无

2.2.12 Linked List Cycle

描述

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

解题思路

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

Algorithm

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

相关题目

- Linked List Cycle II, 见 §2.2.13

2.2.13 Linked List Cycle II

描述

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

解题思路

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。

Algorithm

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

相关题目

- Linked List Cycle, 见 §2.2.12

2.2.14 Reorder List

描述

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

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

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

解题思路

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。

Algorithm

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

```
            fast = fast->next->next;
        }
        prev->next = nullptr; // cut at middle

        slow = reverse(slow);

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

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

相关题目

- 无

2.2.15 LRU Cache

描述

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

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

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

解题思路

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

Algorithm

```
// 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 §4.4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 后缀数组

3.5 最长重复子串

3.6 最长公共子串

3.7 最长回文子串 manacher 算法

3.8 字符串散列

求一个字符串设计一种散列
一种比较常见的方法是对于第 i 个字符，让它对散列的贡献值表示成 $s[i] * p^i$，其中 p 为一个素数

3.9 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;
    }
```

3.10 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。

3.10.1 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>
#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
```

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

```
 */
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

3.10.2 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 */

/*
```

[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.

```
 * @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
 * @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

3.10.3 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
```

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

```
 * @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;
    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

3.10.4 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

3.11 正则表达式

3.11.1 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.

3.12 LeetCode

3.12.1 Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example,

Given:

s1 = ”aabcc”,

s2 = ”dbbca”,

When s3 = ”aadbbcbcac”, return true.

When s3 = ”aadbbbaccc”, return false.

思路

【Iteration】s1 取一部分，s2 取一部分，然后再 s1 取一部分，反复直到匹配完成 s3，算法去模拟这样的操作。

当 s1 和 s3 匹配了一部分的时候，剩下 s1‘和剩下的 s3’与 s2 又是一个子问题，则递归，但是需要注意两点：

1. 递归中，总是拿 s1 首先去匹配 s3，如果不匹配，直接返回 false，如此保持匹配是“交替”进行的；
2. 当出现既可以匹配 s1，又可以匹配 s2 的时候，一样可以通过递归来解决，看下面的代码。

```
bool isInterleave(string s1, string s2, string s3) {
    if(s3.size()!=s1.size()+s2.size()) return false;
    if(s1.empty()) return s2==s3;
    if(s2.empty()) return s1==s3;
    if(s1[0]!=s3[0]){
        if(s3[0]==s2[0])
```

```
            return isInterleave(s2,s1,s3);
        return false;
    }
    int i1=0, i3=0;
    while(i1<s1.size() && i3<s3.size() && s1[i1]==s3[i3]){
        i1++, i3++;
        if(s2[0]==s3[i3] && isInterleave(s2, s1.substr(i1), s3.substr(i3)))
            return true;
    }
    return isInterleave(s2, s1.substr(i1), s3.substr(i3));
}
```

【Danymic Programming】动态规划矩阵 matched[l1][l2] 表示 s1 取 l1 长度（最后一个字母的 pos 是 l1-1），s2 取 l2 长度 (最后一个字母的 pos 是 l2-1)，是否能匹配 s3 的 l1+l2 长度。

$$match[l1][l2] = s1[l1-1] == s3[l1+l2-1] \&\& match[l1-1][l2] || s2[l2-1] == s3[l1+l2-1] \&\& match[l1][l2-1]$$

边界条件是，其中一个长度为 0，另一个去匹配 s3.

这里 s1 和 s2 交替出现的规律并不明显，所以没有直观地想到。

```
bool isInterleave(string s1, string s2, string s3) {
    if(s3.size()!=s1.size()+s2.size()) return false;
    if(s1.empty()) return s2==s3;
    if(s2.empty()) return s1==s3;
    bool match[s1.size()+1][s2.size()+1];
    memset(match, 0x00, sizeof(match));
    match[0][0] = true;
    for(int i1=1;i1<=s1.size() && s1[i1-1]==s3[i1-1];i1++)
        match[i1][0] = true;
    for(int i2=1;i2<=s2.size() && s2[i2-1]==s3[i2-1];i2++)
        match[0][i2] = true;
    for(int i1=1;i1<=s1.size();i1++){
        for(int i2=1;i2<=s2.size();i2++){
            match[i1][i2] = (s1[i1-1] == s3[i1+i2-1] && match[i1-1][i2])
                        ||    (s2[i2-1] == s3[i1+i2-1] && match[i1][i2-1]);
        }
    }
    return match[s1.size()][s2.size()];
}
```

3.12.2 Valid Palindrome

描述

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

For example,

`"A man, a plan, a canal: Panama"` is a palindrome.

`"race a car"` is not a palindrome.

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

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

分析

无

代码

```
// Leet Code, Valid Palindrome
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
```

```
        bool isPalindrome(string s) {
            transform(s.begin(), s.end(), s.begin(), ::tolower);
            auto left = s.begin(), right = prev(s.end());
            while (left < right) {
                if (!::isalnum(*left))  ++left;
                else if (!::isalnum(*right)) --right;
                else if (*left != *right) return false;
                else{ left++, right--; }
            }
            return true;
        }
    };
```

相关题目

- Palindrome Number, 见 §??

3.12.3 Implement strStr()

描述

Implement strStr().

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

分析

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

暴力匹配

```
    // LeetCode, Implement strStr()
    // 暴力解法，时间复杂度 O(N*M)，空间复杂度 O(1)
    class Solution {
        public:
        char *strStr(const char *haystack, const char *needle) {
            // if needle is empty return the full string
            if (!*needle) return (char*) haystack;

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

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

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

KMP

```
// LeetCode, Implement strStr()
// KMP, 时间复杂度 O(N+M), 空间复杂度 O(M)
class Solution {
    public:
    char *strStr(const char *haystack, const char *needle) {
        int pos = kmp(haystack, needle);
        if (pos == -1) return nullptr;
        else return (char*)haystack + pos;
    }
    private:
    /*
    * @brief 计算部分匹配表, 即 next 数组.
    *
    * @param[in] pattern 模式串
    * @param[out] next next 数组
    * @return 无
    */
    static void compute_prefix(const char *pattern, int next[]) {
        int i;
        int j = -1;
        const int m = strlen(pattern);

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

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

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

        compute_prefix(pattern, next);

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

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

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

```
};
```

相关题目

- String to Integer (atoi)，见 §4.1.4

3.12.4 String to Integer (atoi)

描述

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

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

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

Requirements for atoi:

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

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

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

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

分析

细节题。注意几个测试用例：

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

代码

```
// LeetCode, String to Integer (atoi)
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int atoi(const char *str) {
        int num = 0;
        int sign = 1;
        const int n = strlen(str);
        int i = 0;

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

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

    for (; i < n; i++) {
        if (str[i] < '0' || str[i] > '9')
```

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

相关题目

- Implement strStr()， 见 §3.12.3

3.12.5 Add Binary

描述

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

For example,

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

Return `"100"`.

分析

无

代码

```
//LeetCode, Add Binary
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
    public:
    string addBinary(string a, string b) {
        string result;
        const size_t n = a.size() > b.size() ? a.size() : b.size();
        reverse(a.begin(), a.end());
        reverse(b.begin(), b.end());
        int carry = 0;
        for (size_t i = 0; i < n; i++) {
            const int ai = i < a.size() ? a[i] - '0' : 0;
            const int bi = i < b.size() ? b[i] - '0' : 0;
            const int val = (ai + bi + carry) % 2;
            carry = (ai + bi + carry) / 2;
            result.insert(result.begin(), val + '0');
        }
        if (carry == 1) {
            result.insert(result.begin(), '1');
        }
        return result;
    }
};
```

相关题目

- Add Two Numbers, 见 §2.2.2

3.12.6 Longest Palindromic Substring

描述

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

分析

最长回文子串，非常经典的题。

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

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

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

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

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

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

备忘录法

```
// LeetCode, Longest Palindromic Substring
// 备忘录法，会超时
// 时间复杂度 O(n^2)，空间复杂度 O(n^2)
typedef string::const_iterator Iterator;

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

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

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

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

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

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

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

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

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

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

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

动规

```
// LeetCode, Longest Palindromic Substring
// 动规, 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
    public:
    string longestPalindrome(string s) {
        const int n = s.size();
        bool f[n][n];
        fill_n(&f[0][0], n * n, false);
        // 用 vector 会超时
        //vector<vector<bool> > f(n, vector<bool>(n, false));
        size_t max_len = 1, start = 0;  // 最长回文子串的长度, 起点

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

Manacher's Algorithm

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

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

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

```
    }

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

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

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

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

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

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

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

相关题目

- 无

3.12.7 Regular Expression Matching

描述

Implement regular expression matching with support for `'.'` and `'*'`.

`'.'` Matches any single character. `'*'` Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:

```
bool isMatch(const char *s, const char *p)
```

Some examples:

```
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
```

```
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true
```

分析

这是一道很有挑战的题。

递归版

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

        // next char is not '*', then must match current character
        if (*(p + 1) != '*') {
            if (*p == *s || (*p == '.' && *s != '\0'))
            return isMatch(s + 1, p + 1);
            else
            return false;
        } else { // next char is '*'
        while (*p == *s || (*p == '.' && *s != '\0')) {
            if (isMatch(s, p + 2))
            return true;
            s++;
        }
        return isMatch(s, p + 2);
    }
}
};
```

迭代版

相关题目

- Wildcard Matching, 见 §3.12.8

3.12.8 Wildcard Matching

描述

Implement wildcard pattern matching with support for `'?'` and `'*'`.

`'?'` Matches any single character. `'*'` Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:

```
bool isMatch(const char *s, const char *p)
```

Some examples:

```
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

分析

跟上一题很类似。

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

递归版

```
// LeetCode, Wildcard Matching
// 递归版，会超时，用于帮助理解题意
// 时间复杂度 O(n!*m!)，空间复杂度 O(n)
class Solution {
    public:
    bool isMatch(const char *s, const char *p) {
        if (*p == '*') {
            while (*p == '*') ++p;  //skip continuous '*'
            if (*p == '\0') return true;
            while (*s != '\0' && !isMatch(s, p)) ++s;

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

迭代版

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

相关题目

- Regular Expression Matching, 见 §3.12.7

3.12.9 Longest Common Prefix

描述

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

分析

从位置 0 开始，对每一个位置比较所有字符串，直到遇到一个不匹配。

纵向扫描

```
// LeetCode, Longest Common Prefix
// 纵向扫描，从位置 0 开始，对每一个位置比较所有字符串，直到遇到一个不匹配
// 时间复杂度 O(n1+n2+...)
// @author 周倩 (http://weibo.com/zhouditty)
class Solution {
    public:
    string longestCommonPrefix(vector<string> &strs) {
        if (strs.empty()) return "";

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

横向扫描

```
// LeetCode, Longest Common Prefix
// 横向扫描，每个字符串与第 0 个字符串，从左到右比较，直到遇到一个不匹配，
// 然后继续下一个字符串
// 时间复杂度 O(n1+n2+...)
class Solution {
    public:
    string longestCommonPrefix(vector<string> &strs) {
        if (strs.empty()) return "";

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

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

相关题目

- 无

3.12.10 Valid Number

描述

Validate if a given string is numeric.

Some examples:

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

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

分析

细节实现题。

本题的功能与标准库中的 `strtod()` 功能类似。

有限自动机

```
// LeetCode, Valid Number
// @author 龚陆安 (http://weibo.com/luangong)
// finite automata，时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    bool isNumber(const char *s) {
        enum InputType {
            INVALID,    // 0
            SPACE,      // 1
            SIGN,       // 2
            DIGIT,      // 3
            DOT,        // 4
            EXPONENT,   // 5
            NUM_INPUTS  // 6
        };
        const int transitionTable[][NUM_INPUTS] = {
            -1, 0, 3, 1, 2, -1, // next states for state 0
            -1, 8, -1, 1, 4, 5,     // next states for state 1
            -1, -1, -1, 4, -1, -1,     // next states for state 2
            -1, -1, -1, 1, 2, -1,     // next states for state 3
            -1, 8, -1, 4, -1, 5,     // next states for state 4
            -1, -1, 6, 7, -1, -1,     // next states for state 5
            -1, -1, -1, 7, -1, -1,     // next states for state 6
            -1, 8, -1, 7, -1, -1,     // next states for state 7
            -1, 8, -1, -1, -1, -1,     // next states for state 8
        };

        int state = 0;
        for (; *s != '\0'; ++s) {
            InputType inputType = INVALID;
            if (isspace(*s))
            inputType = SPACE;
            else if (*s == '+' || *s == '-')
            inputType = SIGN;
            else if (isdigit(*s))
            inputType = DIGIT;
            else if (*s == '.')
            inputType = DOT;
            else if (*s == 'e' || *s == 'E')
            inputType = EXPONENT;
```

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

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

    }
};
```

使用 strtod()

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

        if (endptr == s) return false;

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

        return true;
    }
};
```

相关题目

- 无

3.12.11 Integer to Roman

描述

Given an integer, convert it to a roman numeral.

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

分析

无

代码

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

        string roman;
```

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

相关题目

- Roman to Integer, 见 §3.12.12

3.12.12 Roman to Integer

描述

Given a roman numeral, convert it to an integer.

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

分析

从前往后扫描，用一个临时变量记录分段数字。

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

代码

```
    // LeetCode, Roman to Integer
    // 时间复杂度 O(n)，空间复杂度 O(1)
    class Solution {
        public:
        inline int map(const char c) {
            switch (c) {
                case 'I': return 1;
                case 'V': return 5;
                case 'X': return 10;
                case 'L': return 50;
                case 'C': return 100;
                case 'D': return 500;
                case 'M': return 1000;
                default: return 0;
            }
        }

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

相关题目

- Integer to Roman, 见 §3.12.11

3.12.13 Count and Say

描述

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

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

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

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

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

Given an integer n, generate the nth sequence.

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

分析

模拟。

代码

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

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

        return s;
    }

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

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

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

相关题目

- 无

3.12.14 Anagrams

描述

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

Note: All inputs will be in lower-case.

分析

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

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

代码

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

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

相关题目

- 无

3.12.15 Simplify Path

描述

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

For example,

path = "/home/", => "/home"

path = "/a/./b/../../c/", => "/c"

Corner Cases:

- Did you consider the case where path = "/../"? In this case, you should return "/".
- Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/". In this case, you should ignore redundant slashes and return "/home/foo".

分析

很有实际价值的题目。

代码

```
// LeetCode, Simplify Path
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
    public:
```

```
        string simplifyPath(string const& path) {
            vector<string> dirs; // 当做栈

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

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

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

                i = j;
            }

            stringstream out;
            if (dirs.empty()) {
                out << "/";
            } else {
            for (auto dir : dirs)
            out << '/' << dir;
        }

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

相关题目

- 无

3.12.16 Length of Last Word

描述

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

If the last word does not exist, return 0.

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

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

分析

细节实现题。

用 STL

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

```
            return distance(first, last);
        }
    };
```

顺序扫描

```
    // LeetCode, Length of Last Word
    // 顺序扫描，记录每个 word 的长度
    // 时间复杂度 O(n)，空间复杂度 O(1)
    class Solution {
        public:
        int lengthOfLastWord(const char *s) {
            int len = 0;
            while (*s) {
                if (*s++ != ' ')
                ++len;
                else if (*s && *s != ' ')
                len = 0;
            }
            return len;
        }
    };
```

相关题目

- 无

第 4 章
Interval

4.1 Interval Class

```
template <class T, typename K = int>
class Interval {
public:
    K start;
    K stop;
    T value;
    Interval(K s, K e, const T& v):start(s),stop(e),value(v)
    { }
};

template <class T, typename K>
int intervalStart(const Interval<T,K>& i) {
    return i.start;
}

template <class T, typename K>
int intervalStop(const Interval<T,K>& i) {
    return i.stop;
}

template <class T, typename K>
std::ostream& operator<<(std::ostream& out, Interval<T,K>& i) {
    out << "Interval(" << i.start << ", " << i.stop << "): " << i.value;
    return out;
}

template <class T, typename K = int>
class IntervalStartSorter {
public:
    bool operator() (const Interval<T,K>& a, const Interval<T,K>& b) {
        return a.start < b.start;
    }
};

template <class T, typename K = int>
class IntervalTree {

public:
    typedef Interval<T,K> interval;
    typedef std::vector<interval> intervalVector;
    typedef IntervalTree<T,K> intervalTree;

    intervalVector intervals;
    intervalTree* left;
    intervalTree* right;
    int center;

    IntervalTree<T,K>(void)
    : left(NULL)
    , right(NULL)
```

```
        , center(0)
        { }

        IntervalTree<T,K>(const intervalTree& other)
        : left(NULL)
        , right(NULL)
        {
            center = other.center;
            intervals = other.intervals;
            if (other.left) {
                left = new intervalTree(*other.left);
            }
            if (other.right) {
                right = new intervalTree(*other.right);
            }
        }

        IntervalTree<T,K>& operator=(const intervalTree& other) {
            center = other.center;
            intervals = other.intervals;
            if (other.left) {
                left = new intervalTree(*other.left);
            } else {
                if (left) delete left;
                left = NULL;
            }
            if (other.right) {
                right = new intervalTree(*other.right);
            } else {
                if (right) delete right;
                right = NULL;
            }
            return *this;
        }

    IntervalTree<T,K>(
        intervalVector& ivals,
        unsigned int depth = 16,
        unsigned int minbucket = 64,
        int leftextent = 0,
        int rightextent = 0,
        unsigned int maxbucket = 512
        ):left(NULL), right(NULL){

            --depth;
            IntervalStartSorter<T,K> intervalStartSorter;
            if (depth == 0 || (ivals.size() < minbucket && ivals.size() <
            maxbucket)) {
                std::sort(ivals.begin(), ivals.end(), intervalStartSorter);
                intervals = ivals;
            } else {
                if (leftextent == 0 && rightextent == 0) {
                // sort intervals by start
                std::sort(ivals.begin(), ivals.end(), intervalStartSorter);
                }

                int leftp = 0;
                int rightp = 0;
                int centerp = 0;

                if (leftextent || rightextent) {
                    leftp = leftextent;
                    rightp = rightextent;
                } else {
```

```
                leftp = ivals.front().start;
                std::vector<K> stops;
                stops.resize(ivals.size());
                transform(ivals.begin(), ivals.end(), stops.begin(),
                intervalStop<T,K>);
                rightp = *max_element(stops.begin(), stops.end());
            }

            //centerp = ( leftp + rightp ) / 2;
            centerp = ivals.at(ivals.size() / 2).start;
            center = centerp;

            intervalVector lefts;
            intervalVector rights;

            for (typename intervalVector::iterator i = ivals.begin(); i !=
        ivals.end(); ++i) {
                interval& interval = *i;
                if (interval.stop < center) {
                    lefts.push_back(interval);
                } else if (interval.start > center) {
                    rights.push_back(interval);
                } else {
                    intervals.push_back(interval);
                }
            }

            if (!lefts.empty()) {
                left = new intervalTree(lefts, depth, minbucket, leftp,
            centerp);
            }
            if (!rights.empty()) {
                right = new intervalTree(rights, depth, minbucket, centerp,
            rightp);
            }
    }
}

void findOverlapping(K start, K stop, intervalVector& overlapping) const {
    if (!intervals.empty() && ! (stop < intervals.front().start)) {
        for (typename intervalVector::const_iterator i = intervals.begin();
        i!= intervals.end(); ++i) {
            const interval& interval = *i;
            if (interval.stop >= start && interval.start <= stop) {
                overlapping.push_back(interval);
            }
        }
    }

    if (left && start <= center) {
        left->findOverlapping(start, stop, overlapping);
    }

    if (right && stop >= center) {
        right->findOverlapping(start, stop, overlapping);
    }
}

void findContained(K start, K stop, intervalVector& contained) const {
    if (!intervals.empty() && ! (stop < intervals.front().start)) {
        for (typename intervalVector::const_iterator i = intervals.begin();
        i != intervals.end(); ++i) {
            const interval& interval = *i;
            if (interval.start >= start && interval.stop <= stop) {
```

```
                    contained.push_back(interval);
                }
            }
        }

        if (left && start <= center) {
            left->findContained(start, stop, contained);
        }

        if (right && stop >= center) {
            right->findContained(start, stop, contained);
        }

    }

    ~IntervalTree(void) {
        // traverse the left and right
        // delete them all the way down
        if (left) {
            delete left;
        }
        if (right) {
            delete right;
        }
    }
};
```

4.1.1 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);
}
```

4.1.2 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;
}
```

4.1.3 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 §4.4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

4.1.4 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;
}
```

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

4.2 字符串排序

4.3 单词查找树

4.4 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。

4.4.1 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>
#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];
```

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

```
        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

4.4.2 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
 * @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;
}
```

[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.

```
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

4.4.3 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
```

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

```
 * @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;
    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

4.4.4 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

4.5 正则表达式

第 5 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）。

5.1 栈

5.1.1 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 5-1所示。

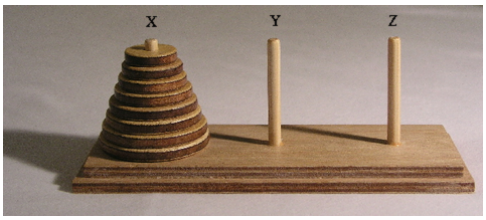


图 5-1　Hanoi 塔问题

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

5.1.2 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

5.1.3 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.

5.1.4 Valid Parentheses

描述

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]" are all valid but "(]" and "([)]" are not.

分析

无

代码

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

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

相关题目

- Generate Parentheses, 见 §??

- Longest Valid Parentheses, 见 §??

5.1.5 Longest Valid Parentheses

描述

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

分析

无

使用栈

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

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

Dynamic Programming, One Pass

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

```
                    f[i] = f[i + 1] + 2;
                    // if a valid sequence exist afterwards "((...))()"
                    if (match + 1 < s.size()) f[i] += f[match + 1];
                }
                ret = max(ret, f[i]);
            }
            return ret;
        }
    };
```

两遍扫描

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

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

相关题目

- Valid Parentheses, 见 §??
- Generate Parentheses, 见 §??

5.1.6 Largest Rectangle in Histogram

描述

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

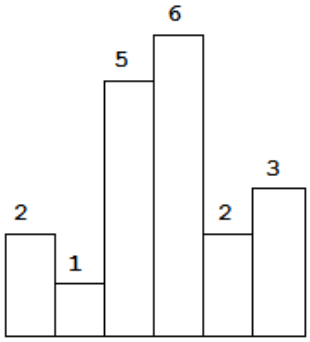


图 5-2 Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.

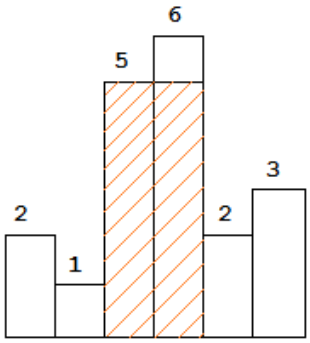


图 5-3 The largest rectangle is shown in the shaded area, which has area = 10 unit.

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

分析

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

如图 §??所示，从左到右处理直方，当 $i = 4$ 时，小于当前栈顶（即直方 3），对于直方 3，无论后面还是前面的直方，都不可能得到比目前栈顶元素更高的高度了，处理掉直方 3（计算从直方 3 到直方 4 之间的矩形的面积，然后从栈里弹出）；对于直方 2 也是如此；直到碰到比直方 4 更矮的直方 1。

这就意味着，可以维护一个递增的栈，每次比较栈顶与当前元素。如果当前元素大于栈顶元素，则入栈，否则合并现有栈，直至栈顶元素小于当前元素。结尾时入栈元素 0，重复合并一次。

代码

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

```
                else {
                    int tmp = s.top();
                    s.pop();
                    result = max(result,
                    height[tmp] * (s.empty() ? i : i - s.top() - 1));
                }
            }
            return result;
        }
    };
```

相关题目

- Trapping Rain Water, 见 §2.1.16
- Container With Most Water, 见 §??

5.1.7 Evaluate Reverse Polish Notation

描述

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

Some examples:

```
["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

分析

无

递归版

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

迭代版

```
    // LeetCode, Max Points on a Line
    // 迭代，时间复杂度 O(n)，空间复杂度 O(logn)
    class Solution {
        public:
        int evalRPN(vector<string> &tokens) {
            stack<string> s;
            for (auto token : tokens) {
                if (!is_operator(token)) {
                    s.push(token);
                } else {
                int y = stoi(s.top());
                s.pop();
                int x = stoi(s.top());
                s.pop();
                if (token[0] == '+')       x += y;
                else if (token[0] == '-')  x -= y;
                else if (token[0] == '*')  x *= y;
                else                       x /= y;
                s.push(to_string(x));
            }
        }
        return stoi(s.top());
    }
    private:
    bool is_operator(const string &op) {
        return op.size() == 1 && string("+-*/").find(op) != string::npos;
    }
};
```

相关题目

- 无

5.2 队列

5.2.1 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 6 章
树

二叉树的节点定义如下：

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

6.1 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 §??
- 3Sum, 见 §2.1.9
- 3Sum Closest, 见 §2.1.10
- 4Sum, 见 §2.1.11

6.2 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

6.3 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 6-1所示。

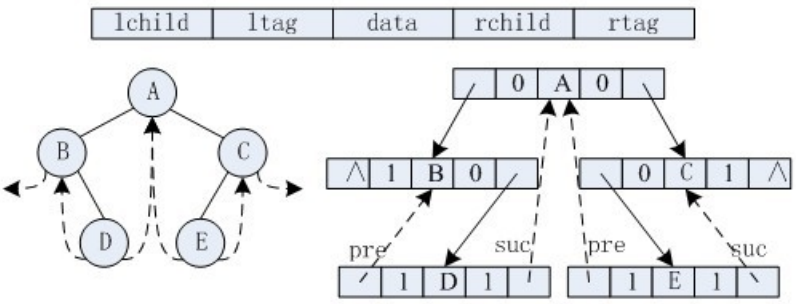


图 6-1 中序线索二叉树

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。

6.4 Morris Traversal

通过前面第 §6.2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。

6.4.1 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 6-2所示，cur 表示当前节点，深色节点表示该节点已输出。

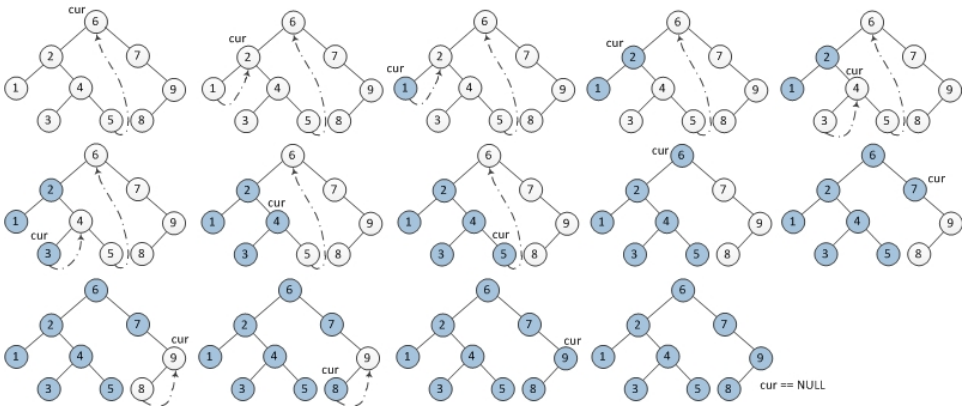


图 6-2 Morris 中序遍历

C 语言实现见第 §6.4.4节。

相关的题目

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

6.4.2 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 6-3所示。

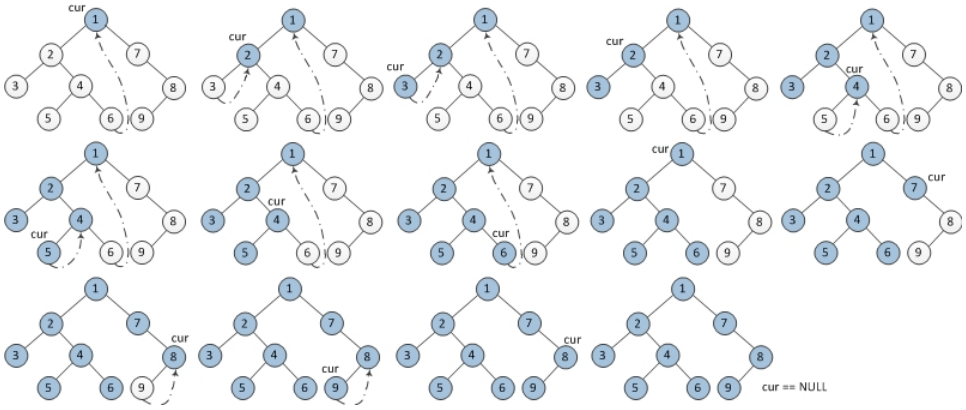


图 6-3 Morris 先序遍历

C 语言实现见第 §6.4.4节。

6.4.3 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 6-4所示。

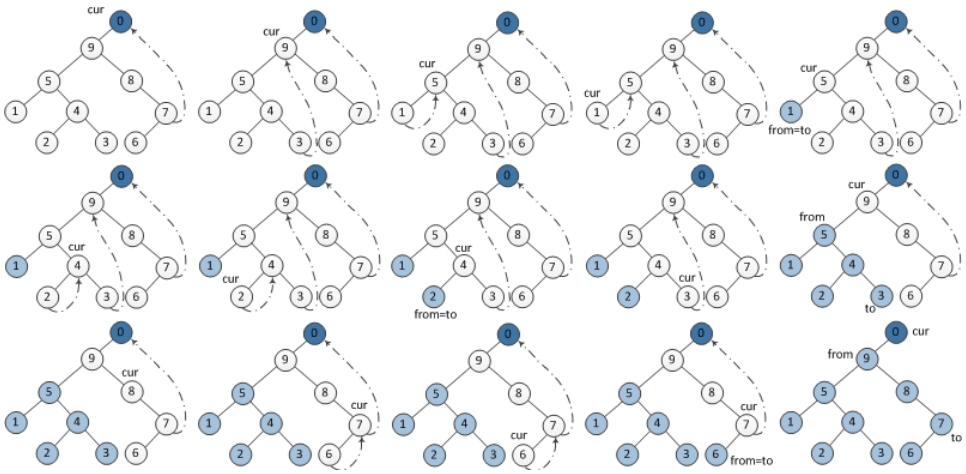


图 6-4 Morris 后序遍历

C 语言实现见第 §6.4.4节。

6.4.4 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

6.5 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

6.6 二叉树的遍历

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

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

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

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

6.6.1 Binary Tree Preorder Traversal

描述

Given a binary tree, return the preorder traversal of its nodes' values.

For example: Given binary tree `{1,#,2,3}`,

```
1
\
2
/
3
```

return `[1,2,3]`.

Note: Recursive solution is trivial, could you do it iteratively?

分析

用栈或者 Morris 遍历。

栈

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

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

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

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

Morris 先序遍历

```
// LeetCode, Binary Tree Preorder Traversal
// Morris 先序遍历，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode *cur, *prev;

        cur = root;
        while (cur != nullptr) {
            if (cur->left == nullptr) {
                result.push_back(cur->val);
                prev = cur; /* cur 刚刚被访问过 */
                cur = cur->right;
            } else {
            /* 查找前驱 */
```

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

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

相关题目

- Binary Tree Inorder Traversal，见 §??
- Binary Tree Postorder Traversal，见 §??
- Recover Binary Search Tree，见 §??

6.6.2 Binary Tree Inorder Traversal

描述

Given a binary tree, return the inorder traversal of its nodes' values.

For example: Given binary tree `{1,#,2,3}`,

```
1
\
2
/
3
```

return `[1,3,2]`.

Note: Recursive solution is trivial, could you do it iteratively?

分析

用栈或者 Morris 遍历。

栈

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

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

```
                p = s.top();
                s.pop();
                result.push_back(p->val);
                p = p->right;
            }
        }
        return result;
    }
};
```

Morris 中序遍历

```
    // LeetCode, Binary Tree Inorder Traversal
    // Morris 中序遍历，时间复杂度 O(n)，空间复杂度 O(1)
    class Solution {
        public:
        vector<int> inorderTraversal(TreeNode *root) {
            vector<int> result;
            TreeNode *cur, *prev;

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

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

相关题目

- Binary Tree Preorder Traversal，见 §??
- Binary Tree Postorder Traversal，见 §??
- Recover Binary Search Tree，见 §??

6.6.3 Binary Tree Postorder Traversal

描述

Given a binary tree, return the postorder traversal of its nodes' values.

For example: Given binary tree `{1,#,2,3}`,

```
1
\
2
/
3
```

return `[3,2,1]`.

Note: Recursive solution is trivial, could you do it iteratively?

分析

用栈或者 Morris 遍历。

栈

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

        p = root;

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

        return result;
    }
};
```

Morris 后序遍历

```
// LeetCode, Binary Tree Postorder Traversal
// Morris 后序遍历，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode dummy(-1);
```

```
            TreeNode *cur, *prev = nullptr;
            std::function < void(const TreeNode*)> visit =
            [&result](const TreeNode *node){
                result.push_back(node->val);
            };

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

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

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

// 访问逆转后的路径上的所有结点
static void visit_reverse(TreeNode* from, TreeNode *to,
std::function< void(const TreeNode*) >& visit) {
    TreeNode *p = to;
    reverse(from, to);

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

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

相关题目

- Binary Tree Preorder Traversal，见 §??
- Binary Tree Inorder Traversal，见 §??
- Recover Binary Search Tree，见 §??

6.6.4 Binary Tree Level Order Traversal

描述

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

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

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

return its level order traversal as:

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

分析

无

递归版

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

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

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

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

迭代版

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

```
        if(root == nullptr) return result;

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

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

相关题目

- Binary Tree Level Order Traversal II，见 §??
- Binary Tree Zigzag Level Order Traversal，见 §??

6.6.5 Binary Tree Level Order Traversal II

描述

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

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

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

return its bottom-up level order traversal as:

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

分析

在上一题（见 §??）的基础上，`reverse()` 一下即可。

递归版

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

```
        return result;
    }

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

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

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

迭代版

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

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

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

相关题目

- Binary Tree Level Order Traversal，见 §??
- Binary Tree Zigzag Level Order Traversal，见 §??

6.6.6 Binary Tree Zigzag Level Order Traversal

描述

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

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

```
3
/ \
9  20
```

```
/  \
15   7
```

return its zigzag level order traversal as:

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

分析

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

递归版

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

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

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

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

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

迭代版

```
//LeetCode, Binary Tree Zigzag Level Order Traversal
//广度优先遍历，用一个 bool 记录是从左到右还是从右到左，每一层结束就翻转一下。
// 迭代版，时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    vector<vector<int> > zigzagLevelOrder(TreeNode *root) {
        vector<vector<int> > result;
        if (nullptr == root) return result;

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

        q.push(root);
        q.push(nullptr);  // level separator
        while (!q.empty()) {
            TreeNode *cur = q.front();
            q.pop();
```

```
                if (cur) {
                    level.push_back(cur->val);
                    if (cur->left) q.push(cur->left);
                    if (cur->right) q.push(cur->right);
                } else {
                if (left_to_right) {
                    result.push_back(level);
                } else {
                reverse(level.begin(), level.end());
                result.push_back(level);
            }
            level.clear();
            left_to_right = !left_to_right;

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

    return result;
}
};
```

相关题目

- Binary Tree Level Order Traversal，见 §??
- Binary Tree Level Order Traversal II，见 §??

6.6.7 Recover Binary Search Tree

描述

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note: A solution using $O(n)$ space is pretty straight forward. Could you devise a constant space solution?

分析

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

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

代码

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

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

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

                if (node->right == nullptr) {
                    node->right = cur;
                    //prev = cur; 不能有这句! 因为 cur 还没有被访问
                    cur = cur->left;
                } else {
                detect(broken, prev, cur);
                node->right = nullptr;
                prev = cur;
                cur = cur->right;
            }
        }
    }

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

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

相关题目

- Binary Tree Inorder Traversal，见 §??

6.6.8 Same Tree

描述

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

分析

无

递归版

递归版

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

迭代版

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

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

            if (!p && !q) continue;
            if (!p || !q) return false;
            if (p->val != q->val) return false;

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

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

相关题目

- Symmetric Tree，见 §??

6.6.9 Symmetric Tree

描述

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

分析

无

递归版

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

迭代版

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

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

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

            if (!p && !q) continue;
            if (!p || !q) return false;
            if (p->val != q->val) return false;

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

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

        return true;
    }
};
```

相关题目

- Same Tree，见 §??

6.8.10 Balanced Binary Tree

描述

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

分析

无

代码

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

    /**
    * Returns the height of `root` if `root` is a balanced tree,
    * otherwise, returns `-1`.
    */
```

```
    int balancedHeight (TreeNode* root) {
        if (root == nullptr) return 0;  // 终止条件

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

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

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

相关题目

- 无

6.6.11 Flatten Binary Tree to Linked List

描述

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

For example, Given

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

The flattened tree should look like:

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

分析

无

递归版 1

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

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

        if (nullptr == root->left) return;

        // 三方合并，将左子树所形成的链表插入到 root 和 root->right 之间
        TreeNode *p = root->left;
```

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

递归版 2

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

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

迭代版

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

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

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

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

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

相关题目

- 无

6.6.12 Populating Next Right Pointers in Each Node II

描述

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

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

Note: You may only use constant extra space.

For example, Given the following binary tree,

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

After calling your function, the tree should look like:

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

分析

要处理一个节点，可能需要最右边的兄弟节点，首先想到用广搜。但广搜不是常数空间的，本题要求常数空间。

注意，这题的代码原封不动，也可以解决 Populating Next Right Pointers in Each Node I.

递归版

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

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

迭代版

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

```
                if (!next) next = root->left ? root->left : root->right;

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

相关题目

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

6.7 二叉树的构建

6.7.1 Construct Binary Tree from Preorder and Inorder Traversal

描述

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

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

分析

无

代码

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

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

        auto root = new TreeNode(*pre_first);

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

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

        return root;
```

```
        }
    };
```

相关题目

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

6.7.2 Construct Binary Tree from Inorder and Postorder Traversal

描述

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

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

分析

无

代码

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

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

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

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

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

        return root;
    }
};
```

相关题目

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

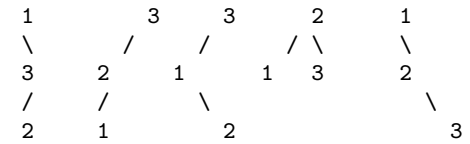
6.8 二叉查找树

6.8.1 Unique Binary Search Trees

描述

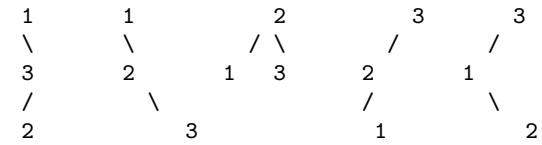
Given n, how many structurally unique BST's (binary search trees) that store values $1...n$?

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



分析

如果把上例的顺序改一下，就可以看出规律了。



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

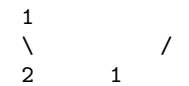
当数组为 $1, 2, 3, ..., n$ 时，基于以下原则的构建的 BST 树具有唯一性：**以 i 为根节点的树，其左子树由 [1, i-1] 构成，其右子树由 [i+1, n] 构成。**

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

如果数组为空，毫无疑问，只有一种 BST，即空树，$f(0) = 1$。

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

如果数组有两个元素 1,2，那么有如下两种可能



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

再看一看 3 个元素的数组，可以发现 BST 的取值方式如下：

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

所以，由此观察，可以得出 f 的递推公式为

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

至此，问题划归为一维动态规划。

代码

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

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

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

相关题目

- Unique Binary Search Trees II，见 §??

6.8.2 Unique Binary Search Trees II

描述

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

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

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

分析

见前面一题。

代码

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

```
                    TreeNode *node = new TreeNode(k);
                    node->left = i;
                    node->right = j;
                    subTree.push_back(node);
                }
            }
        }
        return subTree;
    }
};
```

相关题目

- Unique Binary Search Trees，见 §??

6.8.3 Validate Binary Search Tree

描述

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

分析

代码

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

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

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

相关题目

- Validate Binary Search Tree，见 §??

6.8.4 Convert Sorted Array to Binary Search Tree

描述

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

分析

二分法。

代码

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

    template<typename RandomAccessIterator>
    TreeNode* sortedArrayToBST (RandomAccessIterator first,
    RandomAccessIterator last) {
        const auto length = distance(first, last);

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

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

        return root;
    }
};
```

相关题目

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

6.8.5 Convert Sorted List to Binary Search Tree

描述

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

分析

这题与上一题类似，但是单链表不能随机访问，而自顶向下的二分法必须需要 RandomAccessIterator，因此前面的方法不适用本题。

存在一种自底向上 (bottom-up) 的方法，见 http://leetcode.com/2010/11/convert-sorted-list-to-balanced-binary.html

分治法，自顶向下

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

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

    TreeNode* sortedListToBST (ListNode* head, int len) {
        if (len == 0) return nullptr;
        if (len == 1) return new TreeNode (head->val);

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

```
            root->right = sortedListToBST (nth_node (head, len / 2 + 2),
            (len - 1) / 2);

            return root;
        }

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

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

            return n;
        }

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

            return node;
        }
    };
```

自底向上

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

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

相关题目

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

6.9 二叉树的递归

二叉树是一个递归的数据结构，因此是一个用来考察递归思维能力的绝佳数据结构。

递归一定是深搜（见 §??节“深搜与递归的区别”），由于在二叉树上，递归的味道更浓些，因此本节用“二叉树的递归”作为标题，而不是“二叉树的深搜”，尽管本节所有的算法都属于深搜。

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

6.9.1 Minimum Depth of Binary Tree

描述

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

分析

无

递归版

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

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

迭代版

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

        int result = INT_MAX;

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

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

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

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

            if (node->right && result > depth) // 深度控制，剪枝
```

```
                s.push(make_pair(node->right, depth + 1));
            }

            return result;
        }
    };
```

相关题目

- Maximum Depth of Binary Tree，见 §??

6.9.2 Maximum Depth of Binary Tree

描述

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

分析

无

代码

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

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

相关题目

- Minimum Depth of Binary Tree，见 §??

6.9.3 Path Sum

描述

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

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

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

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

分析

题目只要求返回 `true` 或者 `false`，因此不需要记录路径。

由于只需要求出一个结果，因此，当左、右任意一棵子树求到了满意结果，都可以及时 return。

由于题目没有说节点的数据一定是正整数，必须要走到叶子节点才能判断，因此中途没法剪枝，只能进行朴素深搜。

代码

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

        if (root->left == nullptr && root->right == nullptr) // leaf
        return sum == root->val;

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

相关题目

- Path Sum II，见 §??

6.9.4 Path Sum II

描述

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

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

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

return

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

分析

跟上一题相比，本题是求路径本身。且要求出所有结果，左子树求到了满意结果，不能 return，要接着求右子树。

代码

```
// LeetCode, Path Sum II
// 时间复杂度 O(n)，空间复杂度 O(logn)
class Solution {
    public:
    vector<vector<int> > pathSum(TreeNode *root, int sum) {
        vector<vector<int> > result;
        vector<int> cur; // 中间结果
        pathSum(root, sum, cur, result);
```

```
        return result;
    }
    private:
    void pathSum(TreeNode *root, int gap, vector<int> &cur,
    vector<vector<int> > &result) {
        if (root == nullptr) return;

        cur.push_back(root->val);

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

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

相关题目

- Path Sum，见 §??

6.9.5 Binary Tree Maximum Path Sum

描述

Given a binary tree, find the maximum path sum.

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

```
1
/ \
2   3
```

Return 6.

分析

这题很难，路径可以从任意节点开始，到任意节点结束。

可以利用“最大连续子序列和”问题的思路，见第 §??节。如果说 Array 只有一个方向的话，那么 Binary Tree 其实只是左、右两个方向而已，我们需要比较两个方向上的值。

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

代码

```
// LeetCode, Binary Tree Maximum Path Sum
// 时间复杂度 O(n)，空间复杂度 O(logn)
class Solution {
    public:
    int maxPathSum(TreeNode *root) {
        max_sum = INT_MIN;
        dfs(root);
        return max_sum;
    }
    private:
    int max_sum;
    int dfs(const TreeNode *root) {
        if (root == nullptr) return 0;
```

```
            int l = dfs(root->left);
            int r = dfs(root->right);
            int sum = root->val;
            if (l > 0) sum += l;
            if (r > 0) sum += r;
            max_sum = max(max_sum, sum);
            return max(r, l) > 0 ? max(r, l) + root->val : root->val;
        }
    };
```

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

相关题目

- Maximum Subarray，见 §??

6.9.6 Populating Next Right Pointers in Each Node

描述

Given a binary tree

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

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Note:

- You may only use constant extra space.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example, Given the following perfect binary tree,

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

After calling your function, the tree should look like:

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

分析

无

代码

```
// LeetCode, Populating Next Right Pointers in Each Node
// 时间复杂度 O(n)，空间复杂度 O(logn)
class Solution {
    public:
    void connect(TreeLinkNode *root) {
```

```
        connect(root, NULL);
    }
    private:
    void connect(TreeLinkNode *root, TreeLinkNode *sibling) {
        if (root == nullptr)
        return;
        else
        root->next = sibling;

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

相关题目

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

6.9.7 Sum Root to Leaf Numbers

描述

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123`.

Find the total sum of all root-to-leaf numbers.

For example,

```
1
/ \
2   3
```

The root-to-leaf path `1->2` represents the number `12`. The root-to-leaf path `1->3` represents the number `13`.

Return the sum = `12 + 13 = 25`.

分析

无

代码

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

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

相关题目

- 无

6.10 堆

6.10.1 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

6.10.2 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

6.11 并查集

6.11.1 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 6-5、图 6-6和图 6-7所示。

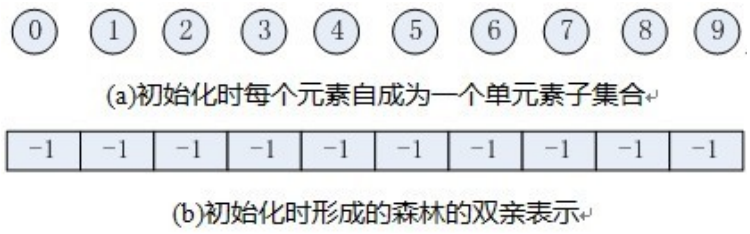


图 6-5 并查集的初始化

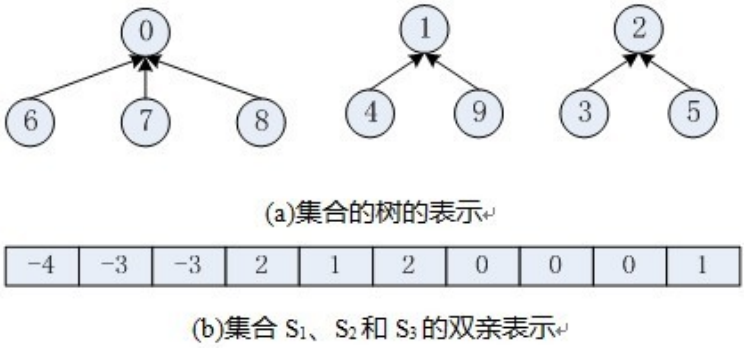


图 6-6 用树表示并查集

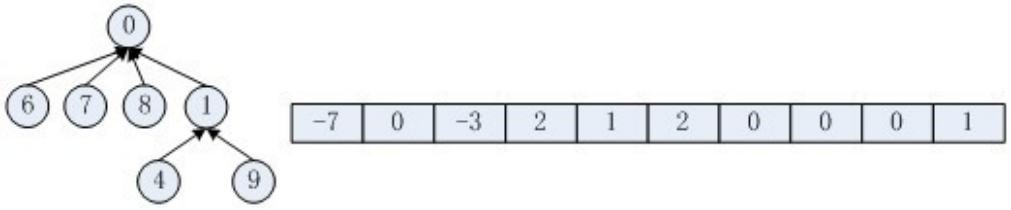


图 6-7 两个集合的并

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

6.11.2 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

6.11.3 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

6.11.4 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

6.12 线段树

6.12.1 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。

6.12.2 Balanced Lineup

描述

给定 $N(1 \le N \le 50,000)$ 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 $b(1 \le a \le b \le N)$，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

6.12.3 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

6.12.4 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

6.12.5 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

6.13 Trie 树

支持插入一个字符串，查询一个字符串是否存在。

6.13.1 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

6.13.2 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

6.13.3 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

```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

6.14 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*-
```

6.15 树状数组

对于一个数组 A[1..n], 在 O(logn) 的时间内完成以下操作：

（1）给 A[i] 加一个数

（2）求 A[1]+A[2]+...+A[i] 的和

6.16 左偏树

实现一个最小优先队列，使得插入、删除、合并等操作均在 O(logn) 的时间复杂度。

树状数组是一个优美小巧的数据结构，在很多时候可以代替线段树。一句话概括就是，凡是树状数组可以解决的问题，线段树都可以解决，反过来线段树可以解决的问题，树状数组不一定能解决。

6.17 Treap

动态维护一个有序表，支持在 O(logn) 的时间内完成插入一个元素，删除一个元素和查找第 K 大元素的任务。

6.18 伸展树 Splay

实现能够在 O(logn) 时间复杂度内实现各类二叉树操作的数据结构。

6.19 ST 表

给定一个数组 A[n], 动态查询数组元素 A[l],A[l+1],...,A[r] 的最小值。

6.20 动态树

6.21 可并堆

第 7 章
查找

7.1 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

7.2 哈希表

7.2.1 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

7.2.2 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

7.3 Search for a Range

描述

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of $O(\log n)$.

If the target is not found in the array, return `[-1, -1]`.

For example, Given `[5, 7, 7, 8, 8, 10]` and target value 8, return `[3, 4]`.

分析

已经排好了序，用二分查找。

使用 STL

```
// LeetCode, Search for a Range
// 偷懒的做法，使用 STL
// 时间复杂度 O(logn)，空间复杂度 O(1)
class Solution {
    public:
    vector<int> searchRange(int A[], int n, int target) {
        const int l = distance(A, lower_bound(A, A + n, target));
        const int u = distance(A, prev(upper_bound(A, A + n, target)));
```

```
            if (A[l] != target) // not found
            return vector<int> { -1, -1 };
            else
            return vector<int> { l, u };
        }
    };
```

重新实现 lower_bound 和 upper_bound

```
    // LeetCode, Search for a Range
    // 重新实现 lower_bound 和 upper_bound
    // 时间复杂度 O(logn), 空间复杂度 O(1)
    class Solution {
        public:
        vector<int> searchRange (int A[], int n, int target) {
            auto lower = lower_bound(A, A + n, target);
            auto uppper = upper_bound(lower, A + n, target);

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

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

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

            return first;
        }

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

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

            return first;
        }
    };
```

相关题目

- Search Insert Position, 见 §??

7.4 Search Insert Position

描述

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.

```
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0
```

分析

即 `std::lower_bound()`。

代码

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

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

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

        return first;
    }
};
```

相关题目

- Search for a Range, 见 §??

7.5 Search a 2D Matrix

描述

Write an efficient algorithm that searches for a value in an $m \times n$ matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example, Consider the following matrix:

```
[
[1,   3,  5,  7],
[10, 11, 16, 20],
[23, 30, 34, 50]
]
```

Given `target = 3`, return true.

分析

二分查找。

代码

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

        int first = 0;
        int last = m * n;

        while (first < last) {
            int mid = first + (last - first) / 2;
            int value = matrix[mid / n][mid % n];

            if (value == target)
            return true;
            else if (value < target)
            first = mid + 1;
            else
            last = mid;
        }

        return false;
    }
};
```

相关题目

- 无

第 8 章
排序

8.1 插入排序

8.1.1 直接插入排序

直接插入排序 (Straight Insertion Sort) 的基本思想是：把数组 a[n] 中待排序的 n 个元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素 a[0]，无序表中包含有 n-1 个元素 a[1] a[n-1]，排序过程中每次从无序表中取出第一个元素，把它插入到有序表中的适当位置，使之成为新的有序表，这样经过 n-1 次插入后，无序表就变为空表，有序表中就包含了全部 n 个元素，至此排序完毕。在有序表中寻找插入位置是采用从后向前的顺序查找的方法。

直接插入排序的 C 语言实现如下。

straight_insertion_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;

/**
  * @brief 直接插入排序，时间复杂度 O(n^2).
  *
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  */
void straight_insertion_sort(elem_t a[], const int start, const int end) {
    elem_t tmp;
    int i, j;

    for (i = start + 1; i < end; i++) {
        tmp = a[i];
        for (j = i - 1; tmp < a[j] && j >= start; j--) {
            a[j + 1] = a[j];
        }
        a[j + 1] = tmp;
    }
}
```

straight_insertion_sort.c

8.1.2 折半插入排序

在查找插入位置时，若改为折半查找，就是**折半插入排序** (Binary Insertion Sort)。

折半插入排序的 C 语言实现如下。

binary_insertion_sort.c

```
/**
  * @brief 折半插入排序，时间复杂度 O(nlog2n).
  *
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置
  * @return 无
  */
void binary_insertion_sort(elem_t a[], const int start, const int end) {
```

```
    elem_t tmp;
    int i, j, left, right, mid;

    for (i = start + 1; i < end; i++) {
        tmp = a[i];
        left = start;
        right = i - 1;
        while (left <= right) {
            mid = (left + right) / 2;
            if (tmp < a[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        for (j = i - 1;j >= left; j--) {
            a[j + 1] = a[j];
        }
        a[left] = tmp;
    }
}
```

binary_insertion_sort.c

8.1.3 希尔 (Shell) 插入排序

从对直接插入排序的分析得知，其算法时间复杂度为 $O(n^2)$，但是，若待排序记录序列为“正序”时，其时间复杂度可提高至 $O(n)$。由此可设想，若待排序记录序列按关键字“基本有序”，即序列中具有下列特性

$$R_i.key < \max\{R_j.key\},\ j < i$$

的记录较少时，直接插入排序的效率就可大大提高，从另一方面来看，由于直接插入排序算法简单，则在 n 值很小时效率也比较高。希尔排序正是从这两点分析出发对直接插入排序进行改进得到的一种插入排序方法。

希尔排序 (Shell Sort) 的基本思想是：设待排序元素序列有 n 个元素，首先取一个整数 $gap = \lfloor \frac{n}{3} \rfloor + 1$ 作为间隔，将全部元素分为 gap 个子序列，所有距离为 gap 的元素放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔 gap，取 $gap = \lfloor \frac{gap}{3} \rfloor + 1$，重复上述的子序列划分和排序工作，直到最后取 $gap = 1$，将所有元素放在同一个序列中排序为止。

图 8-1展示了希尔排序的过程。

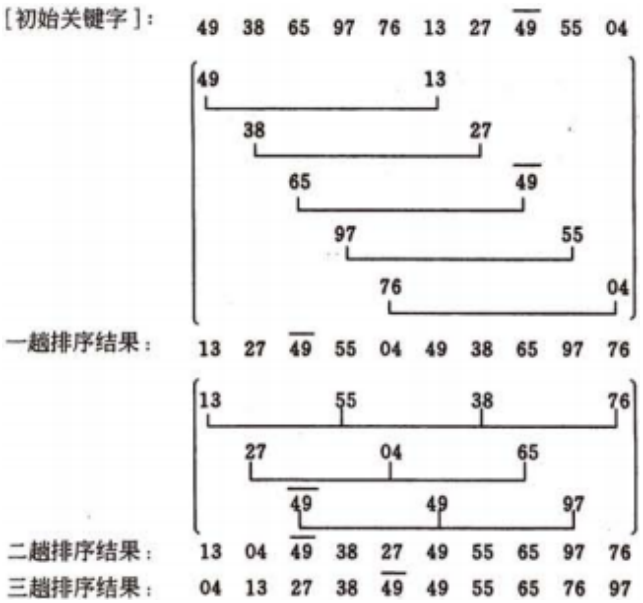


图 8-1 希尔排序

希尔排序的 C 语言实现如下。

shell_sort.c

```
/*
  * @brief 一趟希尔插入排序.
  *
  * 和一趟直接插入排序相比，仅有一点不同，就是前后元素的间距是 gap 而不是 1
  *
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @param[in] gap 间隔
  * @return 无
  */
static void shell_insert(elem_t a[], const int start, const int end, const int gap) {
    elem_t tmp;
    int i, j;
    for (i = start + gap; i < end; i++) {
        tmp = a[i];
        for (j = i - gap; tmp < a[j] && j >= start; j -= gap) {
            a[j + gap] = a[j];
        }
        a[j + gap] = tmp;
    }
}

 /*
  * @brief 希尔排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  */
void shell_sort(elem_t a[], const int start, const int end) {
    int gap = end - start;
    while (gap > 1) {
        gap = gap / 3 + 1;
        shell_insert(a, start, end, gap);
    }
}
```

shell_sort.c

8.2 交换排序

8.2.1 冒泡排序

冒泡排序 (Bubble Sort) 的基本方法是：设待排序元素序列的元素个数为 n，从后向前两两比较相邻元素的值，如果发生逆序（即前一个比后一个大），则交换它们，直到序列比较完。我们称它为一趟冒泡，结果是最小的元素交换到待排序序列的第一个位置，其他元素也都向排序的最终位置移动。下一趟冒泡时前一趟确定的最小元素不参加比较，待排序序列减少一个元素，一趟冒泡的结果又把序列中最小的元素交换到待排序序列的第一个位置。这样最多做 n-1 趟冒泡就能把所有元素排好序。

冒泡排序的 C 语言实现如下。

bubble_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;

/**
  * @brief 冒泡排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
```

```
  * @note 无
  * @remarks 无
  */
void bubble_sort(elem_t a[], const int start, const int end) {
    int exchange; /* 是否发生交换 */
    elem_t tmp;
    int i, j;

    for (i = start; i < end - 1; i++) {
        exchange = 0;
        for (j = end - 1; j > i; j--) { /* 发生逆序，交换 */
            if (a[j - 1] > a[j]) {
                tmp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = tmp;
                exchange = 1;
            }
        }
        if (exchange == 0) return; /* 本趟无逆序，停止处理 */
    }
}
```

bubble_sort.c

8.2.2 快速排序

快速排序 (Quick sort) 的基本思想是任取待排序元素序列中的某个元素（例如取第一个元素）作为基准，按照该元素的关键字大小，将整个元素序列划分为左右两个子序列：左侧子序列中所有元素的关键字都小于基准元素的关键字，右侧子序列中所有元素的关键字都大于或等于基准元素的关键字，基准元素则排在这两个子序列中间（这也是该元素最终应该安放的位置）。然后分别对这两个子序列重复施行上述算法，直到所有的元素都排在相应位置为止。

一趟快排的过程如图 8-2 (a) 所示。整个快速排序的过程可递归，如图 8-2 (b) 所示。

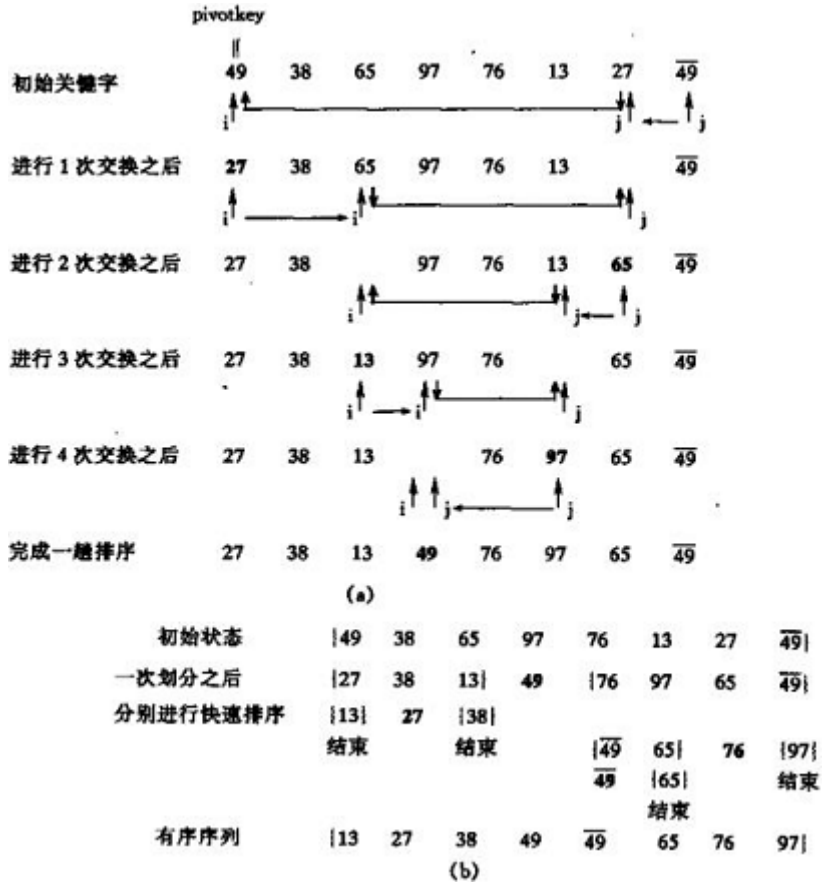


图 8-2 快速排序示例

更多详细解释请参考本项目的 wiki，https://github.com/soulmachine/acm-cheatsheet/wiki/快速排序

快速排序的 C 语言实现如下。

quick_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;
 /*
  * @brief 一趟划分.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 基准元素的新位置
  */
int partition(elem_t a[], const int start, const int end) {
    int i = start;
    int j = end - 1;
    const elem_t pivot = a[i];

    while(i < j) {
        while(i < j && a[j] >= pivot) j--;
        a[i] = a[j];
        while(i < j && a[i] <= pivot) i++;
        a[j] = a[i];
    }
    a[i] = pivot;
    return i;
}

/**
  * @brief 快速排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置
  * @return 无
  */
void quick_sort(elem_t a[], const int start, const int end) {
    if(start < end - 1) { /* 至少两个元素 */
        const int pivot_pos = partition(a, start, end);
        quick_sort(a, start, pivot_pos);
        quick_sort(a, pivot_pos + 1, end);
    }
}
```

quick_sort.c

8.3 选择排序

选择排序 (Selection sort) 的基本思想是：每一趟在后面 n-i(i=1, 2, ..., n-2) 个元素中选取最小的元素作为有序序列的第 i 个元素。

8.3.1 简单选择排序

简单选择排序 (simple selection sort) 也叫直接选择排序 (straight selection sort)，其基本步骤是：

- 在一组元素 a[i] a[n-1] 中选择最小的元素；
- 若它不是这组元素中的第一个元素，则将它与这组元素的第一个元素对调；
- 在剩下的 a[i+1] a[n-1] 中重复执行以上两步，直到剩余元素只有一个为止。

简单选择排序的 C 语言实现如下。

simple_selection_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;

/**
  * @brief 简单选择排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  */
void simple_selection_sort(elem_t a[], int start, int end) {
    elem_t tmp;
    int i, j, k;

    for (i = start; i < end; i++) {
        k = i;
        /* 在 a[i] 到 a[end-1] 中寻找最小元素 */
        for (j = i + 1; j < end; j++)
            if(a[j] < a[k]) k = j;
        /* 交换 */
        if (k != i) {
            tmp = a[i];
            a[i] = a[k];
            a[k]= tmp;
        }
    }
}
```

simple_selection_sort.c

8.3.2 堆排序

堆排序的 C 语言实现如下。

heap_sort.c

```
#include "heap.c"
/**
  * @brief 堆排序.
  * @param[inout] a 待排序元素序列
  * @param[in] n 元素个数
  * @param[in] cmp cmp 比较函数，小于返回-1，等于，大于
  * @return 无
  */
void heap_sort(heap_elem_t *a, const int n,
               int (*cmp)(const heap_elem_t*, const heap_elem_t*)) {
    int i;
    heap_t *h;
    heap_elem_t tmp;

    h = heap_create(n, cmp);
    h->elems = a;

    i = (h->size - 2)/2;   /* 找最初调整位置：最后分支结点 */
    while (i >= 0) {  /* 自底向上逐步扩大形成堆 */
        heap_sift_down(h, i);
        i--;
    }

    for (i = h->size - 1; i > 0; i--) {
        tmp = h->elems[i];
        h->elems[i] = h->elems[0];
        h->elems[0] = tmp;
        h->size = i; /* 相当于 h.size -- */
        heap_sift_down(h, 0);
```

```
    }
    heap_destroy(h);
}
```
heap_sort.c

8.4 归并排序

所谓“归并”，就是将两个或两个以上的有序序列合并成一个有序序列。我们先从最简单的二路**归并排序** (Merge sort) 入手。

图 8-3是一个二路归并排序的例子。

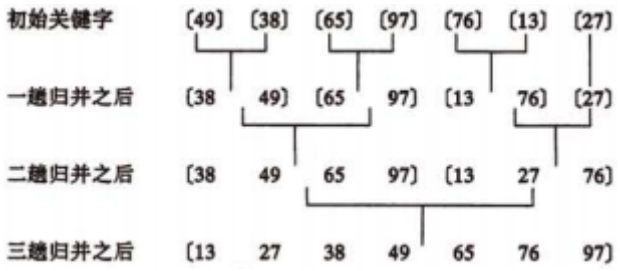


图 8-3　二路归并排序示例

二路归并排序的 C 语言实现如下。

merge_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;

 /** 数组元素的类型 */
typedef int elem_t;

 /*
  * @brief 将两个有序表合并成一个新的有序表
  * @param[inout] a 待排序元素序列，包含两个有序表
  * @param[in] tmp 与 a 等长的辅助数组
  * @param[in] start a[start]~a[mid-1] 为第一个有序表
  * @param[in] mid 分界点
  * @param[in] end a[mid]~a[end-1] 为第二个有序表
  * @return 无
  */
static void merge(elem_t a[], elem_t tmp[], const int start,
        const int mid, const int end) {
    int i, j, k;
    for (i = 0; i < end; i++) tmp[i] = a[i];

    /* i, j 是检测指针，k 是存放指针 */
    for (i = start, j = mid, k = start; i < mid && j < end; k++) {
        if (tmp[i] < tmp[j]) {
            a[k] = tmp[i++];
        } else {
            a[k] = tmp[j++];
        }
    }
    /* 若第一个表未检测完，复制 */
    while (i < mid) a[k++] = tmp[i++];
    /* 若第二个表未检测完，复制 */
    while (j < end) a[k++] = tmp[j++];
}

/**
  * @brief 归并排序.
  * @param[inout] a 待排序元素序列
```

```
  * @param[in] tmp 与 a 等长的辅助数组
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  * @note 无
  * @remarks 无
  */
void merge_sort(elem_t a[], elem_t tmp[], const int start, const int end) {
    if (start < end - 1) {
        const int mid = (start + end) / 2;
        merge_sort(a, tmp, start, mid);
        merge_sort(a, tmp, mid, end);
        merge(a, tmp, start, mid, end);
    }
}
```

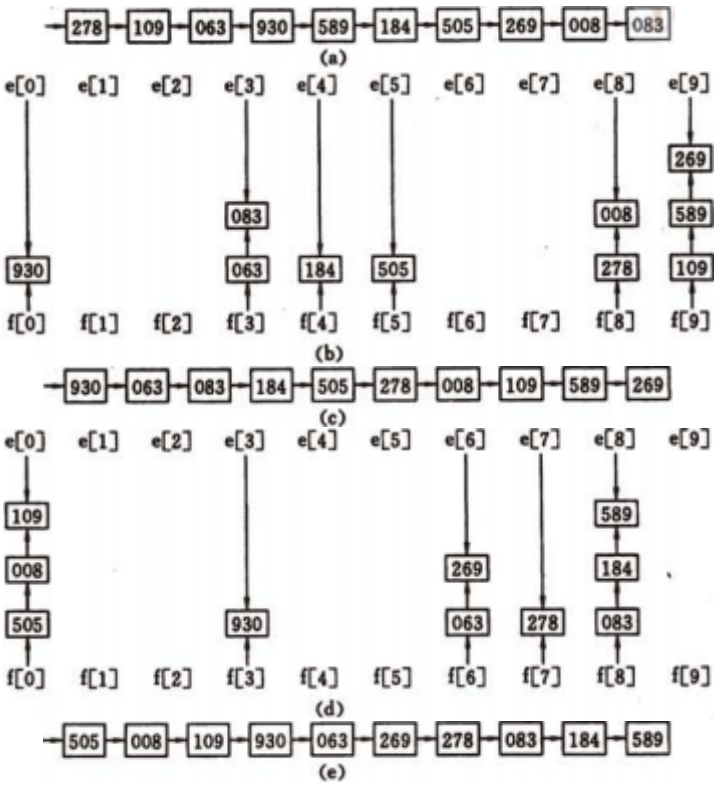
merge_sort.c

8.5 基数排序

利用多关键字实现对单关键字排序的算法就称为**基数排序** (Radix sort)。

有两种顺序，最高位优先 MSD(Most Significant Digit first) 和最低位优先 LSD(Least Significant Digit first)。

下面介绍“LSD 链式基数排序”。首先以静态链表存储 n 个待排元素，并令表头指针指向第一个元素，即 A[1] 到 A[n] 存放元素，A[0] 为表头结点，这样元素在重排时不必移动元素，只需要修改各个元素的 link 指针即可，如图 8-4(a) 所示。每个位设置一个桶（跟散列桶一样），桶采用静态链表结构，同时设置两个数组 f[RADIX] 和 r[RADIX]，记录每个桶的头指针和尾指针。排序过程就是 d（关键字位数）趟“分配”、“收集”的过程，如图 8-4所示。



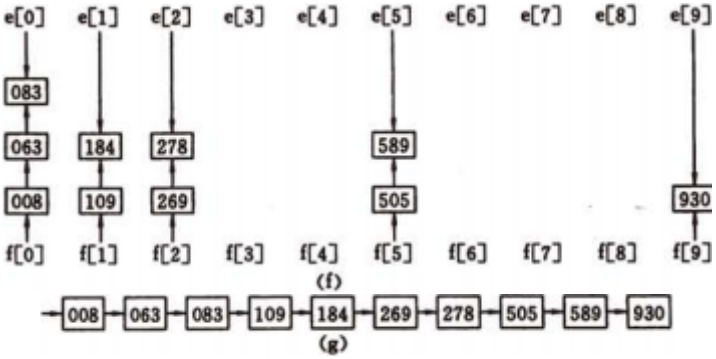


图 8-4 LSD 链式基数排序示例

LSD 链式基数排序的 C 语言实现如下。

radix_sort.c

```
/** @file radix_sort.c
  * @brief LSD 链式基数排序.
  * @author soulmachine@gmail.com
  * @date 2013-05-18
  */
#include <stdio.h>  /* for printf() */

/* 关键字基数, 此时是十进制 */
#define R 10  /*Radix*/

/**
  *@struct
  *@brief 静态链表结点.
  */
typedef struct static_list_node_t {
    int key; /** 关键字 */
    int link; /** 下一个节点 */
}static_list_node_t;

 /*
  * @brief 打印静态链表.
  * @param[in] a 静态链表数组
  * @return 无
  */
static void static_list_print(const static_list_node_t a[]) {
    int i = a[0].link;
    while (i != 0) {
        printf("%d ", a[i].key);
        i = a[i].link;
    }
}

 /*
  * @brief 获取十进制整数的某一位数字.
  * @param[in] n 整数
  * @param[in] i 第 i 位
  * @return 整数 n 第 i 位的数字
  */
static int get_digit(int n, const int i) {
    int j;
    for(j = 1; j < i; j++) {
        n /= 10;
    }

    return n % 10;
}
```

```
/**
  * @brief LSD 链式基数排序.
  * @param[in] a 静态链表, a[0] 是头指针
  * @param[in] n 待排序元素的个数
  * @param[in] d 最大整数的位数
  * @return 无
  * @note 无
  * @remarks 无
  */
void radix_sort(static_list_node_t a[], const int n, const int d) {
    int i, j, k, current, last;
    int rear[R], front[R];

    for(i = 0; i < n; i++) a[i].link = i + 1;
    a[n].link = 0;
    for(i = 0; i < d; i++) {
        /* 分配 */
        for(j = 0; j < R; j++) front[j] = 0;
        for(current = a[0].link; current != 0;
            current = a[current].link) {
            k = get_digit(a[current].key, i + 1);
            if(front[k] == 0) {
                front[k] = current;
                rear[k] = current;
            } else {
                a[rear[k]].link = current;
                rear[k] = current;
            }
        }

        /* 收集 */
        j = 0;
        while(front[j] == 0) j++;
        a[0].link = current = front[j];
        last = rear[j];
        for(j = j + 1; j < R; j++) {
            if(front[j] != 0) {
                a[last].link = front[j];
                last = rear[j];
            }
        }
        a[last].link = 0;
    }
}

void radix_sort_test(void) {
    static_list_node_t a[] = {{0,0}/* 头指针 */, {278,0}, {109,0},
    {63,0}, {930,0}, {589,0}, {184,0}, {505,0}, {269,0},
    {8,0}, {83,0}};
    radix_sort(a, 10, 3);
    static_list_print(a);
}
```

radix_sort.c

8.6 总结和比较

表 8-1 各种排序算法的总结和比较

排序方法	平均时间	最坏情况	辅助存储	是否稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
希尔排序	N/A	N/A	$O(1)$	否
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
二路归并	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d\times(n+R))$	$O(d\times(n+R))$	$O(R)$	是

假设在数组中有两个元素 $A_i, A_j, i<j$，即 A_i 在 A_j 之前，且 $A_i=A_j$，如果在排序之后，A_i 仍然在 A_j 的前面，则称这个排序算法是**稳定**的，否则称这个排序算法是**不稳定**的。

排序方法根据在排序过程中数据是否完全在内存，分为两大类：**内部排序**和**外部排序**。内部排序是指在排序期间数据全部存放在内存；外部排序是指在排序期间所有数据不能同时存放在内存，在排序过程中需要不断在内、外存之间交换。一般说到排序，默认是指内部排序。

8.7 Merge Sorted Array

描述

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

分析

无

代码

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

相关题目

- Merge Two Sorted Lists，见 §??
- Merge k Sorted Lists，见 §??

8.8 Merge Two Sorted Lists

描述

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

分析

无

代码

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

相关题目

- Merge Sorted Array §??
- Merge k Sorted Lists，见 §??

8.9 Merge k Sorted Lists

描述

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

分析

可以复用 Merge Two Sorted Lists（见 §??）的函数

代码

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

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

```
        }

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

相关题目

- Merge Sorted Array §??
- Merge Two Sorted Lists，见 §??

8.10 Insertion Sort List

描述

Sort a linked list using insertion sort.

分析

无

代码

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

            for (ListNode *cur = head; cur != nullptr;) {
                auto pos = findInsertPos(&dummy, cur->val);
                ListNode *tmp = cur->next;
                cur->next = pos->next;
                pos->next = cur;
                cur = tmp;
            }
            return dummy.next;
        }

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

```
        }
    };
```

相关题目

- Sort List, 见 §??

8.11 Sort List

描述

Sort a linked list in $O(nlogn)$ time using constant space complexity.

分析

常数空间且 $O(nlogn)$，单链表适合用归并排序，双向链表适合用快速排序。本题可以复用"Merge Two Sorted Lists"的代码。

代码

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

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

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

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

相关题目

- Insertion Sort List，见 §??

8.12 First Missing Positive

描述

Given an unsorted integer array, find the first missing positive integer.

For example, Given `[1,2,0]` return 3, and `[3,4,-1,1]` return 2.

Your algorithm should run in $O(n)$ time and uses constant space.

分析

本质上是桶排序 (bucket sort)，每当 `A[i]!= i+1` 的时候，将 A[i] 与 A[A[i]-1] 交换，直到无法交换为止，终止条件是 `A[i]== A[A[i]-1]`。

代码

```
// LeetCode, First Missing Positive
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int firstMissingPositive(int A[], int n) {
        bucket_sort(A, n);

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

相关题目

- Sort Colors, 见 §??

8.13 Sort Colors

描述

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not suppose to use the library's sort function for this problem.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

分析

由于 0, 1, 2 非常紧凑，首先想到计数排序 (counting sort)，但需要扫描两遍，不符合题目要求。

由于只有三种颜色，可以设置两个 index，一个是 red 的 index，一个是 blue 的 index，两边往中间走。时间复杂度 $O(n)$，空间复杂度 $O(1)$。

第 3 种思路，利用快速排序里 partition 的思想，第一次将数组按 0 分割，第二次按 1 分割，排序完毕，可以推广到 n 种颜色，每种颜色有重复元素的情况。

代码 1

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

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

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

    }
};
```

代码 2

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

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

代码 3

```
// LeetCode, Sort Colors
// use partition()
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    void sortColors(int A[], int n) {
```

```
            partition(partition(A, A + n, bind1st(equal_to<int>(), 0)), A + n,
            bind1st(equal_to<int>(), 1));
        }
    };
```

代码 4

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

            for (; first != last; ++first)
            if (pred(*first))
            swap(*first, *pos++);

            return pos;
        }
    };
```

相关题目

- First Missing Positive, 见 §??

第 9 章
暴力枚举法

暴力枚举法 (brute force enumeration) 又称为暴力搜索法 (Brute-force search)，详细定义见 Wikipedia, http://en.wikipedia.org/wiki/Brute_force_search

9.1 枚举排列

枚举排列，即输出某个集合的所有排列。例如，集合 1,2,3 的所有排列是 (1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)。

9.1.1 生成 1 到 n 的全排列

描述

给定一个正整数 n，输出 1 到 n 的所有排列。

分析

我们尝试用递归的思路：先输出以 1 开头的排列（这一步是递归调用），然后输出以 2 开头的排列（又是递归调用），...，最后才是以 n 开头的排列。

以 1 开头的排列，第一位是 1，后面是 2 到 9 的排列，这是一个子问题，可以接着递归。

伪代码如下：

```
void print_permutation(序列 P, 集合 S) {
    if (S 为空) 输出序列 P
    else {
        for(按照从小到大的顺序依次考虑 S 中的每个元素 e) {
            print_permutation(在 A 的末尾添加 e 后得到的新序列, S-{e});
        }
    }
}
// 调用
print_permutation({}, S);
```

代码

下面考虑用 C 语言实现。不难想到用数组表示 P 和 S。由于 P 和 S 是互补的，它们二者知道其中给一个，另一个就完全确定了，因此不用保存 P。

C 语言实现如下：

print_permutation_n.c

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

/*
 * @brief 输出 1 到 n 的全排列
 * @param[in] n n
 * @param[in] cur 当前进行到哪个位置
 * @param[out] 存放一个排列
 * @return 无
 */
static void print_permutation_r(int n, int cur, int P[]) {
```

```
    int i, j;
    if (cur == n) { // 收敛条件
        for (i = 0; i < n; i++)
            printf("%d", P[i]);
        printf("\n");
    }

    // 扩展状态，尝试在 A[cur] 中填各种整数 i，按从小到大的顺序
    for (i = 1; i <= n; i++) {
        int used = 0;
        for (j = 0; j < cur; j++)
            if (P[j] == i)
                used = 1; // 如果 i 已经在 A[0]~A[cur-1] 出现过，则不能再选
        if (!used) {
            P[cur] = i;
            print_permutation_r(n, cur + 1, P); // 递归调用
            // 不需要恢复 P[cur]，返回上层时时会被覆盖
        }
    }
}

/**
 * @brief 输出 1 到 n 的全排列
 * @param[in] n n
 * @return 无
 */
void print_permutation(int n) {
    int *P = (int*)malloc(n * sizeof(int));
    print_permutation_r(n, 0, P);
    free(P);
    return;
}

//test
int main() {
    print_permutation(3);
    return 0;
}
```

print_permutation_n.c

9.1.2 生成可重集的排列

描述

如果把问题改成，给定一个数组 S，按字典序输出所有排列。

分析

此时 S 不再仅限于整数，可以是字符等。

首先想到可以利用上一题的思路，先输出 1 到 n 的全排列，然后把打印语句改成 `printf("%c", S[P[i]-1])` 即可。

但是这个方法有点问题，当集合中有重复元素时，它不能正确处理。例如 S="AAA"，它会打印出 6 个 AAA。见下面的代码。

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

/*
 * @brief 输出 1 到 n 的全排列，把数字当做下标
 * @param[in] S 字符集合
 * @param[in] n n
 * @param[in] cur 当前进行到哪个位置
 * @param[out] 存放一个排列
 * @return 无
```

```
 */
static void print_permutation_r(char S[], int n, int cur, int P[]) {
    int i, j;
    if (cur == n) { // 递归边界
        for (i = 0; i < n; i++)
            printf("%c", S[P[i]-1]);
        printf("\n");
    } else {
        // 尝试在 A[cur] 中填各种整数 i，按从小到大的顺序
        for (i = 1; i <= n; i++) {
            int ok = 1;
            for (j = 0; j < cur; j++)
                if (P[j] == i)
                    ok = 0; // 如果 i 已经在 A[0]~A[cur-1] 出现过，则不能再选
            if (ok) {
                P[cur] = i;
                print_permutation_r(S, n, cur + 1, P); // 递归调用
            }
        }
    }
}

/**
 * @brief 输出字符集合的全排列
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation(char S[], int n) {
    int *P = (int*)malloc(n * sizeof(int));
    print_permutation_r(S, n, 0, P);
    free(P);
    return;
}

//test
int main() {
    char *S="ABC";
    char *S1="AAA";
    print_permutation(S,3);
    print_permutation(S1,3); // 不能正确处理可重集
    return 0;
}
```

再换一个思路，还是在上一节的代码上进行修改，把 `if(P[j]==i)` 和 `P[cur]=i` 分别改成 `if(P[j]==S[i])` 和 `P[cur]=S[i]`。这样，只要把 S 中的所有元素按从小到大的顺序排序，然后调用 `print_permutation_r(S, n, 0, P)` 即可。

这个方法看上去不错，可惜还是有个小问题。例如 `S="AAA"`，程序什么也不输出（正确答案应该是唯一的全排列 AAA），原因在于，我们禁止 S 数组中出现重复，而在 S 中本来就有重复元素，这个“禁令”是错误的。

解决方法是，统计 P[0] 到 P[cur-1] 中 S[i] 出现的次数 c1，以及 S 数组中 S[i] 的出现次数 c2，只要 c1<c2，就能继续选择 S[i]。

结果又如何呢？这次有输出了，可是输出了 27 个 AAA。遗漏是没有了，可是出现了重复。程序先把第 1 个 A 作为开头，递归调用结束后用第 2 个 A 作为开头，递归调用结束后用第 3 个 A 作为开头。每次输出 3 个排列，共 27 个。

枚举时应该**不重不漏**地取遍集合的所有元素。由于数组已经排序，所以只需要检查当前元素和前一个元素不相同，就可以做到不重不漏了。即只需要在 `for (i = 1; i <= n; i++)` 和其后的花括号之间加上 `if(i==0 && S[i] != S[i-1])`。见下面的代码。

代码

print_permutation.c

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

/*
 * @brief 输出 1 到 n 的全排列
 * @param[in] n n
 * @param[in] cur 当前进行到哪个位置
 * @param[out] 存放一个排列
 * @return 无
 */
static void print_permutation_r(int n, int cur, int P[]) {
    int i, j;
    if (cur == n) { // 收敛条件
        for (i = 0; i < n; i++)
            printf("%d", P[i]);
        printf("\n");
    }

    // 扩展状态，尝试在 A[cur] 中填各种整数 i，按从小到大的顺序
    for (i = 1; i <= n; i++) {
        int used = 0;
        for (j = 0; j < cur; j++) {
            if (P[j] == i) {
                used = 1; // 如果 i 已经在 A[0]~A[cur-1] 出现过，则不能再选
                break;
            }
        }
        if (!used) {
            P[cur] = i;
            print_permutation_r(n, cur + 1, P); // 递归调用
            // 不需要恢复 P[cur]，返回上层时时会被覆盖
        }
    }
}

/**
 * @brief 输出 1 到 n 的全排列
 * @param[in] n n
 * @return 无
 */
void print_permutation(int n) {
    int *P = (int*)malloc(n * sizeof(int));
    print_permutation_r(n, 0, P);
    free(P);
    return;
}

//test
int main() {
    print_permutation(3);
    return 0;
}
```

print_permutation.c

9.1.3 下一个排列

还可以利用 STL 中的 `next_permutation()`，或者自己实现，见第 §18.1节的 `next_permutation()`。

代码

print_permutation_next.c

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

```
#include <algorithm>

/**
 * @brief 输出字符集合的全排列，利用 next_permutation
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation(char S[], int n) {
    sort(&S[0], &S[n]);
    do {
        for(int i = 0; i < n; i++) printf("%c", S[i]);
        printf("\n");
    }while(next_permutation(&S[0], &S[n]));
    return;
}

/* 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 "next_permutation.c"

/**
 * @brief 输出字符集合的全排列，利用第 15 章的 next_permutation
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation1(char S[], int n) {
    sort(&S[0], &S[n]);
    int *N = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; ++i) N[i]=S[i]-'A';
    do {
        for(int i = 0; i < n; i++) printf("%c", S[N[i]]);
        printf("\n");
    }while(next_permutation(&N[0], &N[n]));
    return;
}

//test
int main() {
    char S[]="ABC";
    char S1[]="AAA";
    print_permutation(S,3);
    print_permutation(S1,3);
    printf("\n\n");
    print_permutation1(S,3);
    print_permutation1(S1,3);
    return 0;
}
```

print_permutation_next.c

9.2 子集生成

给定一个集合，输出它所有的子集。为了简单起见，本节讨论的集合中没有重复元素。

9.2.1 增量构造法

一次选出一个元素，放或者不放到集合中。

代码

subset.c

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

/**
 * @brief 增量构造法
 * @param[in] S 输入集合
 * @param[in] n 集合大小
 * @param[inout] P 某个子集
 * @param[in] cur p 的当前位置
 * @param[in] ed S 的当前位置，前面的元素已经选过了
 * @return 无
 */
void print_subset1(int *S, int n, int *P, int cur, int ed) {
    int i, j;
    for (i = ed; i < n; i++) {
        // 选择 S[i]
        P[cur] = S[i];
        for (j = 0; j <= cur; j++) printf("%d ", P[j]);
        printf("\n");
        // 不选择 S[i]
        print_subset1(S, n, P, cur + 1, i + 1);
    }
}
```

subset.c

9.2.2 位向量法

开一个位向量 B，B[i]=1 表示选择 S[i], B[i]=0 表示不选择。

代码

```c
/**
 * @brief 位向量法
 * @param[in] S 输入集合
 * @param[in] n 集合大小
 * @param[in] B 位向量
 * @param[in] cur B 的当前位置
 * @return 无
 */
void print_subset2(int *S, int n, char *B, int cur) {
    int i;
    if (cur == n) {
        for (i = 0; i < n; i++) if (B[i]) printf("%d ", S[i]);
        printf("\n");
        return;
    }
    B[cur] = 1;
    print_subset2(S, n, B, cur + 1);
    B[cur] = 0;
    print_subset2(S, n, B, cur + 1);
}
```

subset.c

9.2.3 二进制法

前提：集合的元素不超过 int 位数。用一个 int 整数表示位向量，第 i 位为 1，则表示选择 S[i]，为 0 则不选择。例如 S={A,B,C,D}，则 0110=6 表示子集 {B,C}。

这种方法最巧妙。因为它不仅能生成子集，还能方便的表示集合的并、交、差等集合运算。设两个集合的位向量分别为 B1 和 B2，则 B1|B2, B1&B2, B1^B2 分别对应集合的并、交、对称差。

代码

subset.c

```
/**
 * @brief 二进制法
 * @param[in] S 输入集合
 * @param[in] n 集合大小
 * @param[in] B 位向量
 * @param[in] cur B 的当前位置
 * @return 无
 */
void print_subset3(int *S, int n) {
    int i, j;
    for (i = 1; i < (1 << n); i++) {
        for (j = 0; j < n; j++)
            if (i & (1 << j)) printf("%d ", S[j]);
        printf("\n");
    }
}

int main() {
    int n, i;

    while(scanf("%d",&n) > 0) {
        int *S = (int*)malloc(n * sizeof(int));
        int *P = (int*)malloc(n * sizeof(int));
        char *B = (char*)malloc(n * sizeof(char));

        for(i = 0; i < n; i++) scanf("%d",&S[i]);

        print_subset1(S, n, P, 0, 0); putchar('\n');
        print_subset2(S, n, B, 0); putchar('\n');
        print_subset3(S, n);

        free(S);
        free(P);
        free(B);
    }
    return 0;
}
```

subset.c

9.3 Subsets

描述

Given a set of distinct integers, S, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example, If `S = [1,2,3]`, a solution is:

```
[
[3],
[1],
[2],
[1,2,3],
[1,3],
[2,3],
[1,2],
[]
]
```

9.3.1 递归

增量构造法

每个元素，都有两种选择，选或者不选。

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

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

位向量法

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

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

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

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

```
            // 选 S[step]
            selected[step] = true;
            subsets(S, selected, step + 1, result);
        }
    };
```

9.3.2 迭代

增量构造法

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

二进制法

本方法的前提是：集合的元素不超过 int 位数。用一个 int 整数表示位向量，第 i 位为 1，则表示选择 $S[i]$，为 0 则不选择。例如 `S={A,B,C,D}`，则 `0110=6` 表示子集 `{B,C}`。

这种方法最巧妙。因为它不仅能生成子集，还能方便的表示集合的并、交、差等集合运算。设两个集合的位向量分别为 B_1 和 B_2，则 $B_1 \cup B_2, B_1 \cap B_2, B_1 \triangle B_2$ 分别对应集合的并、交、对称差。

二进制法，也可以看做是位向量法，只不过更加优化。

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

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

相关题目

- Subsets II，见 §??

9.4 Subsets II

描述

Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

Elements in a subset must be in non-descending order. The solution set must not contain duplicate subsets. For example, If `S = [1,2,2]`, a solution is:

```
[
[2],
[1],
[1,2,2],
[2,2],
[1,2],
[]
]
```

分析

这题有重复元素，但本质上，跟上一题很类似，上一题中元素没有重复，相当于每个元素只能选 0 或 1 次，这里扩充到了每个元素可以选 0 到若干次而已。

9.4.1 递归

增量构造法

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

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

        dfs(S, S.begin(), path, result);
        return result;
    }

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

        for (auto i = start; i < S.end(); i++) {
            if (i != start && *i == *(i-1)) continue;
            path.push_back(*i);
            dfs(S, i + 1, path, result);
            path.pop_back();
        }
    }
};
```

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

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

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

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

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

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

位向量法

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

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

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

    private:
```

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

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

9.4.2 迭代

增量构造法

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

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

二进制法

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

        for (size_t i = 0; i < 1U << n; ++i) {
            for (size_t j = 0; j < n; ++j) {
```

```
                if (i & 1 << j)
                v.push_back(S[j]);
            }
            result.insert(v);
            v.clear();
        }
        vector<vector<int> > real_result;
        copy(result.begin(), result.end(), back_inserter(real_result));
        return real_result;
    }
};
```

相关题目

- Subsets，见 §??

9.5 Permutations

描述

Given a collection of numbers, return all possible permutations.

For example, `[1,2,3]` have the following permutations: `[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]`, and `[3,2,1]`.

9.5.1 next_permutation()

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

代码

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

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

9.5.2 重新实现 next_permutation()

见第 §2.1.13 节。

代码

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

```
        vector<vector<int>> permutations;

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

        return permutations;
    }
};
```

9.5.3 递归

本题是求路径本身，求所有解，函数参数需要标记当前走到了哪步，还需要中间结果的引用，最终结果的引用。
扩展节点，每次从左到右，选一个没有出现过的元素。
本题不需要判重，因为状态装换图是一颗有层次的树。收敛条件是当前走到了最后一个元素。

代码

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

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

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

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

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

相关题目

- Next Permutation, 见 §2.1.13
- Permutation Sequence, 见 §2.1.14
- Permutations II, 见 §??
- Combinations, 见 §??

9.6 Permutations II

描述

Given a collection of numbers that might contain duplicates, return all possible unique permutations.
For example, `[1,1,2]` have the following unique permutations: `[1,1,2]`, `[1,2,1]`, and `[2,1,1]`.

9.6.1 next_permutation()

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

9.6.2 重新实现 next_permutation()

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

9.6.3 递归

递归函数 `permute()` 的参数 `p`，是中间结果，它的长度又能标记当前走到了哪一步，用于判断收敛条件。
扩展节点，每次从小到大，选一个没有被用光的元素，直到所有元素被用光。
本题不需要判重，因为状态装换图是一颗有层次的树。

代码

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

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

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

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

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

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

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

```
            }

            // 扩展状态
            for (auto i = first; i != last; i++) {
                int count = 0; // 统计 *i 在 p 中出现过多少次
                for (auto j = p.begin(); j != p.end(); j++) {
                    if (i->first == *j) {
                        count ++;
                    }
                }
                if (count < i->second) {
                    p.push_back(i->first);
                    permute(first, last, p, result);
                    p.pop_back(); // 撤销动作，返回上一层
                }
            }
        }
    };
```

相关题目

- Next Permutation, 见 §2.1.13
- Permutation Sequence, 见 §2.1.14
- Permutations, 见 §??
- Combinations, 见 §??

9.7 Combinations

描述

Given two integers n and k, return all possible combinations of k numbers out of $1 \ldots n$.

For example, If $n = 4$ and $k = 2$, a solution is:

```
[
[2,4],
[3,4],
[2,3],
[1,2],
[1,3],
[1,4],
]
```

9.7.1 递归

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

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

9.7.2 迭代

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

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

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

相关题目

- Next Permutation, 见 §2.1.13
- Permutation Sequence, 见 §2.1.14
- Permutations, 见 §??
- Permutations II, 见 §??

9.8 Letter Combinations of a Phone Number

描述

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

图 9-1 Phone Keyboard

Input:Digit string "23"

Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

Note: Although the above answer is in lexicographical order, your answer could be in any order you want.

分析

无

9.8.1 递归

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

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

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

9.8.2 迭代

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

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

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

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

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

相关题目

- 无

第 10 章
广度优先搜索

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

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

广度优先搜索算法是按照广度优先的顺序遍历状态空间，一般采用队列实现。

【标记重复以节省重复搜索】【双向广度优先搜索】【迭代加深搜索】【剪枝】【启发式搜索】通过估价函数进行优化处理，例如最短路径时，$f(x) = dist(x) + h(x, B)$ 最小的点进行扩展。其中 $dist(x)$ 表示当前已知点到 x 的最短距离，而 $h(x, B)$ 表示启发函数从 x 到 B 得距离。具体过程：每次我们先利用堆找到 f 值的最小 x，让其出堆，然后从它开始拓展新节点。对于 x 的相邻节点 y，如果 y 还没有被扩展，将其入堆，否则考虑 $dist(x) + h(x, B)$, 其中 $dist(x)$ 表示边（x，y）的长度，如果这个值小于 $dist(y)$ 则更新 $dist(y)$ 并将其放入堆中。

10.1 走迷宫

描述

一个迷宫由一个 01 矩阵表示，每个单元格要么是空地（用 0 表示），要么是障碍物（用 1 表示）。你的任务是找到一条从入口到出口的最短路径。任何时候都不能在障碍物格子中，也不能走到迷宫之外。只能横着走或竖着走，不能斜着走。数据保证有唯一解。

输入

一个 5×5 的二维数组

输出

左上角到右下角的最短路径

样例输入

```
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

样例输出

(0, 0) (1, 0) (2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (3, 4) (4, 4)

分析

既然求的是“最短”，很自然的思路是用 BFS。举个例子，在如下图所示的迷宫中，假设入口是左上角 $(0, 0)$，我们就从入口开始用 BFS 遍历迷宫，就可以算出从入口到所有点的最短路径（如图 10-1(a) 所示），以及这些路径上每个节点的前驱（如图 10-1(b) 所示）。

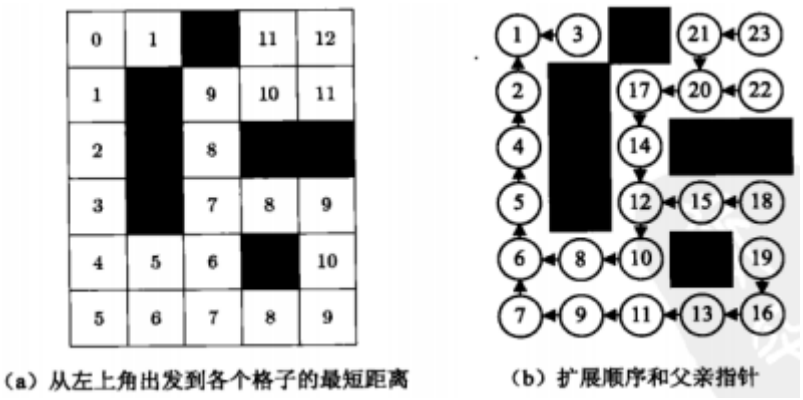


图 10-1 用 BFS 求迷宫中最短路径

代码

maze.c

```
/* POJ 3984 迷宫问题, http://poj.org/problem?id=3984 */
#include <cstdio>
#include <cstring>
#include <queue>

const int MAXN = 5;

// 迷宫的行数, 列数
int m = MAXN, n = MAXN;
// 迷宫, 0 表示空地, 1 表示障碍物
int map[MAXN][MAXN];

// 四个方向
const char name[4] = { 'U', 'R', 'D', 'L' };
const int dx[4] = { -1, 0, 1, 0 }; // 行
const int dy[4] = { 0, 1, 0, -1 }; // 列


typedef struct state_t {
    int data;
    int action;
    int father;
} state_t;

const int STATE_MAX = MAXN * MAXN;  /* 状态总数 */

state_t nodes[STATE_MAX];

int state_hash(const state_t &s);

int state_index(const state_t &s) {
    return state_hash(s);
}

void print_action(const int end) {
    if (nodes[end].father == -1) return;

    print_action(nodes[end].father);
    putchar(name[nodes[end].action]);
}

void print_path(const int end) {
    if (nodes[end].father == -1) {
        printf("(%d, %d)\n", end / n, end % n);
        return;
```

```
    }
    print_path(nodes[end].father);
    printf("(%d, %d)\n", end / n, end % n);
}

void hashset_init();

bool hashset_find(const state_t &s);

void hashset_insert(const state_t &s);

void state_extend_init(const state_t &s);

bool state_extend(const state_t &s, state_t &next);

bool state_is_target(const state_t &s);

int bfs(state_t &start) {
    queue<state_t> q;
    hashset_init();

    start.action = -1;
    start.father = -1;

    nodes[state_index(start)] = start;
    hashset_insert(start);
    if (state_is_target(start))
        return state_index(start);
    q.push(start);

    while (!q.empty()) {
        const state_t s = q.front(); q.pop();
        state_t next;

        state_extend_init(s);
        while (state_extend(s, next)) {
            if (state_is_target(next)) {
                return state_index(next);
            }
            q.push(next);
            hashset_insert(next);
        }
    }
    return -1;
}

int main(void) {
    state_t start = {0, -1, -1}; /* 左上角为起点 */
    int end;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &map[i][j]);
        }
    }

    end = bfs(start);
    print_path(end);
    return 0;
}

/********** functions implement **************/
/* 存在完美哈希方案 */
```

```
const int HASH_CAPACITY = STATE_MAX;
bool visited[HASH_CAPACITY];

int state_hash(const state_t &s) {
    return s.data;
}

void hashset_init() {
    memset(visited, 0, sizeof(visited));
}

bool hashset_find(const state_t &s) {
    return visited[state_hash(s)] == true;
}

void hashset_insert(const state_t &s) {
    visited[state_hash(s)] = true;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4
/** 扩展点，即当前位置 */
int x, y;

void state_extend_init(const state_t &s) {
    action_cur = ACTION_BEGIN;
    x = s.data / n;
    y = s.data % n;
}

bool state_extend(const state_t &s, state_t &next) {
    while(action_cur < ACTION_END) {
        const int nx = x + dx[action_cur];
        const int ny = y + dy[action_cur];

        if (nx >= 0 && nx < m && ny >= 0 && ny < n && !map[nx][ny]) {
            next.data = nx * n + ny;

            if (!hashset_find(next)) { /* 判重 */
                /* 记录路径 */
                next.action = action_cur;
                next.father = state_hash(s);
                nodes[state_index(next)] = next;

                action_cur++;  /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

const state_t END = {24, -1, -1};
bool state_is_target(const state_t &s) {
    return s.data == END.data;
}
```

maze.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[1]第 108 页 6.4.2 节
- POJ 3984 迷宫问题, http://poj.org/problem?id=3984

与本题相似的题目：

- POJ 2049 Finding Nemo, http://poj.org/problem?id=2049

10.2 八数码问题

描述

编号为 1~8 的 8 个正方形滑块摆成 3 行 3 列，有一个格子空着，如图 10-2所示。

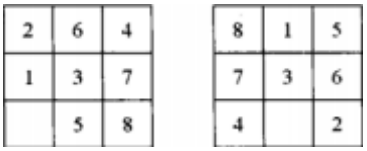


图 10-2 用 BFS 求迷宫中最短路径

每次可以把与空格相邻的滑块（有公共边才算相邻）移到空格中，而它原来的位置就成了新的空格。目标局面固定如下（用 x 表示空格）：

```
1 2 3
4 5 6
7 8 x
```

给定初始局面，计算出最短的移动路径。

输入

用一行表示一个局面，例如下面的这个局面：

```
1 2 3
x 4 6
7 5 8
```

可以表示为 1 2 3 x 4 6 7 5 8。

输出

如果有解答，输出一个由四个字母'r','l','u','d' 组成的移动路径。如果没有，输出"unsolvable"。

样例输入

```
2 3 4 1 5 x 7 6 8
```

样例输出

```
ullddrurdllurdruldr
```

分析

计算"最短"，很自然的想到 BFS。

如何表示一个状态？一个 3*3 的棋盘，首先可以想到用一个数组 `int data[9]` 来表示。把每个格子看做是整数的一位的话，可以用一个整数来表示，最大的棋盘 87654321x，表示成整数就是 876543210，没有超过 32 位整数的范围。

如何判重？用哈希表，自己实现或者用 STL。C++ STL 里有 `set`, C++ 11 新加入了 `unordered_set`。建议先用 STL 写一个版本，确保主算法正确，然后把 `set`(或 `unordered_set`) 替换成自己写的哈希表。

由于本题的特殊性，存在一种完美哈希 (perfect hashing) 方案。即**康托展开**。棋盘本身是一个排列，将排列转化为序数，用序数作为 hash 值。例如，1,2,3 这三个数字的全排列，按字典序，依次为

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

```
123 -- 0
132 -- 1
213 -- 2
231 -- 3
312 -- 4
321 -- 5
```

其中，左侧为排列，右侧为其序数。

此题更优的解法还有双向 BFS（见 §10.4），A* 算法（见 §10.5）。

代码

方案 1，完美哈希，使用康托展开。

eight_digits_bfs.c

```
/* POJ 1077 Eight, http://poj.org/problem?id=1077 */
#include <cstdio>
#include <cstring>
#include <queue>

const int DIGITS = 9; // 棋盘中数字的个数，也是变进制数需要的位数
const int MATRIX_EDGE = 3;       // 棋盘边长

/***** 一些常量 *****/
const int SPACE_NUMBER = 0; // 空格对应着数字 0
// 上下左右四个方向
const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};
const char name[] = { 'u', 'd', 'l', 'r' };

typedef char int8_t;

/**
 * @strut 状态
 */
typedef struct state_t {
    int8_t data[DIGITS];  /** 状态的数据. */
    int action; /* 由父状态移动到本状态的动作 */
    int father; /* 父状态在 nodes[] 中的下标，也即父状态的哈希值 */
    int count;  /** 所花费的步骤数（也即路径长度-1） */
} state_t;

// 3x3 的棋盘，状态最多有 9! 种
const int STATE_MAX = 362880;  /* 状态总数 */

state_t nodes[STATE_MAX+1];

int state_hash(const state_t &s);

int state_index(const state_t &s) {
    return state_hash(s);
}

/**
 * @brief 打印动作序列.
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_action(const int end) {
    if (nodes[end].father == -1) return;

    print_action(nodes[end].father);
    putchar(name[nodes[end].action]);
}
```

```
void hashset_init();

bool hashset_find(const state_t *s);

void hashset_insert(const state_t *s);

void state_extend_init(const state_t *s);

bool state_extend(const state_t *s, state_t *next);

bool state_is_target(const state_t *s);


int bfs(state_t *start) {
    queue<state_t> q;
    hashset_init();

    start->action = -1;
    start->father = -1;
    start->count = 0;

    nodes[state_index(start)] = *start;
    hashset_insert(start);
    if (state_is_target(start))
        return state_index(start);
    q.push(*start);

    while (!q.empty()) {
        const state_t s = q.front(); q.pop();
        state_t next;

        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                // printf("%d\n", next.count);
                return state_index(&next);
            }
            q.push(next);
            hashset_insert(&next);
        }
    }
    return -1;
}

/**
 * @brief 输入.
 * @return 无
 */
void input(state_t *start) {
    int ch;
    for (int i = 0; i < DIGITS; ++i) {
        do {
            ch = getchar();
        } while ((ch != EOF) && ((ch < '1') || (ch > '8')) && (ch != 'x'));
        if (ch == EOF) return;
        if (ch == 'x') start->data[i] = 0; // x 映射成数字 0
        else           start->data[i] = ch - '0';
    }
}

/** for wikioi 1225 */
void input1(state_t *start) {
    int n;
```

```
    scanf("%d", &n);

    /* 将整数转化为棋盘 */
    for(int i = DIGITS-1; i >= 0; i--) {
        start->data[i] = n % 10;
        n /= 10;
    }
}

int main(void) {
    state_t start;
    int end; /* 目标状态在 nodes[] 中的下标 */
    input(&start);

    end = bfs(&start);

    print_action(end);
    printf("\n");
    return 0;
}

/********** functions implement **************/

/********** 方案 1，完美哈希，使用康托展开 **************/

// 9 位变进制数（空格）能表示 0 到 (9!-1) 内的所有自然数，恰好有 9! 个，
// 与状态一一对应，因此可以把状态一一映射到一个 9 位变进制数

// 9 位变进制数，每个位数的单位，0!~8!
const int fac[] = {40320, 5040, 720, 120, 24, 6, 2, 1, 1};
/* 哈希表容量，要大于状态总数，若存在完美哈希方案，则等于状态总数 */
const int HASH_CAPACITY = STATE_MAX;

bool visited[HASH_CAPACITY];

int state_hash(const state_t *s) {
    int key = 0;
    for (int i = 0; i < DIGITS; i++) {
        int cnt = 0;  /* 逆序数 */
        for (int j = i + 1; j < DIGITS; j++) if (s->data[i] > s->data[j]) cnt++;
        key += fac[i] * cnt;
    }
    return key;
}

void hashset_init() {
    memset(visited, 0, sizeof(visited));
}

bool hashset_find(const state_t *s) {
    return visited[state_hash(s)] == true;
}

void hashset_insert(const state_t *s) {
    visited[state_hash(s)] = true;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4

/* 扩展点，即 0 的位置 */
int z;
```

```
void state_extend_init(const state_t *s) {
    action_cur = ACTION_BEGIN;
    for (z = 0; z < DIGITS; z++) {
        if (s->data[z] == SPACE_NUMBER) {
            break;  // 找 0 的位置
        }
    }
}

bool state_extend(const state_t *s, state_t *next) {
    const int x = z / MATRIX_EDGE; // 行
    const int y = z % MATRIX_EDGE; // 列

    while (action_cur < ACTION_END) {
        const int newx = x + dx[action_cur];
        const int newy = y + dy[action_cur];
        const int newz = newx * MATRIX_EDGE + newy;

        if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
                newy < MATRIX_EDGE) { // 没有越界
            *next = *s;
            next->data[newz] = SPACE_NUMBER;
            next->data[z] = s->data[newz];
            next->count = s->count + 1;
            if (!hashset_find(next)) { /* 判重 */
                next->action = action_cur;
                next->father = state_hash(s);
                /* 记录路径 */
                nodes[state_index(next)] = *next;
                action_cur++; /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {{1, 2, 3, 4, 5, 6, 7, 8, 0}, -1, -1};
// for wikioi 1225
const state_t END1 = {{1, 2, 3, 8, 0, 4, 7, 6, 5}, -1, -1};

bool state_is_target(const state_t *s) {
    return memcmp(s->data, END.data, DIGITS * sizeof(int8_t)) == 0;
}
```

eight_digits_bfs.c

方案 2，不知道是否存在完美哈希方案，但能够预估状态个数的上限，用树双亲表示法存储路径，用自己实现的哈希表判重。

eight_digits_bfs2.c

```
/* POJ 1077 Eight, http://poj.org/problem?id=1077 */
#include <cstdio>
#include <cstring>
#include <queue>
#include <algorithm>

const int DIGITS = 9; // 棋盘中数字的个数，也是变进制数需要的位数
const int MATRIX_EDGE = 3;       // 棋盘边长

/***** 一些常量 *****/
const int SPACE_NUMBER = 0; // 空格对应着数字 0
// 上下左右四个方向
const int dx[] = {-1, 1, 0, 0};
```

```
const int dy[] = {0, 0, -1, 1};
const char name[] = { 'u', 'd', 'l', 'r' };

typedef signed char int8_t;

/** 状态 */
struct state_t {
    int8_t data[DIGITS];  /** 状态的数据. */
    int action; /* 由父状态移动到本状态的动作 */
    int index;  /** 本状态在 nodes[] 中的下标 */
    int father; /** 父状态在 nodes[] 中的下标 */
    int count;  /** 所花费的步骤数（也即路径长度-1） */
};

// 3x3 的棋盘，状态最多有 9! 种
const int STATE_MAX = 362880;  /* 状态总数 */

state_t nodes[STATE_MAX+1];
int path_index = 0;

int state_index(const state_t &s) {
    return s.index;
}

/**
 * @brief 打印动作序列.
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_action(const int end) {
    if (nodes[end].father == -1) return;

    print_action(nodes[end].father);
    putchar(name[nodes[end].action]);
}

void hashset_init();

bool hashset_find(const state_t &s);

void hashset_insert(const state_t &s);

void state_extend_init(const state_t &s);

bool state_extend(const state_t &s, state_t &next);

bool state_is_target(const state_t &s);

int bfs(state_t &start) {
    queue<state_t> q;
    hashset_init();

    start.action = -1;
    start.index = path_index++;
    start.father = -1;
    start.count = 0;

    nodes[state_index(start)] = start;
    hashset_insert(start);
    if (state_is_target(start))
        return state_index(start);
    q.push(start);

    while (!q.empty()) {
```

```
        const state_t s = q.front(); q.pop();
        state_t next;

        state_extend_init(s);
        while (state_extend(s, next)) {
            if (state_is_target(next)) {
                // printf("%d\n", next.count);
                return state_index(next);
            }
            q.push(next);
            hashset_insert(next);
        }
    }
    return -1;
}

/**
 * @brief 输入.
 * @return 无
 */
void input(state_t &start) {
    int ch;
    for (int i = 0; i < DIGITS; ++i) {
        do {
            ch = getchar();
        } while ((ch != EOF) && ((ch < '1') || (ch > '8')) && (ch != 'x'));
        if (ch == EOF) return;
        if (ch == 'x') start.data[i] = 0; // x 映射成数字 0
        else           start.data[i] = ch - '0';
    }
}

/** for wikioi 1225 */
void input1(state_t &start) {
    int n;
    scanf("%d", &n);

    /* 将整数转化为棋盘 */
    for(int i = DIGITS-1; i >= 0; i--) {
        start.data[i] = n % 10;
        n /= 10;
    }
}

int main(void) {
    state_t start;
    int end;
    input(start);

    end = bfs(start);

    print_action(end);
    printf("\n");
    return 0;
}

/********** functions implement **************/

/********** 方案 2 不知道完美哈希方案，自己实现哈希表 **************/
/** 哈希表取模的质数，也即哈希桶的个数，越大越好，不过一般小于 HASH_SET_CAPACITY. */
const int PRIME = 99997;

/* 哈希表容量，要大于状态总数 */
const int HASH_SET_CAPACITY = STATE_MAX + 1;
```

```
int head[PRIME];
int next[HASH_SET_CAPACITY];

/** 元素的哈希函数 */
int state_hash(const state_t &s) {
    int ret = 0;
    for(int i = 0; i < DIGITS; i++) ret = ret * 10 + s.data[i];
    return ret % PRIME;
}

void hashset_init() {
    fill(head, head + PRIME, -1);
    fill(next, next + PRIME, -1);
}

bool hashset_find(const state_t &s) {
    // 从表头开始查找单链表
    for (int i = head[state_hash(s)]; i != -1; i = next[i]) {
        // 找到了
        if(memcmp(nodes[i].data, s.data,
                DIGITS * sizeof(int8_t)) == 0)
            return true;
    }
    return false;
}

void hashset_insert(const state_t &s) {
    const int h = state_hash(s);
    // 从表头开始查找单链表
    for (int i = head[h]; i != -1; i = next[i]) {
        // 找到了
        if (memcmp(nodes[i].data, s.data,
                DIGITS * sizeof(int8_t)) == 0)
            return;
    }
    next[s.index] = head[h]; /* 插入到首节点前面, 头查法 */
    head[h] = s.index;
    return;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4

/* 扩展点, 即 0 的位置 */
int z;

void state_extend_init(const state_t &s) {
    action_cur = ACTION_BEGIN;
    for (z = 0; z < DIGITS; z++) {
        if (s.data[z] == SPACE_NUMBER) {
            break;  // 找 0 的位置
        }
    }
}

bool state_extend(const state_t &s, state_t &next) {
    const int x = z / MATRIX_EDGE; // 行
    const int y = z % MATRIX_EDGE; // 列

    while (action_cur < ACTION_END) {
        const int newx = x + dx[action_cur];
        const int newy = y + dy[action_cur];
```

```
        const int newz = newx * MATRIX_EDGE + newy;

        next.count = s.count + 1;
        if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
                newy < MATRIX_EDGE) { // 没有越界
            next = s;
            next.data[newz] = SPACE_NUMBER;
            next.data[z] = s.data[newz];

            if (!hashset_find(next)) { /* 判重 */
                next.action = action_cur;
                next.index = path_index++;
                next.father = s.index;
                /* 记录路径 */
                nodes[state_index(next)] = next;
                action_cur++; /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {{1, 2, 3, 4, 5, 6, 7, 8, 0}, -1, -1};
// for wikioi 1225
const state_t END1 = {{1, 2, 3, 8, 0, 4, 7, 6, 5}, -1, -1};

bool state_is_target(const state_t &s) {
    return memcmp(s.data, END.data, DIGITS * sizeof(int8_t)) == 0;
}
```

eight_digits_bfs2.c

方案 3，不知道完美哈希方案，但能够预估状态个数的上限制，用树双亲表示法存储路径，用标准库的哈希表判重。

eight_digits_bfs3.cpp

```
//前面的代码与方案 2 一摸一样
//...

/********** 方案 3 不知道完美哈希方案，使用标准库的哈希表 **************/

// 重载 state_t 的 == 操作符
typedef struct state_t {
    int8_t data[DIGITS];  /** 状态的数据. */
    int action; /* 由父状态移动到本状态的动作 */
    int index;  /** 本状态在 nodes[] 中的下标 */
    int father; /** 父状态在 nodes[] 中的下标 */
    int count;  /** 所花费的步骤数（也即路径长度-1） */

    bool operator==(const state_t& other) const {
        return memcmp(data, other.data, DIGITS * sizeof(int8_t)) == 0;
    }
} state_t;

#include <unordered_set>

// 定制一个哈希函数
namespace std {
template<> struct hash<state_t> {
    size_t operator()(const state_t & x) const {
        int i;
        int ret = 0;
        for (i = 0; i < DIGITS; i++)
            ret = ret * 10 + x.data[i];
```

```
        return ret;
    }
};
}

unordered_set<state_t> visited;

void hashset_init() {
    visited.clear();
}

bool hashset_find(const state_t &s) {
    return visited.count(s) > 0;
}

void hashset_insert(const state_t &s) {
    visited.insert(s);
}

//...
//后面的代码也与方案 2 一模一样
```

eight_digits_bfs3.cpp

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①] 第 131 页 7.5.3 节
- POJ 1077 Eight, http://poj.org/problem?id=1077
- wikioi 1225 八数码难题, http://www.wikioi.com/problem/1225/

与本题相似的题目：

- POJ 2893 M × N Puzzle, http://poj.org/problem?id=2893

10.3 四子连棋

描述

在一个 4*4 的棋盘上摆放了 14 颗棋子，其中有 7 颗白色棋子，7 颗黑色棋子，有两个空白地带，任何一颗黑白棋子都可以向上下左右四个方向移动到相邻的空格，这叫行棋一步，黑白双方交替走棋，任意一方可以先走，如果某个时刻使得任意一种颜色的棋子形成四个一线（包括斜线），这样的状态为目标棋局。

输入

一个 4*4 的初始棋局，黑棋子用 B 表示，白棋子用 W 表示，空格地带用 O 表示。

输出

移动到目标棋局的最少步数。

样例输入

```
BWBO
WBWB
BWBW
WBWO
```

[①] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

样例输出

```
5
```

分析

求最少步数，很自然的想到广搜。

如何表示一个状态？用一个二维数组 `int board[4][4]` 表示，还需要记录该状态是由白子还是黑子移动而导致的，走到该状态已经花费的步数。

如何扩展节点？每一步，从队列弹出一个状态，两个空格都可以向四个方向扩展，把得到的状态入队列。

如何判重？棋盘用二维矩阵存储，用 0 表示空格，1 表示黑色，2 表示白色，所以最后可以看成一个 16 位的三进制数。用这个数作为棋盘的编码，就可以用来判重了。注意，本题要黑白交替走，所以我们要区分状态是由白子还是黑子移动而导致的。

可以用 C++ 的 `map` 来判重，

```
/* visited[0] 记录白子的历史， visited[1] 记录黑子的历史. */
map<int, bool> visited[2];
```

也可以开一个大数组当做哈希表，

```
#define HASH_MOD 43036875 /* hash 表大小 */
/* visited[0] 记录白子的历史， visited[1] 记录黑子的历史. */
bool visited[2][HASH_MOD];
```

代码

four_adjacent.cpp

```
/** wikioi 1004 四子连棋  , http://www.wikioi.com/problem/1004 */
#include <cstdio>
#include <cstring>
#include <queue>

#define LEN 4   /* 边长 */

/* 右，左，上，下（左下角为坐标原点）*/
const int dx[] = { 1, -1, 0, 0 };
const int dy[] = { 0, 0, 1, -1 };


/**
 * @strut 状态
 */
typedef struct state_t {
    // 状态的数据
    int board[LEN][LEN]; /* 棋局，1 表示黑子，2 表示白子，0 表示空白 */
    int color; /* 本状态是由白子还是黑子移动而导致的 */
    int count;  /** 所花费的步骤数（也即路径长度-1），求路径长度时需要 */
} state_t;

void hashset_init();

bool hashset_find(const state_t &s);

void hashset_insert(const state_t &s);

void state_extend_init(const state_t &s);

bool state_extend(const state_t &s, state_t &next);

bool state_is_target(const state_t &s);

void bfs(state_t &start) {
    queue<state_t> q;
```

```
    hashset_init();

    start.count = 0;
    start.color = 1;

    hashset_insert(start);
    q.push(start);

    start.color = 2;

    hashset_insert(start); // 千万别忘记了标记此处的访问记录
    if (state_is_target(start)) /* 如果起点就是终点，返回 */
        return;
    q.push(start);

    while (!q.empty()) {
        const state_t s = q.front(); q.pop();
        state_t next;

        state_extend_init(s);
        while (state_extend(s, next)) {
            if (state_is_target(next)) {
                printf("%d\n", next.count);
                return;
            }
            q.push(next);
            hashset_insert(next);
        }
    }
}

int main() {
    char s[LEN + 1];
    state_t start;

    for (int i = 0; i < LEN; i++) {
        scanf("%s", s);
        for (int j = 0; j < LEN; j++) {
            if (s[j] == 'B') start.board[i][j] = 1;
            else if (s[j] == 'W') start.board[i][j] = 2;
            else start.board[i][j] = 0;
        }
    }

    bfs(start);
    return 0;
}

/************ functions implement ************/

/* 哈希表容量，要大于状态总数，若存在完美哈希方案，则等于状态总数 */
#define HASH_CAPACITY 43036875

/** 哈希表，标记状态是否已访问过。
 * visited[0] 记录白子的历史，visited[1] 记录黑子的历史.
 */
bool visited[2][HASH_CAPACITY];

#define RADIX 3 /* 三进制 */

/**
 * @brief 计算状态的哈希值。
 * 棋盘用二维矩阵存储，用 0 表示空格，1 表示黑色，2 表示白色，所以最后可以看成
 * 一个 16 位的三进制数。最大为
```

```
 * 2222
 * 2221
 * 1111
 * 1100
 * 值为 43036875。
 * @return 棋盘所表示的三进制数转化为十进制数
 */
//TODO: 共 C16 7×C9 2 个状态，用类似康托的方法储存
int state_hash(const state_t &s) {
    int ret = 0;

    for (int i = 0; i < LEN; i++) {
        for (int j = 0; j < LEN; j++) {
            ret = ret * RADIX + s.board[i][j];
        }
    }
    return ret;
}

void hashset_init() {
    fill(visited[0], visited[0] + HASH_CAPACITY, false);
    fill(visited[1], visited[1] + HASH_CAPACITY, false);
}

bool hashset_find(const state_t &s) {
    return visited[s.color - 1][state_hash(s)] == true;
}

void hashset_insert(const state_t &s) {
    visited[s.color - 1][state_hash(s)] = true;
}

/* 扩展的时候，先定空格，再定方向 */
/* 记录当前方向，例如 action_cur[0] 记录了第一个空格，当前在扩展哪个方向
 */
int action_cur[2];
#define ACTION_BEGIN 0
#define ACTION_END 4

typedef struct point_t {
    int x, y;
} point_t;

/* 记录当前在扩展哪一个空格，值为 0 或 1 */
int space_cur;
/* 两个空格的位置 */
point_t extend_pos[2];

void state_extend_init(const state_t &s) {
    action_cur[0] = ACTION_BEGIN;
    action_cur[1] = ACTION_BEGIN;
    space_cur = 0;

    int k = 0;
    // 寻找两个空白的格子的位置
    for (int i = 0; i < LEN; i++) {
        for (int j = 0; j < LEN; j++) {
            if (s.board[i][j] == 0) {
                extend_pos[k].x = i;
                extend_pos[k].y = j;
                k++;
            }
        }
    }
```

```
}

bool state_extend(const state_t &s, state_t &next) {
    for (int i = 0; i < 2; i++) { /* 先第一个空格，再第二个空格 */
        while (action_cur[i] < ACTION_END) {
            const int x = extend_pos[i].x;
            const int y = extend_pos[i].y;
            int nextx = x + dx[action_cur[i]];
            int nexty = y + dy[action_cur[i]];
            next = s;
            next.count = s.count + 1;
            next.color = 3 - s.color;

            if (nextx >= 0 && nextx < LEN && nexty >= 0 && nexty < LEN
                    /* 必须黑白交替走 */
                    && next.color == s.board[nextx][nexty]) {
                /* swap */
                {
                    int temp = next.board[x][y];
                    next.board[x][y] = next.board[nextx][nexty];
                    next.board[nextx][nexty] = temp;
                }

                if (!hashset_find(next)) { /* 判重 */
                    action_cur[i]++; /* return 前别忘了增 1 */
                    return true;
                }
            }
            action_cur[i]++;
        }
    }
    return false;
}

bool state_is_target(const state_t &s) {
    for (int i = 0; i < LEN; i++) {  /* 逐行检查 */
        int flag = 1;  /* 某一行全是同一颜色 */
        for (int j = 1; j < LEN; j++)
            if (s.board[i][j - 1] != s.board[i][j])
                flag = 0;
        if (flag)
            return 1;
    }
    for (int j = 0; j < LEN; j++) { //逐列检查
        int flag = 1;  /* 某一行全是同一颜色 */
        for (int i = 1; i < LEN; i++)
            if (s.board[i][j] != s.board[i - 1][j]) flag = 0;
        if (flag) return 1;
    }
    /* 斜线 */
    if (s.board[0][0] == s.board[1][1] && s.board[1][1] == s.board[2][2]
            && s.board[2][2] == s.board[3][3])
        return 1;
    if (s.board[0][3] == s.board[1][2] && s.board[1][2] == s.board[2][1]
            && s.board[2][1] == s.board[3][0])
        return 1;
    return 0;
}
```

four_adjacent.cpp

相关的题目

与本题相同的题目：

- wikioi 1004 四子连棋, http://www.wikioi.com/problem/1004/

与本题相似的题目：

- None

10.4 双向 BFS

10.4.1 八数码问题

题目见 §10.2。

代码

eight_digits_bibfs.c

eight_digits_bibfs.c

10.5 A* 算法

A* 算法 = 宽搜 + 优先队列

将广搜模板（见第 §10.6节）中的队列改为优先队列，设计好当前状态距离目标状态的预估距离 $h()$，就变成了 A* 算法！

10.5.1 八数码问题

题目见 §10.2。

代码

eight_digits_astar.c

```
/** POJ 1077 Eight, http://poj.org/problem?id=1077
 简单解释几个要点，便于理解代码.
1. 怎么判断是否有解？只要计算出的逆序个数总和为奇数，该数据必然无解
2. 如何判断某一状态是否到过？本题存在一种完美哈希方案，即用康托展开。
        详见 http://128kj.iteye.com/blog/1699795
3. 使用优先队列，即堆，加速挑选最优值。
4. 函数 g= 此状态在搜索树中已经走过的路径的节点数.
5. 估价函数 h ，采用曼哈顿距离，见代码 calcH 函数。曼哈顿距离的定义是，
 假设有两个点 (x1,y1),(x2,y2)，则曼哈顿距离 L1=|x1-x2| + |y1-y2|
 */
#include <cstdio>
#include <cstring>
#include <cmath>
#include <queue>

#define DIGITS 9 // 棋盘中数字的个数，也是变进制数需要的位数
#define     MATRIX_EDGE 3       // 棋盘边长
#define RADIX 10

/***** 一些常量 *****/
const int SPACE_NUMBER = 0; // 空格对应着数字 0
// 上下左右四个方向
const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};
const char name[] = { 'u', 'd', 'l', 'r' };

// 目标状态
const int GOAL = 123456780;
// 每个数字在棋盘中的位置，例如 0，在 (2,2)=8 这个位置上
int GOAL_POS[DIGITS];
```

```
/** 状态 */
struct state_t {
    int board;  /** 状态的数据，即棋局. */
    int action; /* 由父状态移动到本状态的动作 */
    int father; /* 父状态在 nodes[] 中的下标，也即父状态的哈希值 */
    int count;  /** 所花费的步骤数（也即路径长度-1），作为 g */
    int h; /** 距离目标状态的估算距离 */

    bool operator>(const state_t &other) const {
        // priority_queue 默认是大根堆，反一下，就是小根堆了
        return (count + h) > (other.count + other.h);
    }
};

// 3x3 的棋盘，状态最多有 9! 种
#define STATE_MAX 362880  /* 状态总数 */

state_t nodes[STATE_MAX+1];

int state_hash(const state_t &s);

int state_index(const state_t &s) {
    return state_hash(s);
}

/**
 * @brief 打印动作序列.
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_action(const int end) {
    if (nodes[end].father == -1) return;

    print_action(nodes[end].father);
    putchar(name[nodes[end].action]);
}

void hashset_init();

bool hashset_find(const state_t &s);

void hashset_insert(const state_t &s);

void state_extend_init(const state_t &s);

bool state_extend(const state_t &s, state_t &next);

bool state_is_target(const state_t &s);

/**
 * 距离目标状态的估算距离 h。
 * @param s 状态
 * @return h
 */
static int state_get_h(const state_t &state) {
    int h = 0;
    int s = state.board;

    for (int i = DIGITS - 1; i >= 0; --i) {
        const int p = s % RADIX;
        s /= RADIX;
        /* 曼哈顿距离 */
```

```
        h += abs((float)(i / MATRIX_EDGE - GOAL_POS[p] / MATRIX_EDGE)) +
                abs((float)(i % MATRIX_EDGE - GOAL_POS[p] % MATRIX_EDGE));
    }
    return h;
}

/* 计算 GOAL_POS */
static void calc_goal_pos() {
    int cur = GOAL;
    for (int i = DIGITS-1; i >= 0 ; i--) {
        int digit = cur % RADIX;
        GOAL_POS[digit] = i;
        cur /= RADIX;
    }
}

int bfs(state_t &start) {
    priority_queue<state_t, vector<state_t>,
                                greater<state_t> > q;
    calc_goal_pos();
    hashset_init();

    start.action = -1;
    start.father = -1;
    start.count = 0;
    start.h = state_get_h(start);

    nodes[state_index(start)] = start;
    hashset_insert(start);
    if (state_is_target(start))
        return state_index(start);
    q.push(start);

    while (!q.empty()) {
        const state_t s = q.top(); q.pop();
        //const state_t s = q.front(); q.pop();
        state_t next;

        state_extend_init(s);
        while (state_extend(s, next)) {
            if (state_is_target(next)) {
                // printf("%d\n", next.count);
                return state_index(next);
            }
            q.push(next);
            hashset_insert(next);
        }
    }
    return -1;
}

static void int_to_board(int n, int board[DIGITS]) {
    for (int i = DIGITS - 1; i >= 0; i--) {
        board[i] = n % RADIX;
        n /= RADIX;
    }
}

static int board_to_int(const int board[DIGITS]) {
    int s = 0;
    for (int i = 0; i < DIGITS; i++)
        s = s * RADIX + board[i];
    return s;
}
```

```
/**
 * @brief 输入.
 * @return 无
 */
void input(state_t &start) {
    int ch;
    int board[DIGITS];
    for (int i = 0; i < DIGITS; ++i) {
        do {
            ch = getchar();
        } while ((ch != EOF) && ((ch < '1') || (ch > '8')) && (ch != 'x'));
        if (ch == EOF) return;
        if (ch == 'x') board[i] = 0; // x 映射成数字 0
        else           board[i] = ch - '0';
    }
    start.board = board_to_int(board);
}

/** for wikioi 1225 */
void input1(state_t &start) {
    scanf("%d", &start.board);
}

/**
 * 计算一个排列的逆序数, 0 除外.
 */
static int inversion_count(int permutation) {
    int d[DIGITS];
    int c = 0; // 逆序数

    for(int i = DIGITS - 1; i >=0; i--) {
        d[i] = permutation % RADIX;
        permutation /= RADIX;
    }

    for (int i = 1; i < DIGITS; i++)  if (d[i] != SPACE_NUMBER) {
        for (int j = 0; j < i; j++) {
            if(d[j] != SPACE_NUMBER) {
                if (d[j] > d[i]) {
                    c++;
                }
            }
        }
    }
    return c;
}

/**
 * 判断是否有解.
 *
 * 求出除 0 之外所有数字的逆序数之和, 也就是每个数字后面比它小的数字的个数的和,
 * 称为这个状态的逆序。
 *
 * 若起始状态和目标状态的逆序数奇偶性相同, 则可相互到达, 否则不可相互到达。
 *
 * @param s 目标状态
 * @return 1 表示有解, 0 表示无解
 */
static int solvable(const int s) {
    return (inversion_count(s) + inversion_count(GOAL)) % 2 == 0;
}

int main(void) {
```

```
    state_t start;
    int end; /* 目标状态在 nodes[] 中的下标 */
    input(start);

    if (!solvable(start.board)) return 0;

    end = bfs(start);

    print_action(end);
    printf("\n");
    return 0;
}

/********** functions implement **************/

/********** 方案 1，完美哈希，使用康托展开 **************/

// 9 位变进制数（空格）能表示 0 到 (9!-1) 内的所有自然数，恰好有 9! 个，
// 与状态一一对应，因此可以把状态一一映射到一个 9 位变进制数

// 9 位变进制数，每个位数的单位，0!~8!
const int fac[] = {40320, 5040, 720, 120, 24, 6, 2, 1, 1};
/* 哈希表容量，要大于状态总数，若存在完美哈希方案，则等于状态总数 */
#define HASH_CAPACITY STATE_MAX

bool visited[HASH_CAPACITY];

/**
 * @brief 计算状态的 hash 值，这里用康托展开，是完美哈希.
 * @param[in] s 当前状态
 * @return 序数，作为 hash 值
 */
int state_hash(const state_t &state) {
    int board[DIGITS];
    int key = 0;

    int_to_board(state.board, board);

    for (int i = 0; i < DIGITS; i++) {
        int c = 0; // 逆序数
        for (int j = i + 1; j < DIGITS; j++) {
            if(board[j] < board[i]) {
                c++;
            }
        }
        key += c * fac[i];
    }

    return key;
}

void hashset_init() {
    fill(visited, visited + HASH_CAPACITY, false);
}

bool hashset_find(const state_t &s) {
    return visited[state_hash(s)] == true;
}

void hashset_insert(const state_t &s) {
    visited[state_hash(s)] = true;
}

int action_cur;
```

```
#define ACTION_BEGIN 0
#define ACTION_END 4

/* 扩展点，即 0 的位置 */
int z;
int board[DIGITS];  /* 棋盘，暂存数据 */

void state_extend_init(const state_t &s) {
    action_cur = ACTION_BEGIN;

    int_to_board(s.board, board);
    for (z = 0; z < DIGITS; z++) {
        if (board[z] == SPACE_NUMBER) {
            break;  // 找 0 的位置
        }
    }
}

bool state_extend(const state_t &s, state_t &next) {
    const int x = z / MATRIX_EDGE; // 行
    const int y = z % MATRIX_EDGE; // 列

    while (action_cur < ACTION_END) {
        const int newx = x + dx[action_cur];
        const int newy = y + dy[action_cur];
        const int newz = newx * MATRIX_EDGE + newy;

        if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
                newy < MATRIX_EDGE) { // 没有越界
            board[z] = board[newz];
            board[newz] = SPACE_NUMBER;
            next = s;
            next.board = board_to_int(board);
            board[newz] = board[z]; /* 恢复 s 的棋盘 */
            board[z] = SPACE_NUMBER;

            next.count = s.count + 1;
            next.h = state_get_h(next);
            if (!hashset_find(next)) { /* 判重 */
                next.action = action_cur;
                next.father = state_index(s);
                /* 记录路径 */
                nodes[state_index(next)] = next;
                action_cur++; /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {123456780, -1, -1};
// for wikioi 1225
const state_t END1 = {123804765, -1, -1};

bool state_is_target(const state_t &s) {
    return s.board == END.board;
}
```

eight_digits_astar.c

10.6 小结

10.6.1 适用场景

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

10.6.2 思考的步骤

1. 是求路径长度，还是路径本身（或动作序列）？
   (a) 如果是求路径长度，则状态里面要存路径长度
   (b) 如果是求路径本身或动作序列
      i. 要用一棵树存储宽搜过程中的路径
      ii. 是否可以预估状态个数的上限？能够预估状态总数，则开一个大数组，用树的双亲表示法；如果不能预估状态总数，则要使用一棵通用的树。这一步也是第 4 步的必要不充分条件。
2. 如何表示状态？即一个状态需要存储哪些些必要的数据，才能够完整提供如何扩展到下一步状态的所有信息。一般记录当前位置或整体局面。
3. 如何扩展状态？这一步跟第 2 步相关。状态里记录的数据不同，扩展方法就不同。对于固定不变的数据结构（一般题目直接给出，作为输入数据），如二叉树，图等，扩展方法很简单，直接往下一层走，对于隐式图，要先在第 1 步里想清楚状态所带的数据，想清楚了这点，那如何扩展就很简单了。
4. 关于判重，状态是否存在完美哈希方案？即将状态一一映射到整数，互相之间不会冲突。
   (a) 如果不存在，则需要使用通用的哈希表（自己实现或用标准库，例如 `unordered_set`）来判重；自己实现哈希表的话，如果能够预估状态个数的上限，则可以开两个数组，head 和 next，表示哈希表，参考第 §10.2节方案 2。
   (b) 如果存在，则可以开一个大布尔数组，作为哈希表来判重，且此时可以精确计算出状态总数，而不仅仅是预估上限。
5. 目标状态是否已知？如果题目已经给出了目标状态，可以带来很大便利，这时候可以从起始状态出发，正向广搜；也可以从目标状态出发，逆向广搜；也可以同时出发，双向广搜。

10.6.3 代码模板

广搜需要一个队列，用于一层一层扩展，一个 hashset，用于判重，一棵树（只求长度时不需要），用于存储整棵树。

对于队列，如果用纯 C，需要造一个队列轮子；如果用 C++，可以用 `queue`，也可以把 `vector` 当做队列使用。当求长度时，有两种做法：

1. 只用一个队列，但在状态结构体 `state_t` 里增加一个整数字段 `step`，表示走到当前状态用了多少步，当碰到目标状态，直接输出 `step` 即可。这个方案，可以很方便的变成 A* 算法，把队列换成优先队列即可。
2. 用两个队列，`current, next`，分别表示当前层次和下一层，另设一个全局整数 `level`，表示层数（也即路径长度），当碰到目标状态，输出 `level` 即可。这个方案，状态可以少一个字段，节省内存。

对于 hashset，如果有完美哈希方案，用布尔数组 (`bool visited[STATE_MAX]` 或 `vector<bool> visited(STATE_MAX, false)`) 来表示；如果没有，可以用 STL 里的 `set` 或 `unordered_set`。

对于树，如果用 STL，可以用 `unordered_map<state_t, state_t > father` 表示一颗树，代码非常简洁。如果能够预估状态总数的上限（设为 STATE_MAX），可以用数组 (`state_t nodes[STATE_MAX]`)，即树的双亲表示法来表示树，效率更高，当然，需要写更多代码。

C++ 风格的模板。

bfs_template1.cpp

```
/**
 * @brief 反向生成路径.
 * @param[in] father 树
 * @param[in] target 目标节点
 * @return 从起点到 target 的路径
 */
template<typename state_t>
vector<state_t> gen_path(const unordered_map<state_t, state_t> &father,
        const state_t &target) {
    vector<state_t> path;
    path.push_back(target);

    state_t cur = target;
    while (father.find(cur) != father.end()) {
        cur = father.at(cur);
        path.push_back(cur);
    }
    reverse(path.begin(), path.end());

    return path;
}

/**
 * @brief 广搜.
 * @param[in] state_t 状态，如整数，字符串，一维数组等
 * @param[in] start 起点
 * @param[in] state_is_target 判断状态是否是目标的函数
 * @param[in] state_extend 状态扩展函数
 * @return 从起点到目标状态的一条最短路径
 */
template<typename state_t>
vector<state_t> bfs(state_t &start, bool (*state_is_target)(const state_t&),
        vector<state_t>(*state_extend)(const state_t&,
                unordered_set<string> &visited)) {
    queue<state_t> next, current; // 当前层，下一层
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father;

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

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

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

                next.push(new_state);
                // visited.insert(new_state); 必须放到 state_extend() 里
```

```
                father[new_state] = state;
            }
        }
        swap(next, current); //!!! 交换两个队列
    }

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

bfs_template1.cpp

C 风格的模板。

bfs_template2.cpp

```
#include <cstdio>
#include <cstring>

/***** 输入数据，用全局变量存放 *****/
...
/*
例如
int m = MAXN, n = MAXN;  // 迷宫的行数，列数
// 迷宫，0 表示空地，1 表示障碍物
int map[MAXN][MAXN];  // 迷宫，0 表示空地，1 表示障碍物
 */

/***** 一些常量 *****/
...
/* 例如
// 四个方向
const char name[4] = { 'U', 'R', 'D', 'L' };
const int dx[4] = { -1, 0, 1, 0 }; // 行
const int dy[4] = { 0, 1, 0, -1 }; // 列
*/

/** 状态 */
struct state_t {
    ... data1;  /** 状态的数据，可以有多个字段. */
    ... data2;  /** 状态的数据，可以有多个字段. */
    int action; /** 由父状态移动到本状态的动作，求动作序列时需要. */
    int index;  /** 本状态在 nodes[] 中的下标，求路径和动作序列时需要。
                    如果存在完美哈希，则不需要本字段（此时将
                   状态的哈希值作为下标，而哈希值可以直接计算出） */
    int father; /** 父状态在 nodes[] 中的下标，求路径或动作序列时需要 */
    int count;  /** 所花费的步骤数（也即路径长度-1），求路径长度时需要 */
};

/********** 如果题目要求输出路径或动作序列，则需要下面的变量和函数 ***********/

const int STATE_MAX ...  /* 状态总数 */
/**
 * 记录动作序列，树的双亲表示法.
 * 如果不能预估状态个数的上限，则不能用数组，要用通用树
 * 一般此时会有完美哈希方案，开一个大数组存储每个状态的动作，下标就是状态的哈希值
 * 其实一般用不了这么大的数组，广搜很很快就会结束
 */
state_t nodes[STATE_MAX];
int path_index = 0;  /** 每出现一个新状态，就增 1，将状态存到该位置，
                         如果有完美哈希方案，则不需要该变量. */

/**
```

```
 * @brief 返回状态在 nodes[] 中的下标.
 *
 * 起点状态没有父亲和动作为-1，因为它没有父状态，也就没有所谓的从父节点到它的动作。
 * @param[in] s 状态
 * @return 状态在 nodes[] 中的下标
 */
int state_index(const state& s) {
    /* 如果有完美哈希方案 */
    return state_hash(s);
    /* 否则 */
    return s.index;
}

/**
 * @brief 打印动作序列.
 * 如果有完美哈希方案，状态的哈希值就是下标。
 * 起点状态没有父亲和动作为-1，因为它没有父状态，也就没有所谓的从父节点到它的动作。
 * @param[in] end 终点状态的下标
 * @return 父状态
 */
void print_action(const int end) {
    if (nodes[end].father == -1) return;

    print_action(nodes[end].father);
    putchar(name[nodes[end].action]);
}

/**
 * @brief 打印坐标序列.
 * 如果有完美哈希方案，状态的哈希值就是下标。
 * 起点状态没有父亲和动作为-1，因为它没有父状态，也就没有所谓的从父节点到它的动作。
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_path(const int end) {
    if (nodes[end].father == -1) {
        printf("(%d, %d)\n", end / n, end % n);
        return;
    }
    print_path(nodes[end].father);
    printf("(%d, %d)\n", end / n, end % n);
}

/********** 如果题目要求输出路径或动作序列，则需要上面的变量和函数 ***********/

/** 初始化查找表. */
void hashset_init();

/**
 * @brief 状态判重.
 * 一般用哈希表 (set::unordered_set)，如果存在完美哈希，则用数组
 * @param[in] s 状态
 * @return 已经访问过，返回 true，否则 false
 */
bool hashset_find(const state_t &s);

/**
 * @brief 标记该状态已经被访问
 * @param[in] s 状态
 * @return 无
 */
void hashset_insert(const state_t &s);

/**
```

```
 * @brief 扩展第一个状态.
 * @param[in] s 状态
 * @param[out] next 第一个状态
 * @return 如果还有下一个状态，返回 true，否则返回 false
 */
void state_extend_init(const state_t &s);

/**
 * @brief 扩展下一个状态.
 * @param[in] s 状态
 * @param[out] next 下一个状态
 * @return 如果还有下一个状态，返回 true，否则返回 false
 */
bool state_extend(const state_t &s, state_t &next);

/**
 * @brief 判断状态是否为目标.
 * @param[in] s 状态
 * @return 如果已经达到目标状态，返回 true，否则 false
 */
bool state_is_target(const state_t &s);

/*
 * @brief 广搜
 *
 * @param[in] x 入口的 x 坐标
 * @param[in] y 入口的 y 坐标
 * @return 目标状态在 nodes[] 中的下标，如果不需要求路径，则声明为 void
 */
int bfs(state_t &start) {
    queue<state_t> q;
    hashset_init();

    start.action = -1;   /* 起点状态没有动作 */
    start.index = path_index++;
    start.father = -1;   /* 起点状态没有父状态 */
    start.count = 0;

    nodes[state_index(start)] = start;
    hashset_insert(start); // 千万别忘记了标记此处的访问记录
    if (state_is_target(start)) /* 如果起点就是终点，返回 */
        return state_index(start);
    q.push(start);

    while (!q.empty()) {
        const state_t s = q.front(); q.pop();
        state_t next;

        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                // printf("%d\n", next.count);
                return state_index(next);
            }
            q.push(next);
            hashset_insert(&next);
        }
    }
    return -1;
}

int main(void) {
    state_t start;
    int end; /* 目标状态在 nodes[] 中的下标 */
```

```
    input(start);

    end = bfs(start);

    print_action(end);
    printf("\n");
    print_path(end);
    return 0;
}

/************ functions implement ************/

/** 哈希表取模的质数，也即哈希桶的个数，越大越好，不过一般小于 HASH_SET_CAPACITY. */
#define PRIME ...  /* 99997 */

/* 哈希表容量，要大于状态总数，若存在完美哈希方案，则等于状态总数 */
#define HASH_CAPACITY STATE_MAX

/** 哈希表，标记状态是否已访问过。
 * 如果存在完美哈希方案，则用数组作为哈希表，否则用 unordered_set
 */
... visited
// 例如 bool visited[HASH_CAPACITY];
// 或者
// int head[PRIME];
// int next[HASH_CAPACITY];

/** 计算状态的哈希值.
 * 哈希值应该只依赖 state_t 的数据字段。
 */
int state_hash(const state_t &s) {
    ...
}

void hashset_init() {
    /* 如果是数组 */
    memset(visited, 0, sizeof(visited));
    /* 如果是 unordered_set */
    visited.clear();
}

bool hashset_find(const state_t &s) {
    /* 如果是数组 */
    return visited[state_hash(s)] == true;
    /* 如果是 unordered_set */
    return visited.count(s) > 0;
}

void hashset_insert(const state_t &s) {
    /* 如果是数组 */
    visited[state_hash(s)] = true;
    /* 如果是 unordered_set */
    visited.insert(s);
}

/** 扩展节点时，记录当前到了什么哪一步.
 * 是 state_extend_init() 和 state_extend() 的共享变量
 */
int action_cur;
/* 动作的范围 */
#define ACTION_BEGIN ...
#define ACTION_END ...

/* 扩展点的位置，是 state_extend_init() 和 state_extend() 的共享变量 */
```

```
int extend_pos;

void state_extend_init(const state_t &s) {
    action_cur = ACTION_BEGIN;
    extend_pos = // 根据 s 进行计算
}

bool state_extend(const state_t &s, state_t &next) {
    // extract values from s
    value1 = ...
    value2 = ...
    while(action_cur < ACTION_END) {
        // apply action_cur to values
        value1 = ...
        value2 = ...
        next.count = s.count + 1;
        if (values are valid) {
            // assign new values to next's data fields
            next.data1 = value1
            next.data2 = value2

            if (!hashset_find(next)) { /* 判重 */
                next.action = action_cur;
                next.index = path_index++;
                next.father = state_index(s);
                /* 记录路径 */
                nodes[state_index(next)] = next;
                action_cur++;  /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {..., 0};

bool state_is_target(const state_t &s) {
    ...
    /* 例如 return memcmp(s.data, END.data, xx * sizeof(yy) == 0; */
    /* 例如 return s.data == END.data; */
}
```

bfs_template2.cpp

10.7 Word Ladder

描述

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

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

For example, Given:

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

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

Note:

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

分析

代码

```
//LeetCode, Word Ladder
// 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
    public:
    int ladderLength(const string& start, const string &end,
    const unordered_set<string> &dict) {
        queue<string> current, next;    // 当前层，下一层
        unordered_set<string> visited;  // 判重

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

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

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

                    swap(c, new_word[i]);

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

            return result;
        };

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

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

```
        }
    };
```

相关题目

- Word Ladder II，见 §??

10.8 Word Ladder II

描述

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

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

For example, Given:

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

Return

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

Note:

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

分析

跟 Word Ladder 比，这题是求路径本身，不是路径长度，也是 BFS，略微麻烦点。
这题跟普通的广搜有很大的不同，就是要输出所有路径，因此在记录前驱和判重地方与普通广搜略有不同。

代码

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

        bool found = false;

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

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

                    swap(c, new_word[i]);
```

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

            return result;
        };

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

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

相关题目

- Word Ladder，见 §??

10.9 Surrounded Regions

描述

Given a 2D board containing `'X'` and `'O'`, capture all regions surrounded by `'X'`.

A region is captured by flipping all 'O's into 'X's in that surrounded region .

For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

分析

广搜。从上下左右四个边界往里走，凡是能碰到的'O'，都是跟边界接壤的，应该保留。

代码

```
// LeetCode, Surrounded Regions
// BFS，时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    void solve(vector<vector<char>> &board) {
        if (board.empty()) return;

        const int m = board.size();
        const int n = board[0].size();
        for (int i = 0; i < n; i++) {
            bfs(board, 0, i);
            bfs(board, m - 1, i);
        }
        for (int j = 1; j < m - 1; j++) {
            bfs(board, j, 0);
            bfs(board, j, n - 1);
        }
        for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
        if (board[i][j] == 'O')
        board[i][j] = 'X';
        else if (board[i][j] == '+')
        board[i][j] = 'O';
    }
    private:
    void bfs(vector<vector<char>> &board, int i, int j) {
        typedef pair<int, int> state_t;
        queue<state_t> q;
        const int m = board.size();
        const int n = board[0].size();

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

        auto state_extend = [&](const state_t &s) {
            vector<state_t> result;
            const int x = s.first;
            const int y = s.second;
            // 上下左右
```

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

                return result;
            };

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

相关题目

- 无

10.10 小结

10.10.1 适用场景

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

10.10.2 思考的步骤

1. 是求路径长度，还是路径本身（或动作序列）?
   (a) 如果是求路径长度，则状态里面要存路径长度（或双队列 + 一个全局变量）
   (b) 如果是求路径本身或动作序列
      i. 要用一棵树存储宽搜过程中的路径
      ii. 是否可以预估状态个数的上限？能够预估状态总数，则开一个大数组，用树的双亲表示法；如果不能预估状态总数，则要使用一棵通用的树。这一步也是第 4 步的必要不充分条件。
2. 如何表示状态？即一个状态需要存储哪些些必要的数据，才能够完整提供如何扩展到下一步状态的所有信息。一般记录当前位置或整体局面。
3. 如何扩展状态？这一步跟第 2 步相关。状态里记录的数据不同，扩展方法就不同。对于固定不变的数据结构（一般题目直接给出，作为输入数据），如二叉树，图等，扩展方法很简单，直接往下一层走，对于隐式图，要先在第 1 步里想清楚状态所带的数据，想清楚了这点，那如何扩展就很简单了。
4. 关于判重，状态是否存在完美哈希方案？即将状态一一映射到整数，互相之间不会冲突。

(a) 如果不存在，则需要使用通用的哈希表（自己实现或用标准库，例如 `unordered_set`）来判重；自己实现哈希表的话，如果能够预估状态个数的上限，则可以开两个数组，head 和 next，表示哈希表，参考第 §10.2节方案 2。

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

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

10.10.3 代码模板

广搜需要一个队列，用于一层一层扩展，一个 hashset，用于判重，一棵树（只求长度时不需要），用于存储整棵树。对于队列，可以用 `queue`，也可以把 `vector` 当做队列使用。当求长度时，有两种做法：

1. 只用一个队列，但在状态结构体 `state_t` 里增加一个整数字段 `step`，表示走到当前状态用了多少步，当碰到目标状态，直接输出 `step` 即可。这个方案，可以很方便的变成 A* 算法，把队列换成优先队列即可。
2. 用两个队列，`current, next`，分别表示当前层次和下一层，另设一个全局整数 `level`，表示层数（也即路径长度），当碰到目标状态，输出 `level` 即可。这个方案，状态可以少一个字段，节省内存。

对于 hashset，如果有完美哈希方案，用布尔数组 (`bool visited[STATE_MAX]` 或 `vector<bool> visited(STATE_MAX, false)`) 来表示；如果没有，可以用 STL 里的 `set` 或 `unordered_set`。

对于树，如果用 STL，可以用 `unordered_map<state_t, state_t > father` 表示一颗树，代码非常简洁。如果能够预估状态总数的上限（设为 STATE_MAX），可以用数组 (`state_t nodes[STATE_MAX]`)，即树的双亲表示法来表示树，效率更高，当然，需要写更多代码。

双队列的写法

bfs_template1.cpp

```
/** 状态 */
struct state_t {
    int data1;  /** 状态的数据，可以有多个字段. */
    int data2;  /** 状态的数据，可以有多个字段. */
    // dataN;   /** 其他字段 */
    int action; /** 由父状态移动到本状态的动作，求动作序列时需要. */
    int count;  /** 所花费的步骤数（也即路径长度-1），求路径长度时需要;
    不过，采用双队列时不需要本字段，只需全局设一个整数 */
    bool operator==(const state_t &other) const {
        return true;  // 根据具体问题实现
    }
};

// 定义 hash 函数

// 方法 1: 模板特化，当 hash 函数只需要状态本身，不需要其他数据时，用这个方法比较简洁
namespace std {
    template<> struct hash<state_t> {
        size_t operator()(const state_t & x) const {
            return 0; // 根据具体问题实现
        }
    };
}

// 方法 2: 函数对象，如果 hash 函数需要运行时数据，则用这种方法
class Hasher {
    public:
    Hasher(int _m) : m(_m) {};
    size_t operator()(const state_t &s) const {
        return 0; // 根据具体问题实现
    }
```

```
    private:
    int m; // 存放外面传入的数据
};


/**
* @brief 反向生成路径.
* @param[in] father 树
* @param[in] target 目标节点
* @return 从起点到 target 的路径
*/
template<typename state_t>
vector<state_t> gen_path(const unordered_map<state_t, state_t> &father,
const state_t &target) {
    vector<state_t> path;
    path.push_back(target);

    for (state_t cur = target; father.find(cur) != father.end();
    cur = father.at(cur))
    path.push_back(cur);

    reverse(path.begin(), path.end());

    return path;
}

/**
* @brief 广搜.
* @param[in] state_t 状态，如整数，字符串，一维数组等
* @param[in] start 起点
* @param[in] grid 输入数据
* @return 从起点到目标状态的一条最短路径
*/
template<typename state_t>
vector<state_t> bfs(const state_t &start, const vector<vector<int>> &grid) {
    queue<state_t> next, current; // 当前层，下一层
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father; // 树

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

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

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

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

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

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

bfs_template1.cpp

只用一个队列的写法

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

bfs_template2.cpp

```
// 与模板 1 相同的部分，不再重复
// ...

/**
* @brief 广搜.
* @param[in] state_t 状态，如整数，字符串，一维数组等
* @param[in] start 起点
* @param[in] grid 输入数据
* @return 从起点到目标状态的一条最短路径
*/
template<typename state_t>
vector<state_t> bfs(state_t &start, const vector<vector<int>> &grid) {
    queue<state_t> q; // 队列
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father; // 树

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

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

    start.count = 0;
    q.push(start);
    visited.insert(start);
```

```
        while (!q.empty() && !found) {
            const state_t state = q.front();
            q.pop();
            vector<state_t> new_states = state_extend(state);
            for (auto iter = new_states.cbegin();
            iter != new_states.cend() && ! found; ++iter) {
                const state_t new_state(*iter);

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

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

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

bfs_template2.cpp

第 11 章
深度优先搜索

深度优先搜素按照深度优先顺序遍历状态空间。当搜索到状态空间中的某个节点时，算法会尝试扩展该节点的某个相邻节点。当没有其他节点可以扩展时，算法会转到该节点的父节点，然后搜索该节点的其他相邻节点。也称回溯搜索。

11.1 四色问题

描述

给定 $N(N \leq 8)$ 个点的地图，以及地图上各点的相邻关系，请输出用 4 种颜色将地图涂色的所有方案数（要求相邻两点不能涂成相同的颜色）。

数据中 0 代表不相邻，1 代表相邻。

输入

第一行一个整数 N，代表地图上有 N 个点。

接下来 N 行，每行 N 个整数，每个整数是 0 或者 1。第 i 行第 j 列的值代表了第 i 个点和第 j 个点之间是相邻的还是不相邻，相邻就是 1，不相邻就是 0。我们保证 a[i][j] = a[j][i]。

输出

染色的方案数

样例输入

```
8
0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 1
0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
```

样例输出

```
15552
```

分析

这是一道经典的题目。深搜。

代码

four_colors.c

```
/* wikioi 1116 四色问题    , http://www.wikioi.com/problem/1116/ */
#include <stdio.h>
#include <string.h>
```

```c
#define MAXN 8

int N;
int g[MAXN][MAXN];

/* 记录每个点的颜色，四种颜色用 1234 表示，0 表示未染色. */
int history[MAXN];
int count; /* 方案个数 */

/**
 * 深搜，给第 i 个节点涂色.
 * @param i 第 i 个地点
 * @return 无
 */
void dfs(int i) {
    int j, c;
    if (i == N) {
        count++;
        return;
    }

    for (c = 1; c < 5; c++) {
        int ok = 1;
        for (j = 0; j < i; j++) {
            if (g[i][j] && c == history[j])
                ok = 0; /* 相邻且同色 */
        }
        if (ok) {
            history[i] = c;
            dfs(i + 1);
        }
    }
}

int main() {
    int i, j;

    scanf("%d", &N);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            scanf("%d", &g[i][j]);
        }
    }

    dfs(0);
    printf("%d\n", count);
    return 0;
}
```

four_colors.c

相关的题目

与本题相同的题目：

- wikioi 1116 四色问题, http://www.wikioi.com/problem/1116/

与本题相似的题目：

- None

11.2 全排列

描述

给出一个正整数 n, 请输出 n 的所有全排列

输入

一个整数 $n(1 \leq n \leq 10)$

输出

一共 $n!$ 行，每行 n 个用空格隔开的数，表示 n 的一个全排列。并且按全排列的字典序输出。

样例输入

```
3
```

样例输出

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

分析

这也是一道短小精悍的经典题目。深搜。从代码上看，与上一题的思路几乎一摸一样。

代码

all_permutations.c

```
/* wikioi 1294 全排列    , http://www.wikioi.com/problem/1294/ */
#include <stdio.h>
#include <string.h>

#define MAXN 10

int N;

int history[MAXN];
int count;

void dfs(int i) {
    int j, k;
    if (i == N) {
        count++;
        for (j = 0; j < N; j++) {
            printf("%d ", history[j]);
        }
        printf("\n");
        return;
    }

    for (k = 1; k <= N; k++) {
        int ok = 1;
        for (j = 0; j < i; j++) {
            if (history[j] == k)
                ok = 0;
        }
        if (ok) {
            history[i] = k;
            dfs(i + 1);
        }
    }
}
```

```
int main() {
    scanf("%d", &N);
    dfs(0);
    return 0;
}
```

all_permutations.c

相关的题目

与本题相同的题目：

- wikioi 1294 全排列, http://www.wikioi.com/problem/1294/

与本题相似的题目：

- None

11.3 八皇后问题

描述

在 8×8 的棋盘上，放置 8 个皇后，使得她们互不攻击，每个皇后的攻击范围是同行、同列和同对角线，要求找出所有解。如图 11-1所示。

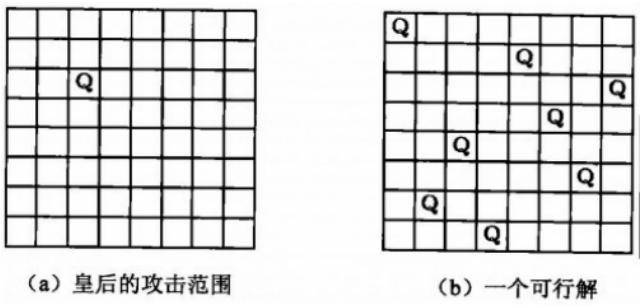


图 11-1 八皇后问题

分析

最简单的暴力枚举方法是，从 64 个格子中选一个子集，使得子集含有 8 个格子，且任意两个格子都不在同一行、同一列或同一个对角线上。这正是子集枚举问题，然而 64 个格子的子集有 2^{64} 个，太大了，这并不是一个很好的模型。

第二个思路是，从 64 个格子中选 8 个格子，这是组合生成问题。根据组合数学，有 $C_{64}^{8} \approx 4.426 \times 10^{9}$ 种方案，比第一种方案优秀，但仍然不够好。

经过思考不难发现，由于每一行只能放一个皇后，那么第一行有 8 种选择，第二行有 7 中选择，…，第 8 行有 1 中选择，总共有 $8! = 40320$ 个方案。如果用 C[x] 表示第 x 行皇后的列编号，则问题变成了一个全排列生成问题，枚举量不会超过 8!。

代码

eight_queen.c

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

#define N 8 // 皇后的个数，也是棋盘的长和宽

int total = 0;  // 可行解的总数
int C[N];  // C[i] 表示第 i 行皇后所在的列编号

/**
 * @brief 输出所有可行的棋局，按列打印.
 *
```

```
 * http://poj.grids.cn/practice/2698/ , 这题需要按列打印
 *
 * @return 无
 */
void output() {
    int i, j;
    printf("No. %d\n", total);
    for (j = 0; j < N; ++j) {
        for (i = 0; i < N; ++i) {
            if (C[i] != j) {
                printf("0 ");
            } else {
                printf("1 ");
            }
        }
        printf("\n");
    }
}

/**
 * @brief 输出所有可行的棋局, 按行打印.
 * @return 无
 */
void output1() {
    int i, j;
    printf("No. %d\n", total);
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            if (j != C[i]) {
                printf("0 ");
            } else {
                printf("1 ");
            }
        }
        printf("\n");
    }
}

/**
 * @brief 检查当前位置 (row, column) 能否放置皇后.
 *
 * @param[in] row 当前行
 * @return 能则返回 1, 不能则返回 0
 */
int check(const int row, const int column) {
    int ok = 1;
    int i;
    for(i = 0; i < row; ++i) {
        // 两个点的坐标为 (row, column), (i, C[i])
        // 检查是否在同一列, 或对角线上
        if(column == C[i] || row - i == column - C[i] ||
            row - i == C[i] - column) {
            ok = 0;
            break;
        }
    }
    return ok;
}

/**
 * @brief 八皇后, 深搜
 *
 * @param[in] row 搜索当前行, 该在哪一列上放一个皇后
 * @return 可行解的个数
```

```
 */
int search(const int row) {
    int j;
    if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
        ++total;
        output();
        return total;
    }

    for (j = 0; j < N; ++j) {  // 一列一列的试
        const int ok = check(row, j);
        if (ok) {  // 如果合法，继续递归
            C[row] = j;
            search(row + 1);
        }
    }

    return total;
}

// 表示已经放置的皇后占据了哪些列
int columns[N];
// 占据了哪些主对角线
int principal_diagonals[2 * N];
// 占据了哪些副对角线
int counter_diagonals[2 * N];

/**
 * @brief 检查当前位置 (row, column) 能否放置皇后.
 *
 * @param[in] row, 当前行
 * @return 能则返回 1，不能则返回 0
 */
int check2(const int row, const int column) {
    return columns[column] == 0 && principal_diagonals[row + column] == 0
        && counter_diagonals[row - column + N] == 0;
}

/**
 * @brief 八皇后，深搜，更优化的版本，用空间换时间
 *
 * @param[in] row 搜索当前行，该在哪一列上放一个皇后
 * @return 可行解的个数
 */
int search2(const int row) {
    int j;
    if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
        ++total;
        output();
        return total;
    }

    for (j = 0; j < N; ++j) {  // 一列一列的试
        const int ok = check2(row, j);
        if (ok) {  // 如果合法，继续递归
            // 执行扩展动作
            C[row] = j;
            columns[j] = principal_diagonals[row + j] =
                    counter_diagonals[row - j + N] = 1;
            search2(row + 1);
            // 撤销动作
            C[row] = -1;  // 这句可以省略，因为 C[row] 会被覆盖掉
            columns[j] = principal_diagonals[row + j] =
                    counter_diagonals[row - j + N] = 0;
```

```
        }
    }

    return total;
}

int main() {
    // search(0);
    search2(0);
    return 0;
}
```

eight_queen.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①] 第 123 页 7.4.1 节
- 百练 2698 八皇后问题, http://poj.grids.cn/practice/2698/
- wikioi 1295 N 皇后问题, http://www.wikioi.com/problem/1295/

与本题相似的题目：

- POJ 1321 棋盘问题, http://poj.org/problem?id=1321

11.4 还原 IP 地址

描述

本题是 LeetCode Online Judge 上的”Restore IP Addresses”。
给定一个只包含数字的字符串，还原出所有合法的 IP 地址。
例如：给定”25525511135”，返回 [”255.255.11.135”, ”255.255.111.35”]。(顺序无关紧要)

分析

这题很明显分为四步，有层次，因此可以尝试用回溯法解决。

代码

restore_ip_adresses.cpp

```
// LeetCode, Restore IP Addresses
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        string ip; // 存放中间结果
        dfs(s, 0, 0, ip, result);
        return result;
    }

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

[①] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

```
            vector<string> &result) {
        if (s.size() - start > (4 - step) * 3)
            return;  // 非法结果，剪枝
        if (s.size() - start < (4 - step))
            return;  // 非法结果，剪枝

        if (start == s.size() && step == 4) {  // 找到一个合法解
            ip.resize(ip.size() - 1);
            result.push_back(ip);
            return;
        }

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

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

restore_ip_adresses.cpp

11.5 Combination Sum

描述

本题是 LeetCode Online Judge 上的"Combination Sum"。

给定一个数的集合 (C) 和一个目标数 (T)，找到 C 中所有不重复的组合，让这些被选出来的数加起来等于 T。

每一个数可以被选无数次。

注意:

- 所有的数（包括目标）都是正整数
- 一个组合（$a_1, a_2, \cdot, a_k$）中的元素必须以非递减顺序排列
- 一个组合不能与另一个组合重复

例如，给定一组数 2,3,6,7, 和目标 7，则答案是

```
[7]
[2, 2, 3]
```

分析

这题没有固定的步骤数，但是步骤也是有限的，因此可以尝试用回溯法。

代码

combination_sum.cpp

```
// LeetCode, Combination Sum
class Solution {
public:
    vector<vector<int> > combinationSum(vector<int> &nums, int target) {
        sort(nums.begin(), nums.end());
        vector<vector<int> > result; // 最终结果
        vector<int> intermediate; // 中间结果
        dfs(nums, target, 0, intermediate, result);
        return result;
    }
```

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

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

combination_sum.cpp

11.6 Combination Sum II

描述

本题是 LeetCode Online Judge 上的"Combination Sum II"。

本题与上一题唯一不同的是，每个数只能使用一次。

分析

这题没有固定的步骤数，但是步骤也是有限的，因此可以尝试用回溯法。

代码

combination_sum2.cpp

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

        int previous = -1;
        for (size_t i = index; i < nums.size(); i++) {
            // 如果上一轮循环没有选 nums[i]，则本次循环就不能再选 nums[i]，
            // 确保 nums[i] 最多只用一次
            if (previous == nums[i]) continue;

            if (gap < nums[i]) return;  // 剪枝

            previous = nums[i];
```

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

combination_sum2.cpp

11.7 Palindrome Partitioning

描述

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

Return all possible palindrome partitioning of s.

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

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

分析

在每一步都可以判断中间结果是否为合法结果，用回溯法。
一个长度为 n 的字符串，有 $n-1$ 个地方可以砍断，每个地方可断可不断，因此复杂度为 $O(2^{n-1})$

深搜 1

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

    // s[0, prev-1] 之间已经处理，保证是回文串
    // prev 表示 s[prev-1] 与 s[prev] 之间的空隙位置，start 同理
    void dfs(string &s, vector<string>& path,
    vector<vector<string>> &result, size_t prev, size_t start) {
        if (start == s.size()) { // 最后一个隔板
            if (isPalindrome(s, prev, start - 1)) { // 必须使用
                path.push_back(s.substr(prev, start - prev));
                result.push_back(path);
                path.pop_back();
            }
            return;
        }
        // 不断开
        dfs(s, path, result, prev, start + 1);
        // 如果 [prev, start-1] 是回文，则可以断开，也可以不断开（上一行已经做了）
        if (isPalindrome(s, prev, start - 1)) {
            // 断开
            path.push_back(s.substr(prev, start - prev));
            dfs(s, path, result, start, start + 1);
            path.pop_back();
        }
    }
```

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

深搜 2

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

```
//LeetCode, Palindrome Partitioning
// 时间复杂度 O(2^n)，空间复杂度 O(n)
class Solution {
    public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 一个 partition 方案
        DFS(s, path, result, 0);
        return result;
    }
    // 搜索必须以 s[start] 开头的 partition 方案
    void DFS(string &s, vector<string>& path,
    vector<vector<string>> &result, int start) {
        if (start == s.size()) {
            result.push_back(path);
            return;
        }
        for (int i = start; i < s.size(); i++) {
            if (isPalindrome(s, start, i)) { // 从 i 位置砍一刀
                path.push_back(s.substr(start, i - start + 1));
                DFS(s, path, result, i + 1);  // 继续往下砍
                path.pop_back(); // 撤销上上行
            }
        }
    }
    bool isPalindrome(const string &s, int start, int end) {
        while (start < end && s[start] == s[end]) {
            ++start;
            --end;
        }
        return start >= end;
    }
};
```

动规

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

        vector<vector<string> > sub_palins[n]; // sub palindromes of s[0,i]
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i; j < n; ++j)
```

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

相关题目

- Palindrome Partitioning II，见 §??

11.8 Unique Paths

描述

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

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

How many possible unique paths are there?

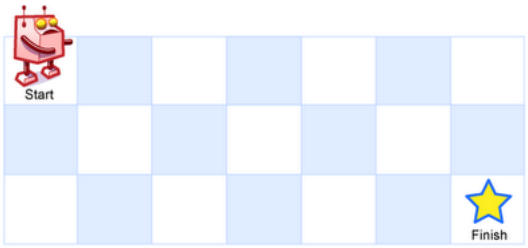


图 11-2 Above is a 3×7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

11.8.1 深搜

深搜，小集合可以过，大集合会超时

代码

```
// LeetCode, Unique Paths
// 深搜，小集合可以过，大集合会超时
// 时间复杂度 O(n^4)，空间复杂度 O(n)
class Solution {
    public:
    int uniquePaths(int m, int n) {
        if (m < 1 || n < 1) return 0; // 终止条件

        if (m == 1 && n == 1) return 1; // 收敛条件

        return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
    }
};
```

11.8.2 备忘录法

给前面的深搜，加个缓存，就可以过大集合了。即备忘录法。

代码

```
// LeetCode, Unique Paths
// 深搜 + 缓存，即备忘录法
// 时间复杂度 O(n^2)，空间复杂度 O(n^2)
class Solution {
    public:
    int uniquePaths(int m, int n) {
        // 0 行和 0 列未使用
        this->f = vector<vector<int> >(m + 1, vector<int>(n + 1, 0));
        return dfs(m, n);
    }
    private:
    vector<vector<int> > f;  // 缓存

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

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

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

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

11.8.3 动规

既然可以用备忘录法自顶向下解决，也一定可以用动规自底向上解决。

设状态为 `f[i][j]`，表示从起点 $(1,1)$ 到达 (i,j) 的路线条数，则状态转移方程为：

```
f[i][j]=f[i-1][j]+f[i][j-1]
```

代码

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

11.8.4 数学公式

一个 m 行，n 列的矩阵，机器人从左上走到右下总共需要的步数是 $m+n-2$，其中向下走的步数是 $m-1$，因此问题变成了在 $m+n-2$ 个操作中，选择 m–1 个时间点向下走，选择方式有多少种。即 C_{m+n-2}^{m-1} 。

代码

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

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

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

相关题目

- Unique Paths II，见 §??
- Minimum Path Sum, 见 §??

11.9 Unique Paths II

描述

Follow up for ”Unique Paths”:

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3×3 grid as illustrated below.

```
[
[0,0,0],
[0,1,0],
[0,0,0]
]
```

The total number of unique paths is 2.

Note: m and n will be at most 100.

11.9.1 备忘录法

在上一题的基础上改一下即可。相比动规，简单得多。

代码

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

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

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

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

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

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

11.9.2 动规

与上一题类似，但要特别注意第一列的障碍。在上一题中，第一列全部是 1，但是在这一题中不同，第一列如果某一行有障碍物，那么后面的行应该为 0。

代码

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

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

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

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

相关题目

- Unique Paths，见 §??
- Minimum Path Sum, 见 §??

11.10 N-Queens

描述

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

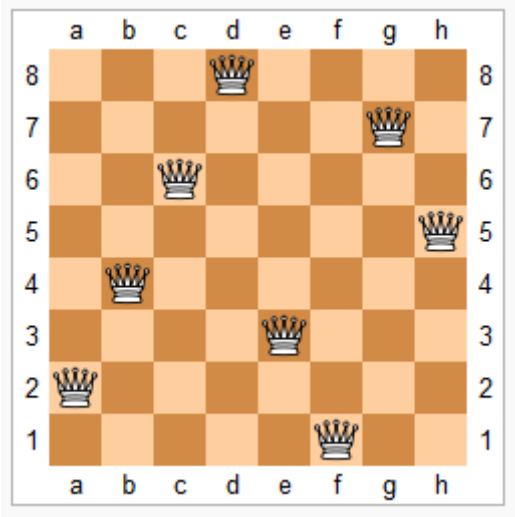


图 11-3 Eight Queens

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example, There exist two distinct solutions to the 4-queens puzzle:

```
[
[".Q..",  // Solution 1
"...Q",
"Q...",
"..Q."],

["..Q.",  // Solution 2
"Q...",
"...Q",
".Q.."]
]
```

分析

经典的深搜题。

代码

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

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

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

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            const bool ok = columns[j] == 0 && main_diag[row + j] == 0 &&
            anti_diag[row - j + N] == 0;
            if (! ok) continue;  // 剪枝：如果合法，继续递归
            // 执行扩展动作
            C[row] = j;
            columns[j] = main_diag[row + j] = anti_diag[row - j + N] = 1;
            dfs(C, result, row + 1);
            // 撤销动作
            // C[row] = 0;
            columns[j] = main_diag[row + j] = anti_diag[row - j + N] = 0;
        }
    }
};
```

相关题目

- N-Queens II，见 §??

11.11 N-Queens II

描述

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

分析

只需要输出解的个数，不需要输出所有解，代码要比上一题简化很多。设一个全局计数器，每找到一个解就增 1。

代码

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

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

    void dfs(vector<int> &C, int row) {
        const int N = C.size();
        if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            ++this->count;
            return;
        }

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            const bool ok = columns[j] == 0 &&
            main_diag[row + j] == 0 &&
            anti_diag[row - j + N] == 0;
            if (!ok) continue;  // 剪枝：如果合法，继续递归
            // 执行扩展动作
            C[row] = j;
            columns[j] = main_diag[row + j] =
            anti_diag[row - j + N] = 1;
            dfs(C, row + 1);
            // 撤销动作
            // C[row] = 0;
            columns[j] = main_diag[row + j] =
            anti_diag[row - j + N] = 0;
        }
    }
};
```

相关题目

- N-Queens，见 §??

11.12 Restore IP Addresses

描述

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example: Given "25525511135",
return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

分析

必须要走到底部才能判断解是否合法，深搜。

代码

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

    /**
    * @brief 解析字符串
    * @param[in] s 字符串，输入数据
    * @param[in] startIndex 从 s 的哪里开始
    * @param[in] step 当前步骤编号，从 0 开始编号，取值为 0,1,2,3,4 表示结束了
    * @param[out] intermediate 当前解析出来的中间结果
    * @param[out] result 存放所有可能的 IP 地址
    * @return 无
    */
    void dfs(string s, size_t start, size_t step, string ip,
    vector<string> &result) {
        if (start == s.size() && step == 4) {  // 找到一个合法解
            ip.resize(ip.size() - 1);
            result.push_back(ip);
            return;
        }

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

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

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

相关题目

- 无

11.13 Combination Sum

描述

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination $(a_1, a_2, ..., a_k)$ must be in non-descending order. (ie, $a_1 \leq a_2 \leq ... \leq a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `2,3,6,7` and target `7`, A solution set is:

```
[7]
[2, 2, 3]
```

分析

无

代码

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

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

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

相关题目

- Combination Sum II，见 §??

11.14 Combination Sum II

描述

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C once number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination $(a_1, a_2, ..., a_k)$ must be in non-descending order. (ie, $a_1 > a_2 > ... > a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `10,1,2,7,6,1,5` and target `8`, A solution set is:

```
[1, 7]
[1, 2, 5]
[2, 6]
[1, 1, 6]
```

分析

无

代码

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

        int previous = -1;
        for (size_t i = start; i < nums.size(); i++) {
            // 如果上一轮循环没有选 nums[i]，则本次循环就不能再选 nums[i]，
            // 确保 nums[i] 最多只用一次
            if (previous == nums[i]) continue;

            if (gap < nums[i]) return;  // 剪枝

            previous = nums[i];

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

相关题目

- Combination Sum ， 见 §??

11.15 Generate Parentheses

描述

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given $n = 3$, a solution set is:

```
"((()))", "(()())", "(())()", "()(())", "()()()"
```

分析

小括号串是一个递归结构，跟单链表、二叉树等递归结构一样，首先想到用递归。

一步步构造字符串。当左括号出现次数 $< n$ 时，就可以放置新的左括号。当右括号出现次数小于左括号出现次数时，就可以放置新的右括号。

代码 1

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

代码 2

另一种递归写法，更加简洁。

```
// LeetCode, Generate Parentheses
// @author 连城 (http://weibo.com/lianchengzju)
class Solution {
    public:
    vector<string> generateParenthesis (int n) {
        if (n == 0) return vector<string>(1, "");
        if (n == 1) return vector<string> (1, "()");
        vector<string> result;

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

        return result;
    }
};
```

相关题目

- Valid Parentheses, 见 §??
- Longest Valid Parentheses, 见 §??

11.16 Sudoku Solver

描述

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character `'.'`.

You may assume that there will be only one unique solution.

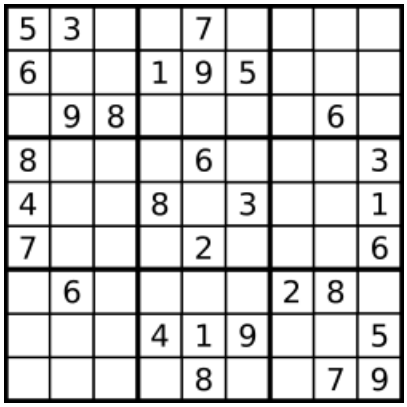


图 11-4 A sudoku puzzle...

图 11-5 ...and its solution numbers marked in red

分析

无。

代码

```
// LeetCode, Sudoku Solver
// 时间复杂度 O(9^4), 空间复杂度 O(1)
class Solution {
    public:
    bool solveSudoku(vector<vector<char> > &board) {
        for (int i = 0; i < 9; ++i)
        for (int j = 0; j < 9; ++j) {
            if (board[i][j] == '.') {
                for (int k = 0; k < 9; ++k) {
```

```
                    board[i][j] = '1' + k;
                    if (isValid(board, i, j) && solveSudoku(board))
                    return true;
                    board[i][j] = '.';
                }
                return false;
            }
        }
        return true;
    }
    private:
    // 检查 (x, y) 是否合法
    bool isValid(const vector<vector<char> > &board, int x, int y) {
        int i, j;
        for (i = 0; i < 9; i++) // 检查 y 列
        if (i != x && board[i][y] == board[x][y])
        return false;
        for (j = 0; j < 9; j++) // 检查 x 行
        if (j != y && board[x][j] == board[x][y])
        return false;
        for (i = 3 * (x / 3); i < 3 * (x / 3 + 1); i++)
        for (j = 3 * (y / 3); j < 3 * (y / 3 + 1); j++)
        if ((i != x || j != y) && board[i][j] == board[x][y])
        return false;
        return true;
    }
};
```

相关题目

- Valid Sudoku, 见 §2.1.15

11.17 Word Search

描述

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where `"adjacent"` cells are those horizontally or vertically neighbouring. The same letter cell may not be used more than once.

For example, Given board =

```
[
["ABCE"],
["SFCS"],
["ADEE"]
]
```

word = `"ABCCED"`, -> returns `true`,

word = `"SEE"`, -> returns `true`,

word = `"ABCB"`, -> returns `false`.

分析

无。

代码

```
// LeetCode, Word Search
// 深搜，递归
// 时间复杂度 O(n^2*m^2)，空间复杂度 O(n^2)
class Solution {
```

```
    public:
    bool exist(vector<vector<char> > &board, string word) {
        const int m = board.size();
        const int n = board[0].size();
        vector<vector<bool> > visited(m, vector<bool>(n, false));
        for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
        if (dfs(board, word, 0, i, j, visited))
        return true;
        return false;
    }
    private:
    static bool dfs(const vector<vector<char> > &board, const string &word,
    int index, int x, int y, vector<vector<bool> > &visited) {
        if (index == word.size())
        return true; // 收敛条件

        if (x < 0 || y < 0 || x >= board.size() || y >= board[0].size())
        return false;  // 越界，终止条件

        if (visited[x][y]) return false; // 已经访问过，剪枝

        if (board[x][y] != word[index]) return false; // 不相等，剪枝

        visited[x][y] = true;
        bool ret = dfs(board, word, index + 1, x - 1, y, visited) || // 上
        dfs(board, word, index + 1, x + 1, y, visited)    || // 下
        dfs(board, word, index + 1, x, y - 1, visited)    || // 左
        dfs(board, word, index + 1, x, y + 1, visited);      // 右
        visited[x][y] = false;
        return ret;
    }
};
```

相关题目

- 无

11.18 小结

11.18.1 适用场景

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

状态转换图：树或者图。

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

11.18.2 思考的步骤

1. 是求路径条数，还是路径本身（或动作序列）？深搜最常见的三个问题，求可行解的总数，求一个可行解，求所有可行解。
   (a) 如果是路径条数，则不需要存储路径。
   (b) 如果是求路径本身，则要用一个数组 `path[]` 存储路径。跟宽搜不同，宽搜虽然最终求的也是一条路径，但是需要存储扩展过程中的所有路径，在没找到答案之前所有路径都不能放弃；而深搜，在搜索过程中始终只有一条路径，因此用一个数组就足够了。

2. 只要求一个解，还是要求所有解？如果只要求一个解，那找到一个就可以返回；如果要求所有解，找到了一个后，还要继续扩展，直到遍历完。广搜一般只要求一个解，因而不需要考虑这个问题（广搜当然也可以求所有解，这时需要扩展到所有叶子节点，相当于在内存中存储整个状态转换图，非常占内存，因此广搜不适合解这类问题）。
3. 如何表示状态？即一个状态需要存储哪些些必要的数据，才能够完整提供如何扩展到下一步状态的所有信息。跟广搜不同，深搜的惯用写法，不是把数据记录在状态 `struct` 里，而是添加函数参数（有时为了节省递归堆栈，用全局变量），`struct` 里的字段与函数参数一一对应。
4. 如何扩展状态？这一步跟上一步相关。状态里记录的数据不同，扩展方法就不同。对于固定不变的数据结构（一般题目直接给出，作为输入数据），如二叉树，图等，扩展方法很简单，直接往下一层走，对于隐式图，要先在第 1 步里想清楚状态所带的数据，想清楚了这点，那如何扩展就很简单了。
5. 关于判重
   (a) 是否需要判重？如果状态转换图是一棵树，则不需要判重，因为在遍历过程中不可能重复；如果状态转换图是一个 DAG，则需要判重。这一点跟 BFS 不一样，BFS 的状态转换图总是 DAG，必须要判重。
   (b) 怎样判重？跟广搜相同，见第 §10.6 节。同时，DAG 说明存在重叠子问题，此时可以用缓存加速，见第 8 步。
6. 终止条件是什么？终止条件是指到了不能扩展的末端节点。对于树，是叶子节点，对于图或隐式图，是出度为 0 的节点。
7. 收敛条件是什么？收敛条件是指找到了一个合法解的时刻。如果是正向深搜（父状态处理完了才进行递归，即父状态不依赖子状态，递归语句一定是在最后，尾递归），则是指是否达到目标状态；如果是逆向深搜（处理父状态时需要先知道子状态的结果，此时递归语句不在最后），则是指是否到达初始状态。

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

   为了判断是否到了收敛条件，要在函数接口里用一个参数记录当前的位置（或距离目标还有多远）。如果是求一个解，直接返回这个解；如果是求所有解，要在这里收集解，即把第一步中表示路径的数组 `path[]` 复制到解集合里。
8. 如何加速？
   (a) 剪枝。深搜一定要好好考虑怎么剪枝，成本小收益大，加几行代码，就能大大加速。这里没有通用方法，只能具体问题具体分析，要充分观察，充分利用各种信息来剪枝，在中间节点提前返回。
   (b) 缓存。
      i. 前提条件：状态转换图是一个 DAG。DAG=> 存在重叠子问题 => 子问题的解会被重复利用，用缓存自然会有加速效果。如果依赖关系是树状的（例如树，单链表等），没必要加缓存，因为子问题只会一层层往下，用一次就再也不会用到，加了缓存也没什么加速效果。
      ii. 具体实现：可以用数组或 HashMap。维度简单的，用数组；维度复杂的，用 HashMap，C++ 有 `map`，C++ 11 以后有 `unordered_map`，比 `map` 快。

拿到一个题目，当感觉它适合用深搜解决时，在心里面把上面 8 个问题默默回答一遍，代码基本上就能写出来了。对于树，不需要回答第 5 和第 8 个问题。如果读者对上面的经验总结看不懂或感觉“不实用”，很正常，因为这些经验总结是我做了很多题目后总结出来的，从思维的发展过程看，“经验总结”要晚于感性认识，所以这时候建议读者先做做前面的题目，积累一定的感性认识后，再回过头来看这一节的总结，一定会有共鸣。

11.18.3 代码模板

dfs_template.cpp

```
/**
* dfs 模板.
* @param[in] input 输入数据指针
* @param[out] path 当前路径，也是中间结果
* @param[out] result 存放最终结果
* @param[inout] cur or gap 标记当前位置或距离目标的距离
* @return 路径长度，如果是求路径本身，则不需要返回长度
*/
```

```
void dfs(type &input, type &path, type &result, int cur or gap) {
    if (数据非法) return 0;   // 终止条件
    if (cur == input.size()) { // 收敛条件
        // if (gap == 0) {
            将 path 放入 result
        }

        if (可以剪枝) return;

        for(...) { // 执行所有可能的扩展动作
            执行动作，修改 path
            dfs(input, step + 1 or gap--, result);
            恢复 path
        }
    }
```

dfs_template.cpp

11.18.4 深搜与回溯法的区别

深搜 (Depth-first search, DFS) 的定义见 http://en.wikipedia.org/wiki/Depth_first_search，回溯法 (backtracking) 的定义见 http://en.wikipedia.org/wiki/Backtracking

回溯法 = 深搜 + 剪枝。一般大家用深搜时，或多或少会剪枝，因此深搜与回溯法没有什么不同，可以在它们之间画上一个等号。本书同时使用深搜和回溯法两个术语，但读者可以认为二者等价。

深搜一般用递归 (recursion) 来实现，这样比较简洁。

深搜能够在候选答案生成到一半时，就进行判断，抛弃不满足要求的答案，所以深搜比暴力搜索法要快。

11.18.5 深搜与递归的区别

深搜经常用递归 (recursion) 来实现，二者常常同时出现，导致很多人误以为他俩是一个东西。

深搜，是逻辑意义上的算法，递归，是一种物理意义上的实现，它和迭代 (iteration) 是对应的。深搜，可以用递归来实现，也可以用栈来实现；而递归，一般总是用来实现深搜。可以说，**递归一定是深搜，深搜不一定用递归**。

递归有两种加速策略，一种是**剪枝 (prunning)**，对中间结果进行判断，提前返回；一种是**缓存**，缓存中间结果，防止重复计算，用空间换时间。

其实，递归 + 缓存，就是 memorization。所谓 **memorization**（翻译为备忘录法，见第 §14.1节），就是"top-down with cache"（自顶向下 + 缓存），它是 Donald Michie 在 1968 年创造的术语，表示一种优化技术，在 top-down 形式的程序中，使用缓存来避免重复计算，从而达到加速的目的。

memorization 不一定用递归，就像深搜不一定用递归一样，可以在迭代 (iterative) 中使用 memorization 。**递归也不一定用 memorization**，可以用 memorization 来加速，但不是必须的。只有当递归使用了缓存，它才是 memorization 。

既然递归一定是深搜，为什么很多书籍都同时使用这两个术语呢？在递归味道更浓的地方，一般用递归这个术语，在深搜更浓的场景下，用深搜这个术语，读者心里要弄清楚他俩大部分时候是一回事。在单链表、二叉树等递归数据结构上，递归的味道更浓，这时用递归这个术语；在图、隐式图等数据结构上，深搜的味道更浓，这时用深搜这个术语。

11.19 小结

11.19.1 适用场景

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

状态转换图：树或者图。

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

11.19.2 思考的步骤

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

   (a) 如果是求路径本身，则要用一个数组 `path[]` 存储路径。跟宽搜不同，宽搜虽然最终求的也是一条路径，但是需要存储扩展过程中的所有路径，在没找到答案之前所有路径都不能放弃；而深搜，在搜索过程中始终只有一条路径，因此用一个数组就足够了。

   (b) 如果是路径条数，则不需要存储路径。

2. 只要求一个解，还是要求所有解？如果只要求一个解，那找到一个就可以返回；如果要求所有解，找到了一个后，还要继续扩展，直到遍历完。广搜一般只要求一个解，因而不需要考虑这个问题（广搜当然也可以求所有解，这时需要扩展到所有叶子节点，相当于在内存中存储整个状态转换图，非常占内存，因此广搜不适合解这类问题）。

3. 如何表示状态？即一个状态需要存储哪些些必要的数据，才能够完整提供如何扩展到下一步状态的所有信息。跟广搜不同，深搜的惯用写法，不是把数据记录在状态 `struct` 里，而是添加函数参数（有时为了节省递归堆栈，用全局变量），`struct` 里的字段与函数参数一一对应。

4. 如何扩展状态？这一步跟上一步相关。状态里记录的数据不同，扩展方法就不同。对于固定不变的数据结构（一般题目直接给出，作为输入数据），如二叉树，图等，扩展方法很简单，直接往下一层走，对于隐式图，要先在第 1 步里想清楚状态所带的数据，想清楚了这点，那如何扩展就很简单了。

5. 关于判重

   (a) 如果状态转换图是一棵树，则不需要判重，因为在遍历过程中不可能重复。

   (b) 如果状态转换图是一个图，则需要判重，方法跟广搜相同，见第 §10.6 节。这里跟第 8 步中的加缓存是相同的，如果有重叠子问题，则需要判重，此时加缓存自然也是有效果的。

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

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

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

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

8. 如何加速？

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

   (b) 缓存。如果子问题的解会被重复利用，可以考虑使用缓存。

      i. 前提条件：子问题的解会被重复利用，即子问题之间的依赖关系是有向无环图 (DAG)。如果依赖关系是树状的（例如树，单链表），没必要加缓存，因为子问题只会一层层往下，用一次就再也不会用到，加了缓存也没什么加速效果。

      ii. 具体实现：可以用数组或 HashMap。维度简单的，用数组；维度复杂的，用 HashMap，C++ 有 `map`，C++ 11 以后有 `unordered_map`，比 `map` 快。

拿到一个题目，当感觉它适合用深搜解决时，在心里面把上面 8 个问题默默回答一遍，代码基本上就能写出来了。对于树，不需要回答第 5 和第 8 个问题。如果读者对上面的经验总结看不懂或感觉“不实用”，很正常，因为这些经验总结是笔者做了很多深搜题后总结出来的，从思维的发展过程看，“经验总结”要晚于感性认识，所以这时候建议读者先做做后面的题目，积累一定的感性认识后，在回过头来看这一节的总结，相信会和笔者有共鸣。

11.19.3 代码模板

dfs_template.cpp

```
/**
 * dfs 模板.
 * @param[in] input 输入数据指针
 * @param[inout] cur or gap 标记当前位置或距离目标的距离
 * @param[out] path 当前路径，也是中间结果
 * @param[out] result 存放最终结果
 * @return 路径长度，如果是求路径本身，则不需要返回长度
 */
void dfs(type *input, type *path, int cur or gap, type *result) {
    if (数据非法) return 0;   // 终止条件
    if (cur == input.size( or gap == 0)) { // 收敛条件
        将 path 放入 result
    }

    if (可以剪枝) return;

    for(...) { // 执行所有可能的扩展动作
        执行动作，修改 path
        dfs(input, step + 1 or gap--, result);
        恢复 path
    }
}
```

dfs_template.cpp

11.19.4 深搜与回溯法的区别

深搜 (Depth-first search, DFS) 的定义见 http://en.wikipedia.org/wiki/Depth_first_search，回溯法 (backtracking) 的定义见 http://en.wikipedia.org/wiki/Backtracking

回溯法 = 深搜 + 剪枝。一般大家用深搜时，或多或少会剪枝，因此深搜与回溯法没有什么不同，可以在它们之间画上一个等号。本书同时使用深搜和回溯法两个术语，但读者可以认为二者等价。

深搜一般用递归 (recursion) 来实现，这样比较简洁。

深搜能够在候选答案生成到一半时，就进行判断，抛弃不满足要求的答案，所以深搜比暴力搜索法要快。

11.19.5 深搜与递归的区别

深搜经常用递归 (recursion) 来实现，二者常常同时出现，导致很多人误以为他俩是一个东西。

深搜，是逻辑意义上的算法，递归，是一种物理意义上的实现，它和迭代 (iteration) 是对应的。深搜，可以用递归来实现，也可以用栈来实现；而递归，一般总是用来实现深搜。可以说，**递归一定是深搜，深搜不一定用递归**。

递归有两种加速策略，一种是**剪枝 (prunning)**，对中间结果进行判断，提前返回；一种是**加缓存**（就变成了 memoization，备忘录法），缓存中间结果，防止重复计算，用空间换时间。

其实，递归 + 缓存，就是一种 memorization 。所谓 **memorization**（翻译为备忘录法，见第 §14.1节），就是"top-down with cache"（自顶向下 + 缓存），它是 Donald Michie 在 1968 年创造的术语，表示一种优化技术，在 top-down 形式的程序中，使用缓存来避免重复计算，从而达到加速的目的。

memorization 不一定用递归，就像深搜不一定用递归一样，可以在迭代 (iterative) 中使用 memorization 。**递归也不一定用 memorization**，可以用 memorization 来加速，但不是必须的。只有当递归使用了缓存，它才是 memorization 。

既然递归一定是深搜，为什么很多书籍都同时使用这两个术语呢？在递归味道更浓的地方，一般用递归这个术语，在深搜更浓的场景下，用深搜这个术语，读者心里要弄清楚他俩大部分时候是一回事。在单链表、二叉树等递归数据结构上，递归的味道更浓，这时用递归这个术语；在图、隐士图等数据结构上，递归的比重不大，深搜的意图更浓，这时用深搜这个术语。

第 12 章
分治法

二分查找，快速排序，归并排序，都属于分治法 (Divide and Conquer)。
多是递归调用实现

12.1 棋盘覆盖

描述

在一个 $2^k \times 2^k (1 \leq k \leq 100)$ 的棋盘中，恰有一个方格被黑色覆盖，其他为白色。用黑色的 L 型牌（如图 12-1所示为 4 种 L 型牌），去覆盖棋盘中所有的白色方格，黑色方格不能被覆盖，且任意两个 L 型牌不能重叠（即不重不漏）。求所需 L 型牌的总数。

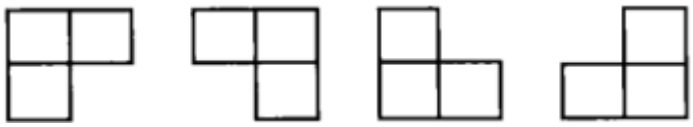


图 12-1　4 种 L 型牌

输入

第一行包含一个整数 T，表示有 T 组测试用例。
每一组测试用例占用一行，包含一个整数 k。

输出

所需 L 型牌的总数

样例输入

```
3
1
2
3
```

样例输出

```
1
5
21
```

分析

本题的棋盘是 $2^k \times 2^k$，很容易想到用分治法。把棋盘切成 4 块，则每一块都是 $2^{k-1} \times 2^{k-1}$ 的。有黑格的那一块可以递归解决，但其他 3 块并没有黑格子，应该怎么办呢？可以构造出一个黑格子，如图 12-2所示，在中心放一个 L 型牌，其它 3 块也变成了子问题。递归边界不难得出，当 $k = 1$ 时 1 块 L 型牌就够了。

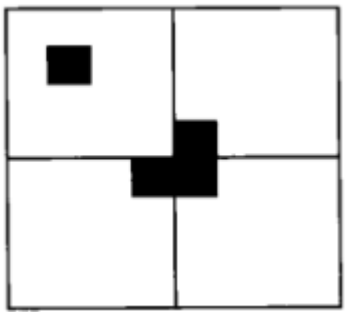


图 12-2 棋盘覆盖问题的递归解法

本题只需要求总数，不需要求具体怎么摆放，因此简化很多。根据上面的思路，设 $f(k)$ 表示棋盘是 $2^k \times 2^k$ 时所需 L 型牌的总数，可得递推公式 $f(k) = 4f(k-1) + 1$。

注意，2^100 是一个很大的数，本题需要处理大数，见 §17.3节。

代码

chessboard_cover.c

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

#define MAXK 100

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (61/MOD_LEN+1)  /* 整数的最大位数, 10^x > 4^100 */

int  d[MAXK][MAX_LEN * 2];  /* d[k-1] = f(k) */

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);  /* 本题输出比较坑爹，最高位数字有前导 0 */
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 计算 f(k) = 4f(k-1)+1，与大整数乘法很类似.
 * @param[in] x x
 * @param[in] y y
 * @param[out] z z=x*y+1
 * @return 无
 */
void bigint_mul_small(const int x[], const int y, int z[]) {
    int i;
    for (i = 0; i < MAX_LEN * 2; i++) z[i] = 0;
```

```
    z[0] = 1;

    for (i = 0; i < MAX_LEN; i++) z[i] += x[i] * y;

    for (i = 0; i < MAX_LEN * 2; i++) {  /* 统一处理进位问题 */
        if (z[i] >= BIGINT_MOD) {  /* 看是否要进位 */
            z[i+1] += z[i] / BIGINT_MOD;  /* 进位 */
            z[i] %= BIGINT_MOD;
        }
    }
}

int main() {
    int k, T;
    d[0][0] = 1;
    for (k = 2; k <= 100; k++) bigint_mul_small(d[k-2], 4, d[k-1]);

    scanf("%d", &T);
    while(T-- > 0) {
        scanf("%d", &k);
        bigint_print(d[k - 1], MAX_LEN * 2);
        printf("\n");
    }
    return 0;
}
```

chessboard_cover.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①]第 148 页 8.3.1 节
- 《计算机算法设计与分析 (第 3 版)》[②]第 19 页 2.6 节
- NYOJ 45 棋盘覆盖, http://acm.nyist.net/JudgeOnline/problem.php?pid=45

与本题相似的题目：

- POJ 2495 Incomplete chess boards, http://poj.org/problem?id=2495

12.2 循环赛日程表

描述

有 2^k 个运动员参加循环比赛，需要设计比赛日程表。要求如下：

- 每个选手必须与其他 n-1 个选手各赛一次
- 每个选手一天只能赛一次
- 比赛一共进行 n-1 天

按此要求设计一张比赛日程表，它有 n 行和 n-1 列，第 i 行第 j 列为第 i 个选手第 j 天遇到的对手。

输入

只有一个数 k，$0 < k < 9$，且 k 为自然数。

输出

一张比赛日程表，它有 n 行和 n-1 列（不算第一列，第一列表示选手的编号），第 i 行第 j 列为第 i 个选手第 j 天遇到的对手。相邻的两个整数用空格隔开。

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009
[②]王晓东, 计算机算法设计与分析 (第 3 版), 电子工业出版社, 2007

样例输入

```
1
```

样例输出

```
1 2
2 1
```

分析

根据分而治之的思想，可从其中一半选手 (2^{k-1} 位) 的比赛日程，推导出全体选手的日程，最终细分到只有两位选手的比赛日程。

图 12-2所示是 k=3 时的一个可行解，它是由 4 块拼起来的。左上角是 k=2 时的一组解，左下角是由左上角每个数加 4 得到，而右上角、右下角分别由左下角、左上角复制得到。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

图 12-3 循环赛日程表问题 k=3 时的解

代码

roundrobin_scheduling.c

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

#define MAXN 512   /* N=2^k, 0<k9 */
short schedule[MAXN][MAXN];

void dc(const int k) {
    int i, j, t;
    int n, n2; /* 当前的 n，即将扩展的 n */

    /* k=1，即两个人时，日程表可以直接写出 */
    n=2;
    schedule[0][0]=1; schedule[0][1]=2;
    schedule[1][0]=2; schedule[1][1]=1;

     /* 迭代处理，依次处理 2^2....2^k 个选手的比赛日程 */
    for(t = 1; t < k; t++, n *= 2) {
        n2 = n * 2;
        //填左下角元素
        for(i = n; i < n2; i++)
            for(j = 0; j < n; j++)
                schedule[i][j] = schedule[i-n][j] + n;

        //将左下角元素抄到右上角
        for(i = 0; i < n; i++)
            for(j = n; j < n2; j++)
                schedule[i][j] = schedule[i+n][j-n];

        //将左上角元素抄到右下角
        for(i = n; i < n2; i++)
            for(j = n;j < n2; j++)
```

```
                schedule[i][j] = schedule[i-n][j-n];
    }
}

/* 另一个版本 */
void dc2(const int k) {
    int i, j, r;
    int n;
    const int N = 1 << k;

    /* 第一列是选手的编号 */
    for(i = 0; i < N; i++) schedule[i][0] = i + 1;
    schedule[0][1] = 1;  /* 当 k=0 时, 只有一个人 */

    for (n = 2; n <= N; n *= 2) {  /* 方块大小, 2, 4, 8 */
        const int half = n / 2;
        for (r = 0; r < N; r += n) { /* 方块所在行 */
            for (i = r; i <= r + half -1; i++) {  /* 左上角小方块的所有行 */
                for (j = 0; j < half; j++) {  /* 左上角小方块的所有行 */
                    /* 右下角 <-- 左上角 */
                    schedule[i + half][j + half] = schedule[i][j];
                    /* 右上角 <-- 左下角 */
                    schedule[i][j + half] = schedule[i + half][j];
                }
            }
        }
    }
}


int main(){
    int k, N;
    int i,j;

    scanf("%d",&k);
    N = 1 << k;

    dc(k);
    // dc2(k);

    // 输出日程表
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) printf("%d ", schedule[i][j]);
        printf("\n");
    }
    return 0;
}
```

roundrobin_scheduling.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①]第 149 页 8.3.2 节
- 《计算机算法设计与分析 (第 3 版)》[②]第 34 页 2.11 节
- NKOJ 1437 校长杯, http://acm.nankai.edu.cn/p1437.html

与本题相似的题目：

- SPOJ 2826 Round-Robin Scheduling, http://www.spoj.com/problems/RRSCHED/
- UVa OJ 678 Schedule of Taiwan Baseball League, http://t.cn/zHJD9TQ

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009
[②]王晓东, 计算机算法设计与分析 (第 3 版), 电子工业出版社, 2007

12.3 Pow(x,n)

描述

Implement pow(x, n).

分析

二分法，$x^n = x^{n/2} \times x^{n/2} \times x^{n\%2}$

代码

```
//LeetCode, Pow(x, n)
// 二分法，$x^n = x^{n/2} * x^{n/2} * x^{n\%2}$
// 时间复杂度 O(logn)，空间复杂度 O(1)
class Solution {
    public:
    double pow(double x, int n) {
        if (n < 0) return 1.0 / power(x, -n);
        else return power(x, n);
    }
    private:
    double power(double x, int n) {
        if (n == 0) return 1;
        double v = power(x, n / 2);
        if (n % 2 == 0) return v * v;
        else return v * v * x;
    }
};
```

相关题目

- Sqrt(x)，见 §??

12.4 Sqrt(x)

描述

Implement int `sqrt(int x)`.

Compute and return the square root of `x`.

分析

二分查找

代码

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

        if (x < 2) return x;

        while(left <= right) {
            const int mid = left + (right - left) / 2;
```

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

相关题目

- Pow(x)，见 §??

第 13 章
贪心法

我们前面见过的一些算法，比如单源最短路径、最小生成树等都属于贪心法 (greedy algorithm)。
如果一个问题具有以下两个要素：

- 最优子结构 (optimal substructure)
- 贪心选择性质 (greedy-choice property)

则可以用贪心法求最优解。

13.1 最优装载

13.2 哈弗曼编码

描述

给定一个英文字符串，使用 0 和 1 对其进行编码，求最优前缀编码，使其所需要的比特数最少。

分析

题目很长，不过就是哈弗曼编码。

代码

poj1521_entropy.cpp

```
// 本题考查哈弗曼编码，但只需要统计哈弗曼编码后的总码长即可，
// 没必要建哈弗曼树得出哈弗曼编码
#include <stdio.h>
#include <string.h>
#include <queue>
#include <functional>

const int LINE_MAX = 256; // 一行最大字符数
const int MAX_ASCII = 128; // ASCII 码最大值

int main_entropy() {
    char    s[LINE_MAX];
    int     count[MAX_ASCII] = {0}; // count[i] 记录 ASCII 码为 i 的字符的出现次数
    int     sum;
    // 小根堆，队列头为最小元素
    priority_queue<int, vector<int>, greater<int> >    pq;

    while (scanf("%s", s) > 0) {
        sum = 0; // 清零
        const int len = strlen(s);

        if (strcmp(s,"END") == 0) {
            break;
        }

        for (int i = 0; i < len; i++) {
            count[s[i]]++;
```

```
        }

        for (int i = 0;i < MAX_ASCII; i++) {
            if (count[i] > 0) {
                pq.push(count[i]);
                count[i] = 0;
            }
        }
        while (pq.size() > 1) {
            const int a = pq.top(); pq.pop();
            const int b = pq.top(); pq.pop();
            sum += a + b;
            pq.push(a + b);
        }
        if (sum == 0) {
            sum = len; // 此时 pq 中只有一个元素
        }

        while (!pq.empty()) { // clear
            pq.pop();
        }
        // 注意精度设置
        printf("%d %d %.1f\n", 8 * len, sum, ((double)8 * len) / sum);
    }
    return 0;
}
```

poj1521_entropy.cpp

相关的题目

与本题相同的题目：

- POJ 1521 Entropy, http://poj.org/problem?id=1521

与本题相似的题目：

- POJ 3253 Fence Repair, http://poj.org/problem?id=3253
- 《算法竞赛入门经典》[①] 第 155 页例题 8-5
- 《Introduction to Algorithms》[②] 第 16.3 节
- 《算法设计与分析 (第 3 版)》[③] 第 109 页 4.4 节

13.3 部分背包问题

13.4 Jump Game

描述

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:

`A = [2,3,1,1,4]`, return true.

`A = [3,2,1,0,4]`, return false.

[①] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

[②] CLRS,Introduction to Algorithms(3rd Edition), 2009

[③] 王晓东, 计算机算法设计与分析 (第 3 版), 2007

分析

由于每层最多可以跳 `A[i]` 步，也可以跳 0 或 1 步，因此如果能到达最高层，则说明每一层都可以到达。有了这个条件，说明可以用贪心法。

思路一：正向，从 0 出发，一层一层网上跳，看最后能不能超过最高层，能超过，说明能到达，否则不能到达。

思路二：逆向，从最高层下楼梯，一层一层下降，看最后能不能下降到第 0 层。

思路三：如果不敢用贪心，可以用动规，设状态为 `f[i]`，表示从第 0 层出发，走到 `A[i]` 时剩余的最大步数，则状态转移方程为：

$$f[i] = max(f[i-1], A[i-1]) - 1, i > 0$$

代码 1

```
// LeetCode, Jump Game
// 思路 1，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    bool canJump(int A[], int n) {
        int reach = 1; // 最右能跳到哪里
        for (int i = 0; i < reach && reach < n; ++i)
        reach = max(reach,  i + 1 + A[i]);
        return reach >= n;
    }
};
```

代码 2

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

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

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

代码 3

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

相关题目

- Jump Game II，见 §??

13.5 Jump Game II

描述

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example: Given array `A = [2,3,1,1,4]`

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

分析

贪心法。

代码 1

```
// LeetCode, Jump Game II
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int jump(int A[], int n) {
        int step = 0; // 最小步数
        int left = 0;
        int right = 0;  // [left, right] 是当前能覆盖的区间
        if (n == 1) return 0;

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

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

代码 2

```
// LeetCode, Jump Game II
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int jump(int A[], int n) {
        int result = 0;
        // the maximum distance that has been reached
        int last = 0;
        // the maximum distance that can be reached by using "ret+1" steps
        int cur = 0;
        for (int i = 0; i < n; ++i) {
            if (i > last) {
                last = cur;
                ++result;
```

```
                }
                cur = max(cur, i + A[i]);
            }

            return result;
        }
    };
```

相关题目

- Jump Game，见 §??

13.6 Best Time to Buy and Sell Stock

描述

Say you have an array for which the i-th element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

分析

贪心法，分别找到价格最低和最高的一天，低进高出，注意最低的一天要在最高的一天之前。

把原始价格序列变成差分序列，本题也可以做是最大 m 子段和，$m = 1$。

代码

```
// LeetCode, Best Time to Buy and Sell Stock
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int maxProfit(vector<int> &prices) {
        if (prices.size() < 2) return 0;
        int profit = 0;  // 差价，也就是利润
        int cur_min = prices[0]; // 当前最小

        for (int i = 1; i < prices.size(); i++) {
            profit = max(profit, prices[i] - cur_min);
            cur_min = min(cur_min, prices[i]);
        }
        return profit;
    }
};
```

相关题目

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

13.7 Best Time to Buy and Sell Stock II

描述

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

分析

贪心法，低进高出，把所有正的价格差价相加起来。

把原始价格序列变成差分序列，本题也可以做是最大 m 子段和，m = 数组长度。

代码

```
// LeetCode, Best Time to Buy and Sell Stock II
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int maxProfit(vector<int> &prices) {
        int sum = 0;
        for (int i = 1; i < prices.size(); i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) sum += diff;
        }
        return sum;
    }
};
```

相关题目

- Best Time to Buy and Sell Stock，见 §??
- Best Time to Buy and Sell Stock III，见 §??

13.8 Longest Substring Without Repeating Characters

描述

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

分析

假设子串里含有重复字符，则父串一定含有重复字符，单个子问题就可以决定父问题，因此可以用贪心法。跟动规不同，动规里，单个子问题只能影响父问题，不足以决定父问题。

从左往右扫描，当遇到重复字母时，以上一个重复字母的 index+1，作为新的搜索起始位置，直到最后一个字母，复杂度是 $O(n)$。如图 ??所示。

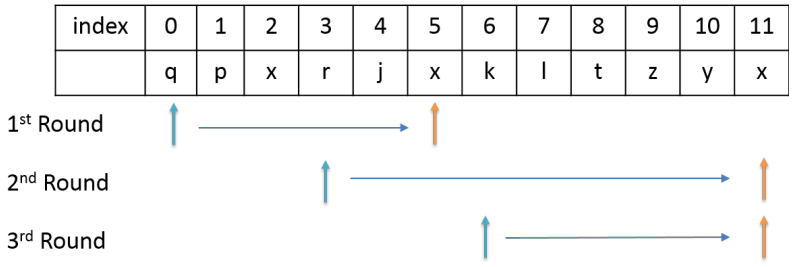


图 13-1 不含重复字符的最长子串

代码

```
// LeetCode, Longest Substring Without Repeating Characters
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
```

```
        int lengthOfLongestSubstring(string s) {
            const int ASCII_MAX = 26;
            int last[ASCII_MAX]; // 记录字符上次出现过的位置
            int start = 0; // 记录当前子串的起始位置

            fill(last, last + ASCII_MAX, -1); // 0 也是有效位置，因此初始化为-1
            int max_len = 0;
            for (int i = 0; i < s.size(); i++) {
                if (last[s[i] - 'a'] >= start) {
                    max_len = max(i - start, max_len);
                    start = last[s[i] - 'a'] + 1;
                }
                last[s[i] - 'a'] = i;
            }
            return max((int)s.size() - start, max_len);  // 别忘了最后一次，例如"abcd"
        }
    };
```

相关题目

- 无

13.9 Container With Most Water

描述

Given n non-negative integers $a_1, a_2, ..., a_n$, where each represents a point at coordinate (i, a_i). n vertical lines are drawn such that the two endpoints of line i is at (i, a_i) and $(i, 0)$. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

分析

每个容器的面积，取决于最短的木板。

代码

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

相关题目

- Trapping Rain Water, 见 §2.1.16
- Largest Rectangle in Histogram, 见 §??

第 14 章
动态规划

【动态规划】

动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

【基本思想与策略】

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

【适用的情况】

动态规划求最优解，适用于具有以下两个要素的优化问题：

- 最优子结构 (optimal substructure)：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。
- 无后续性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。
- 重叠子问题 (overlap subproblem)：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

【求解的基本步骤】

动态规划分为 4 个步骤：

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线 (通常是求最优的活动路线)。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。初始状态 →□ 决策 1 □→□ 决策 2 □→…→□ 决策 n □→ 结束状态

(1) 划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2) 确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3) 确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。

(4) 寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程（包括边界条件）。

实际应用中可以按以下几个简化的步骤进行设计：

（1）分析最优解的性质，并刻画其结构特征。

（2）递归的定义最优解。

（3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

（4）根据计算最优值时得到的信息，构造问题的最优解

【算法实现的说明】

动态规划的主要难点在于理论上的设计，也就是上面 4 个步骤的确定，一旦设计完成，实现部分就会非常简单。

使用动态规划求解问题，最重要的就是确定动态规划三要素：

（1）问题的阶段

（2）每个阶段的状态

（3）从前一个阶段转化到后一个阶段之间的递推关系。

递推关系必须是从次小的问题开始到较大的问题之间的转化，从这个角度来说，动态规划往往可以用递归程序来实现，不过因为递推可以充分利用前面保存的子问题的解来减少重复计算，所以对于大规模问题来说，有递归不可比拟的优势，这也是动态规划算法的核心之处。

确定了动态规划的这三要素，整个求解过程就可以用一个最优决策表来描述，最优决策表是一个二维表，其中行表示决策的阶段，列表示问题状态，表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值（如最短路径，最长公共子序列，最大价值等），填表的过程就是根据递推关系，从 1 行 1 列开始，以行或者列优先的顺序，依次填写表格，最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。

$$f(n,m) = max f(n-1,m), f(n-1,m-w[n]) + P(n,m)$$

写代码实现时有两种方式，“递归 (recursive)+ 自顶向下 (top-down)+ 表格”和“自底向上 (bottom-up)+ 表格”。前者属于一种 memorization （备忘录法），后者才是正宗的动规。

动规用表格将各个子问题的最优解存起来，避免重复计算，是一种空间换时间。

【动规 vs 贪心】

相同点：最优子结构。

不同点：

1 动规的子问题是重叠的，而贪心的子问题是不重叠的 (disjoint subproblems)；

2 动规不具有贪心选择性质；

3 贪心的前进路线是一条线，而动规是一个 DAG。

分治和贪心的相同点：disjoint subproblems。

【动态规划算法基本框架】

```
for(int j=1; j<=m; j=j+1) // 第一个阶段
    xn[j] = 初始值;

for(int i=n-1; i>=1; i=i-1)// 其他 n-1 个阶段
    for(j=1; j>=f(i); j=j+1)//f(i) 与 i 有关的表达式
        xi[j]=j=max（或 min）{g(xi-1[j1:j2]), ......, g(xi-1[jk:jk+1])};

t = g(x1[j1:j2]); // 由子问题的最优解求解整个问题的最优解的方案

print(x1[j1]);

for(int i=2; i<=n-1; i=i+1）{
    t = t-xi-1[ji];

    for(j=1; j>=f(i); j=j+1)
        if(t=xi[ji]) break;
}
```

14.1 动规和备忘录法的区别

动规 (dynamic programming) 一定是自底向上的，备忘录法 (memorization) 一定是自顶向下的。

动规不是 lazy 的，memorization 是 lazy 的，是按需 (on-demand) 计算的。所以，如果所有的子问题至少会碰到一次，则动规有优势；如果有些子问题在搜索过程中不会被碰到（即有剪枝），则 memorization 有优势。更详细的解释请参考 StackOverflow 上的这个帖子 http://t.cn/z80ZS6B 。

备忘录法可以实现跟动规类似的功能，但它不是动规。两者的方向是反的，一个是自顶向下，一个自底向上，我们应该区分开这两个概念。本书后面提到的动规，都是指自底向上的动规。

14.2 动态规划与迭代法

总结下：

1）递归能写出比较清晰简单的代码，但是有比较高的时间复杂度；

2）在递归不满足条件的情况下，动态规划是个比较好的选择；

3）一般来说，独立变量的个数决定动态规划的维度，例如 l1 和 l2 独立变化，所以用了二维动态规划。

14.3 动态规划类型

1. 链（环）式动态规划，如背包问题

2. 区间型动态规划，如石子归并问题：有 n 堆石子，每堆石子有 a[i] 个，每次可以合并相邻的两队石子成为新的一堆，每次合并的代价为两堆石子的个数和，问最少需要多少代价，才能合并到只剩下一堆。

3. 树形动态规划，如树上的最小点覆盖问题

4. 集合动态规划，如多米诺骨牌覆盖

5. 连通性状态压缩动态规划，如回路问题给定一个带权值（可正可负）的棋盘，求权值和最大的连通块。

14.4 最长公共子序列

描述

一个序列的子序列 (subsequence) 是指在该序列中删去若干（可以为 0 个）元素后得到的序列。准确的说，若给定序列 $X = (x_1, x_2, \cdots, x_m)$，则另一个序列 $Z = (z_1, z_2, \cdots, z_k)$，是 X 的子序列是指存在一个严格递增下标序列 $(i_1, i_2, \cdots, i_k)$ 使得对于所有 $j = 1, 2, \cdots, k$ 有 $z_j = x_{i_j}$。例如，序列 $Z = (B, C, D, B)$ 是序列 $X = (A, B, C, B, D, A, B)$ 的子序列，相应的递增下标序列为 $(1, 2, 4, 6)$。

给定两个序列 X 和 Y，求 X 和 Y 的最长公共子序列 (longest common subsequence)。

输入

输入包括多组测试数据，每组数据占一行，包含两个字符串（字符串长度不超过 200），代表两个序列。两个字符串之间由若干个空格隔开。

输出

对每组测试数据，输出最大公共子序列的长度，每组一行。

样例输入

```
abcfbc abfcab
programming contest
abcd mnp
```

样例输出

```
4
2
0
```

分析

最长公共子序列问题具有最优子结构性质。设序列 $X = (x_1, x_2, \cdots, x_m)$ 和 $Y = (y_1, y_2, \cdots, y_n)$ 的最长公共子序列为 $Z = (z_1, z_2, \cdots, z_k)$，则

- 若 $x_m = y_n$，则 $z_k = x_m = y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列。
- 若 $x_m \neq y_n$ 且 $z_k \neq x_m$，则 Z 是 X_{m-1} 和 Y 的最长公共子序列。

- 若 $x_m \neq y_n$ 且 $z_k \neq y_n$，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

其中，$X_{m-1} = (x_1, x_2, \cdots, x_{m-1}), Y_{n-1} = (y_1, y_2, \cdots, y_{n-1}), Z_{k-1} = (z_1, z_2, \cdots, z_{k-1})$。

设状态为 d[i][j]，表示序列 X_i 和 Y_j 的最长公共子序列的长度。由最优子结构可得状态转移方程如下：

$$d[i][j] = \begin{cases} 0 & i = 0, j = 0 \\ d[i-1][j-1] + 1 & i, j > 0; x_i = y_i \\ \max\{d[i][j-1], d[i-1][j]\} & i, j > 0; x_i \neq y_i \end{cases}$$

如果要打印出最长公共子序列，需要另设一个数组 p，p[i][j] 记录 d[i][j] 是由哪个子问题得到的。

代码

lcs.c

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

#define MAX 201  /* 字符串最大长度为 200 */

int d[MAX][MAX]; /* d[i][j] 表示序列 Xi 和 Yj 的最长公共子序列的长度 */
char x[MAX], y[MAX];  /* 字符串末尾有个'0' */

void lcs(const char *x, const int m, const char *y, const int n) {
    int i, j;

    for (i = 0; i <= m; i++) d[i][0] = 0;  /* 边界初始化 */
    for (j = 0; j <= n; j++) d[0][j] = 0;

    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (x[i-1] == y[j-1]) {
                d[i][j] = d[i-1][j-1] + 1;
            } else {
                d[i][j] = d[i-1][j] > d[i][j-1] ? d[i-1][j] : d[i][j-1];
            }
        }
    }
}

void lcs_extend(const char *x, const int m, const char *y, const int n);
void lcs_print(const char *x, const int m, const char *y, const int n);

int main() {
    /* while (scanf ("%s%s", a, b)) { /* TLE */
    /* while (scanf ("%s%s", a, b) == 2) {  /* AC */
    while (scanf ("%s%s", x, y) != EOF) { /* AC */
        const int lx = strlen(x);
        const int ly = strlen(y);
        lcs(x, lx, y, ly);
        printf ("%d\n", d[lx][ly]);
        /*
        lcs_extend(x, lx, y, ly);
        lcs_print(x, lx, y, ly);
        printf("\n"); */
    }
    return 0;
}

int p[MAX][MAX]; /* p[i][j] 记录 d[i][j] 是由哪个子问题得到的 */

void lcs_extend(const char *x, const int m, const char *y, const int n) {
    int i, j;
```

```
    memset(p, 0, sizeof(p));
    for (i = 0; i <= m; i++) d[i][0] = 0;  /* 边界初始化 */
    for (j = 0; j <= n; j++) d[0][j] = 0;

    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (x[i-1] == y[j-1]) {
                d[i][j] = d[i-1][j-1] + 1;
                p[i][j] = 1;
            } else {
                if (d[i-1][j] >= d[i][j-1]) {
                    d[i][j] = d[i-1][j];
                    p[i][j] = 2;
                } else {
                    d[i][j] = d[i][j-1];
                    p[i][j] = 3;
                }
            }
        }
    }
}

void lcs_print(const char *x, const int m, const char *y, const int n) {
    if (m == 0 || n == 0) return;

    if (p[m][n] == 1) {
        lcs_print(x, m - 1, y, n - 1);
        printf("%c", x[m - 1]);
    } else if (p[m][n] == 2) {
        lcs_print(x, m - 1, y, n);
    } else {
        lcs_print(x, m, y, n - 1);
    }
}
```

lcs.c

相关的题目

与本题相同的题目：

- 《计算机算法设计与分析 (第 3 版)》[①]第 56 页 3.3 节
- POJ 1458 Common Subsequence, http://poj.org/problem?id=1458
- HDOJ 1159 Common Subsequence, http://acm.hdu.edu.cn/showproblem.php?pid=1159
- 《程序设计导引及在线实践》[②]第 203 页 10.5 节
- 百练 2806 公共子序列, http://poj.grids.cn/practice/2806/

与本题相似的题目：

- HDOJ 1080 Human Gene Functions, http://acm.hdu.edu.cn/showproblem.php?pid=1080
- HDOJ 1503 Advanced Fruits, http://acm.hdu.edu.cn/showproblem.php?pid=1503

14.5 最大连续子序列和

描述

给定一个整数序列 $S_1, S_2, \cdot, S_n (1 \leq n \leq 1,000,000, -32768 \leq S_i \leq 32768)$，定义函数 $sum(i,j) = S_i + \cdot + S_j (1 \leq i \leq j \leq n)$。

求最大的 $sum(i,j)$，即**最大连续子序列和** (maximum continuous subsequence sum)。

[①]王晓东, 计算机算法设计与分析 (第 3 版), 电子工业出版社, 2007
[②]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

输入

每个测试用例由正整数 n 开头，接着是 n 个整数。

输出

每行输出一个最大和。

样例输入

```
3 1 2 3
6 -1 4 -2 3 -2 3
```

样例输出

```
6
6
```

分析

当我们从头到尾遍历这个数组的时候，对于数组里的一个整数，它有几种选择呢？它只有两种选择：1、加入之前的 SubArray；2. 自己另起一个 SubArray。那什么时候会出现这两种情况呢？

如果之前 SubArray 的总体和大于 0 的话，我们认为其对后续结果是有贡献的。这种情况下我们选择加入之前的 SubArray

如果之前 SubArray 的总体和为 0 或者小于 0 的话，我们认为其对后续结果是没有贡献，甚至是有害的（小于 0 时）。这种情况下我们选择以这个数字开始，另起一个 SubArray。

设状态为 d[j]，表示以 S[j] 结尾的最大连续子序列和，则状态转移方程如下：

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

解释如下：

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

代码

mcss.c

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

#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))

/**
 * @brief 最大连续子序列和，思路四
 * @param[in] S 数列
 * @param[in] n 数组的长度
 * @return 最大连续子序列和
 */
int mcss(int S[], int n) {
    int i, result, cur_min;
    int *sum = (int*) malloc((n + 1) * sizeof(int));  // 前 n 项和

    sum[0] = 0;
    result = INT_MIN;
    cur_min = sum[0];
    for (i = 1; i <= n; i++) { /* 变成前缀和 */
```

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

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

int main() {
    int n, *S;
    while (scanf("%d", &n) > 0) {
        int i;
        S = (int*) malloc(n * sizeof(int));
        for (i = 0; i < n; i++)
            scanf("%d", &S[i]);

        printf("%d\n", mcss_dp(S, n));
        free(S);
    }
    return 0;
}
```

mcss.c

其他解法

- 思路一：直接在 i 到 j 之间暴力枚举，复杂度是 $O(n^3)$
- 思路二：处理后枚举，连续子序列的和等于两个前缀和之差，复杂度 $O(n^2)$。
- 思路三：分治法，把序列分为两段，分别求最大连续子序列和，然后归并，复杂度 $O(n \log n)$
- 思路四：把 $O(n^2)$ 的代码稍作处理，得到 $O(n)$ 的算法
- 思路五：M=1 的最大 M 子段和，见第 §14.7 节

感兴趣的读者请参考这篇博客，http://www.cnblogs.com/gj-Acit/archive/2013/02/12/2910332.html

相关题目

与本题相同的题目：

- LeetCode Maximum Subarray, http://leetcode.com/onlinejudge#question_53

与本题相似的题目：

- HDOJ 1024 Max Sum Plus Plus, http://acm.hdu.edu.cn/showproblem.php?pid=1024

14.6 Minimum Adjustment Cost

描述

Given an integer array, adjust each integers so that the difference of every adjcent integers are not greater than a given number target.

If the array before adjustment is A, the array after adjustment is B, you should minimize the sum of |A[i]-B[i]|.

注意 You can assume each number in the array is a positive integer and not greater than 100

输入

An integer array vector<int> A and a target

输出

minimize the sum of |A[i]-B[i]|

样例输入

```
[1,4,2,3], 1
```

样例输出

```
2
```

分析

dp[i][j]: 把 A[i] 的值修改为 j，所需要的最小花费, 则 dp[i][j] = min{dp[i-1][k] + |j-A[i]|} s.t. |k-j| <= target

代码

MinAdjustmentCost.c

```
#define MAXTARGET 100

int MinAdjustmentCost(vector<int> A, int target) {
    if (A.empty())
    return 0;
    int cur = 0;
    // dp[i][j]: 把 A[i] 的值修改为 j，所需要的最小花费，则 dp[i][j] = min{dp[i-1][k]
        + |j-A[i]|}  s.t. |k-j| <= target
    int dp[2][MAXTARGET + 1];
    memset(dp, 0x00, sizeof(dp));

    for (int i = 0; i < A.size(); i++) {
        int next = cur ^ 1;
        for (int j = 1; j <= MAXTARGET; j++) {
            dp[next][j] = INT_MAX;
            for (int k = max(j-target, 1); k <= min(MAXTARGET, j+target);
            k++)
            dp[next][j] = min(dp[next][j], dp[cur][k] + abs(A[i]-j));
        }
        cur ^= 1;
    }
    int ans = INT_MAX;
    for (int i = 1; i <= MAXTARGET; i++)
    ans = min(ans, dp[cur][i]);

    return ans;
}
```

MinAdjustmentCost.c

相关的题目

与本题相同的题目：

- POJ 3494 Largest Submatrix of All 1's, http://poj.org/problem?id=3494
- LeetCode Maximal Rectangle, http://leetcode.com/onlinejudge#question_85，
  参考代码 https://gist.github.com/soulmachine/b20a15009450016038d9

与本题相似的题目：

- Vijos 1055 奶牛浴场, https://vijos.org/p/1055

14.7 最大 M 子段和

描述

给定一个整数序列 $S_1, S_2, \cdot, S_n (1 \leq n \leq 1,000,000, -32768 \leq S_i \leq 32768)$，定义函数 $sum(i, j) = S_i + ... + S_j (1 \leq i \leq j \leq n)$。

现给定一个正整数 m，找出 m 对 i 和 j，使得 $sum(i_1, j_1) + sum(i_2, j_2) + ... + sum(i_m, j_m)$ 最大。这就是**最大 M 子段和** (maximum m segments sum)。

输入

每个测试用例由两个正整数 m 和 n 开头，接着是 n 个整数。

输出

每行输出一个最大和。

样例输入

```
1 3 1 2 3
2 6 -1 4 -2 3 -2 3
```

样例输出

```
6
8
```

分析

设状态为 d[i,j]，表示前 j 项分为 i 段的最大和，且第 i 段必须包含 S[j]，则状态转移方程如下：

$$\begin{aligned} d[i,j] &= \max\{d[i,j-1] + S[j], \max\{d[i-1,t] + S[j]\}\}, \text{其中} i \leq j \leq n, i-1 \leq t < j \\ target &= \max\{d[m,j]\}, \text{其中} m \leq j \leq n \end{aligned}$$

分为两种情况：

- 情况一，S[j] 包含在第 i 段之中，$d[i, j-1] + S[j]$。
- 情况二，S[j] 独立划分成为一段，$\max\{d[i-1,t] + S[j]\}$。

观察上述两种情况可知 d[i,j] 的值只和 d[i,j-1] 和 d[i-1,t] 这两个值相关，因此不需要二维数组，可以用滚动数组，只需要两个一维数组，用 d[j] 表示现阶段的最大值，即 $d[i, j-1] + S[j]$，用 prev[j] 表示上一阶段的最大值，即 $\max\{d[i-1,t] + S[j]\}$。

代码

mmss.c

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

/**
 * @brief 最大 m 段子序列和
 * @param[in] S 数组
 * @param[in] n 数组长度
 * @param[in] m m 段
 * @return 最大 m 段子序列和
 */
int mmss(int S[], int n, int m) {
    int max_sum, i, j;

    /* d[i] 表示现阶段最大值，prev[i] 表示上阶段最大值 */
    /* d[0], prev[0] 未使用 */
    int *d = (int*) calloc(n + 1, sizeof(int));
    int *prev = (int*) calloc(n + 1, sizeof(int));
    S--;  // 因为 j 是从 1 开始，而 S 从 0 开始，这里要减一

    for (i = 1; i <= m; ++i) {
        max_sum = INT_MIN;
        for (j = i; j <= n; ++j) {
            // 状态转移方程
            if (d[j - 1] < prev[j - 1])
                d[j] = prev[j - 1] + S[j];
            else
                d[j] = d[j - 1] + S[j];

            prev[j - 1] = max_sum;  // 存放上阶段最大值
            if (max_sum < d[j])
                max_sum = d[j];  // 更新 max_sum
        }
        prev[j - 1] = max_sum;
    }

    free(d);
    free(prev);
    return max_sum;
}

int main() {
    int n, m, i, *S;
    while (scanf("%d%d", &m, &n) == 2) {
        S = (int*) malloc(sizeof(int) * n);
        for (i = 0; i < n; ++i)
            scanf("%d", &S[i]);
        printf("%d\n", mmss(S, n, m));
        free(S);
    }
    return 0;
}
```

mmss.c

相关题目

与本题相同的题目：

- HDOJ 1024 Max Sum Plus Plus, http://acm.hdu.edu.cn/showproblem.php?pid=1024

与本题相似的题目：

- LeetCode Best Time to Buy and Sell Stock III, http://leetcode.com/onlinejudge#question_123，
  参考代码 https://gist.github.com/soulmachine/5906637

14.8 背包问题

背包问题 (Knapsack problem[①]) 有很多种版本，常见的是以下三种：

- 0-1 背包问题 (0-1 knapsack problem)：每种物品只有一个
- 完全背包问题 (UKP, unbounded knapsack problem)：每种物品都有无限个可用
- 多重背包问题 (BKP, bounded knapsack problem)：第 i 种物品有 c[i] 个可用

其他版本的背包问题请参考“背包问题九讲”，https://github.com/tianyicui/pack
背包问题是一种“多阶段决策问题”。

14.8.1 0-1 背包问题

描述

有 N 种物品，第 i 种物品的重量为 w_i，价值为 v_i，每种物品只有一个。背包能承受的重量为 W。将哪些物品装入背包可使这些物品的总重量不超过背包容量，且价值总和最大?

输入

第 1 行包含一个整数 T，表示有 T 组测试用例。每组测试用例有 3 行，第 1 行包含两个整数 $N, W (N \leq 1000, W \leq 1000)$ 分别表示物品的种数和背包的容量，第 2 行包含 N 个整数表示每种物品的价值，第 3 行包含 N 个整数表示每种物品的重量。

输出

每行一个整数，表示价值总和的最大值。

样例输入

```
1
5 10
1 2 3 4 5
5 4 3 2 1
```

样例输出

```
14
```

分析

由于每种物品仅有一个，可以选择装或者不装。
定义状态 f[i][j]，表示“把前 i 个物品装进容量为 j 的背包可以获得的最大价值”，则其状态转移方程便是：

$$f[i][j] = \max\{f[i-1][j], f[i-1][j-w[i]+v[i]\}$$

这个方程理解如下，把前 i 个物品装进容量为 j 的背包时，有两种情况：

- 第 i 个不装进去，这时所得价值为：$f[i-1][j]$
- 第 i 个装进去，这时所得价值为：$f[i-1][j-w[i]]+v[i]$

动规过程的伪代码如下：

[①] Knapsack problem, http://en.wikipedia.org/wiki/Knapsack_problem

```
f[0..N][0..W] = 0
for i=1..N
    for j=0..W
        f[i][j]=max{f[i-1][j],f[i-1][j-w[i]]+v[i]};
```

内循环从右向左也可以：

```
f[0..N][0..W] = 0
for i=1..N
    for j=W..0
        f[i][j]=max{f[i-1][j],f[i-1][j-w[i]]+v[i]};
```

内循环从右向左时，可以把二维数组优化成一维数组。伪代码如下：

```
for i=1..N
    for j=W..0
        d[j]=max{d[j],d[j-w[i]]+v[i]};
```

为什么呢？举个例子，测试数据如下：

```
1
3 10
4 5 6
3 4 5
```

f 是从上到下、从右到左计算的，如图 14-1所示。

最大容量M	物品个数N											
10	3		C	0	1	2	3	4	5	6	7	8	9	10
物品大小w	物品价值p	编号	0	0	0	0	0	0	0	0	0	0	0	0
3	4	i= 1	1	0	0	0	4	4	4	4	4	4	4	4
4	5	i-n 2	2	0	0	0	4	5	5	5	9	9	9	9
5	6	3w	3	0	0	0	4	5	6	6	9	10	11	11

图 14-1 0-1 背包问题的计算顺序

当内循环是逆序时，且动规是用自底向上方式实现时，就可以保证同一行可以从右向左更新。

设一维数组为 d（又称为滚动数组[①]），在更新 d[j] 之前，d[j] 里保存的 f[i-1][j]，更新之后，d[j] 里保存的是 f[i][j]。

事实上，使用一维数组解 0-1 背包问题的程序在后面会被多次用到，所以这里抽象出一个处理单个物品的函数，以后的代码中直接调用不加说明。

```
def ZeroOneKnapsack(d[], i)
    for j = W..w[i]
        d[j] = max(d[j], d[j-w[i]] + v[i])
```

有了这个函数以后，0-1 背包问题的伪代码就可以这样写：

```
d[0..W] = 0
for i = 1..N
    ZeroOneKnapsack(d[], i)
```

代码

版本 1，自底向上。

01knapsack.c

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

#define MAXN 1000
#define MAXW 1000

int N, W;
int w[MAXN+1], v[MAXN+1]; /* 0 没有用 */

int f[MAXN + 1][MAXW + 1];
```

[①] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009，第 169 页 9.3.3 节

```
void dp() {
    int i, j;
    memset(f, 0, sizeof(f)); /* 背包不一定要装满 */
    for(i = 1; i <= N; ++i) {
        /* for(j = W; j >= 0; --j) { /* 也可以 */
        for(j = 0; j <= W; ++j) {
            f[i][j] = f[i-1][j];
            if(j >= w[i]) {
                const int tmp = f[i-1][j-w[i]] + v[i];
                if(tmp > f[i][j]) f[i][j] = tmp;
            }
        }
    }
}

int main() {
    int i, T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &N, &W);
        for(i = 1; i <= N; ++i) scanf("%d", &v[i]);
        for(i = 1; i <= N; ++i) scanf("%d", &w[i]);

        dp();
        printf("%d\n", f[N][W]);
    }
    return 0;
}
```

01knapsack.c

版本 2，自底向上，滚动数组。

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

#define MAXN 1000
#define MAXW 1000

int N, W;
int w[MAXN], v[MAXN];

int d[MAXW + 1]; /* 滚动数组 */

/**
 * @brief 0-1 背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @return 无
 */
void zero_one_knapsack(int d[], const int i) {
    int j;
    for(j = W; j >= w[i]; --j) {
        const int tmp = d[j - w[i]] + v[i];
        if(tmp > d[j]) d[j] = tmp; /* 求最小用 <，求最大用 > */
    }
}

void dp() {
    int i;
    memset(d, 0, sizeof(d)); /* 背包不一定要装满 */

    for(i = 0; i < N; ++i) zero_one_knapsack(d, i);
}
```

01knapsack2.c

```
int main() {
    int i, T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &N, &W);
        for(i = 0; i < N; ++i) scanf("%d", &v[i]);
        for(i = 0; i < N; ++i) scanf("%d", &w[i]);

        dp();
        printf("%d\n", d[W]);
    }
    return 0;
}
```

01knapsack2.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①]第 167 页例题 9-5
- HDOJ 2602 Bone Collector, http://acm.hdu.edu.cn/showproblem.php?pid=2602

与本题相似的题目：

- wikioi 1014 装箱问题, http://www.wikioi.com/problem/1014/,
  参考代码 https://gist.github.com/soulmachine/6248518

14.8.2 完全背包问题

描述

给你一个储钱罐 (piggy bank)，往里面存硬币。存入的过程是不可逆的，要想把钱拿出来只能摔碎储钱罐。因此，你想知道里面是否有足够多的钱，把它摔碎是值得的。

你可以通过储钱罐的重量来推测里面至少有多少钱。已知储钱罐空的时候的重量和装了硬币后的重量，还有每种硬币的重量和面值，每种硬币的数量不限。求在最坏情况下，储钱罐里最少有多少钱。

输入

第 1 行包含一个整数 T，表示有 T 组测试用例。每组测试用例，第一行是两个整数 E 和 F，分别表示空储钱罐的重量和装了硬币后的重量，以克 (gram) 为单位，储钱罐的重量不会超过 10kg，即 $1 \leq E \leq F \leq 10000$。第二行是一个整数 $N(1 \leq N \leq 500)$，表示硬币的种类数目。接下来是 N 行，每行包含两个整数 v 和 w$(1 \leq v \leq 50000, 1 \leq w \leq 10000)$，分别表示硬币的面值和重量。

输出

每个案例打印一行。内容是"The minimum amount of money in the piggy-bank is X."，其中 X 表示储钱罐里最少有多少钱。如果不能精确地达到给定的重量，则打印"This is impossible."。

样例输入

```
3
10 110
2
1 1
30 50
10 110
2
1 1
```

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

```
50 30
1 6
2
10 3
20 4
```

样例输出

```
The minimum amount of money in the piggy-bank is 60.
The minimum amount of money in the piggy-bank is 100.
This is impossible.
```

分析

每种物品有无限个可用，这是完全背包问题。

本题没有给出储钱罐的容量，但每个案例给出了，初始为空时的重量 E 和装了硬币后的重量 F，因此可以把储钱罐看作一个容量为 F-E 的背包，背包必须要装满。

这个问题非常类似于 0-1 背包问题，所不同的是每种物品有无限个。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是取 0 个、取 1 个、取 2 个……直至取 $W/w[i]$ 个。

一种好想好写的基本方法是转化为 0-1 背包问题：把第 i 种物品换成 $W/w[i]$ 个 0-1 背包问题中的物品，则得到了物品数为 $\sum \frac{W}{w[i]}$ 的 0-1 背包问题。时间复杂度是 $O(W \sum \frac{W}{w[i]})$。

按照该思路，状态转移方程为：

$$f[i][j] = \max\{f[i-1][j-k*w[i]] + k*v[i], 0 \le k*w[i] \le j\}$$

伪代码如下：

```
for i = 1..N
    for j = W..w[i]
        for k = 1..j/w[i]
            d[j] = max{d[j], d[j-k*w[i]] + k*v[i]};
```

也可以写成：

```
for i = 1..N
    for k = 1..W/w[i]
        ZeroOneKnapsack(d[], w, v)
```

“拆分物品”还有更高效的拆分方法：把第 i 种物品拆分成重量为 $2^k * w[i]$、价值为 $2^k * v[i]$ 的若干物品，其中 k 取所有满足 $2^k * w[i] \le W$ 的非负整数。这是二进制的思想，因为闭区间 [1, W/w[i]] 中的任何整数都可以表示为 $1, 2, 4, ..., 2^k$ 中若干个的和。

这样处理单个物品的复杂度由 $O\left(\frac{W}{w[i]}\right)$ 降到了 $O\left(\log \frac{W}{w[i]}\right)$，伪代码如下：

```
def UnboundedKnapsack(d[], i)
    k=1
    while k*w[i] <= W
        ZeroOneKnapsack(d[], k*w[i], k*v[i])
        k=2*k
```

还存在更优化的算法，复杂度为 $O(NW)$，伪代码如下：

```
for i = 1..N
    for j = 0..W
        d[j] = max{d[j], d[j-w[i]] + v[i]};
```

与 0-1 背包问题相比，仅有一行代码不同，这里内循环是顺序的，而 0-1 背包是逆序的（在使用滚动数组的情况下）。

为什么这个算法可行呢？首先想想为什么 0-1 背包中内循环要逆序，逆序是为了保证每个物品只选一次，保证在“选择第 i 件物品”时，依赖的是一个没有选择第 i 件物品的子结果 $f[i-1][j-w[i]]$。而现在完全背包的特点却是每种物品可选无限个，没有了每个物品只选一次的限制，所以就可以并且必须采用 j 递增的顺序循环。

根据上面的伪代码，状态转移方程也可以写成这种形式：

$$f[i][j] = \max\{f[i-1][j], f[i][j-w[i]] + v[i]\}$$

抽象出处理单个物品的函数：

```
def UnboundedKnapsack(d[], i)
    for j = w[i]..W
        d[j] = max(d[j], d[j-w[i]] + v[i])
```

代码

piggy_bank.c

```c
#include <stdio.h>

#define MAXN 500
#define MAXW 10000
 /* 无效值，不要用 0x7FFFFFFF，执行加运算后会变成负数 */
const int INF = 0x0FFFFFFF;

int N, W;
int w[MAXN], v[MAXN];

int d[MAXW + 1]; /* 滚动数组 */

/**
 * @brief 完全背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @return 无
 */
void unbounded_knapsack(int d[], const int i) {
    int j;
    for(j = w[i]; j <= W; ++j) {
        const int tmp = d[j - w[i]] + v[i];
        if(tmp < d[j]) d[j] = tmp; /* 求最小用 <，求最大用 > */
    }
}
/** c，物品的系数 */
void zero_one_knapsack(int d[], const int i, const int c) {
    int j;
    const int neww = c * w[i];
    const int newv = c * v[i];
    for(j = W; j >= neww; --j) {
        const int tmp = d[j - neww] + newv;
        if(tmp < d[j]) d[j] = tmp;  /* 求最小用 <，求最大用 > */
    }
}

void unbounded_knapsack1(int d[], const int i) {
    int k = 1;
    while(k * w[i] <= W) {
        zero_one_knapsack(d, i, k);
        k *= 2;
    }
}

void dp() {
    int i;
    for(i = 0; i <= W; ++i) d[i] = INF; /* 背包要装满 */
    d[0] = 0;

    for(i = 0; i < N; ++i) unbounded_knapsack(d, i);
}

int main() {
    int i, T;
    int E, F;

    scanf("%d", &T);
```

```
    while(T--) {
        scanf("%d %d", &E, &F);
        W = F - E;
        scanf("%d", &N);
        for(i = 0; i < N; ++i) scanf("%d %d", &v[i], &w[i]);

        dp();
        if(d[W] == INF) {
            printf("This is impossible.\n");
        } else {
            printf("The minimum amount of money in the piggy-bank is %d.\n",
                    d[W]);
        }
    }
    return 0;
}

/* 将第 i 种物品取 0 个，1 个，...，W/w[i] 个，该版本不能 AC，会 TLE */
void unbounded_knapsack2(int d[], const int w, const int v) {
    int j, k;
    for(j = W; j >= w; --j) {
        const int K = j / w;
        for(k = 1; k <= K; ++k) {
            const int tmp = d[j - k * w] + k * v;
            if(tmp < d[j]) d[j] = tmp; /* 求最小用 <，求最大用 > */
        }
    }
}

/* 将第 i 种物品取 0 个，1 个，...，W/w[i] 个，该版本不能 AC，会 TLE */
void unbounded_knapsack3(int d[], const int w, const int v) {
    int k;
    const int K = W / w;
    for(k = 0; k < K; ++k){
        zero_one_knapsack(d, w, v);
    }
}
```

piggy_bank.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①]第 167 页例题 9-4
- POJ 1384 Piggy-Bank, http://poj.org/problem?id=1384
- HDOJ 1114 Piggy-Bank, http://t.cn/zWXbXIn

与本题相似的题目：

- POJ 2063 Investment, http://poj.org/problem?id=2063

14.8.3 多重背包问题

描述

某地发生地震，为了挽救灾区同胞的生命，心系灾区同胞的你准备自己采购一些粮食支援灾区，现在假设你一共有资金 W 元，而市场有 N 种大米，每种大米都是袋装产品，其价格不等，并且只能整袋购买。

请问：你用有限的资金最多能采购多少公斤粮食呢?

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

输入

第 1 行包含一个整数 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 W 和 N($1 \leq W \leq 100, 1 \leq N \leq 100$), 分别表示经费的金额和大米的种类，然后是 N 行数据，每行包含 3 个整数 w，v 和 c ($1 \leq w \leq 20, 1 \leq v \leq 200, 1 \leq c \leq 20$)，分别表示每袋的价格、每袋的重量以及对应种类大米的袋数。

输出

对于每组测试用例，输出能够购买大米的最大重量，你可以假设经费买不光所有的大米，并且经费你可以不用完。每个实例的输出占一行。

样例输入

```
1
8 2
2 100 4
4 100 2
```

样例输出

```
400
```

分析

第 i 种物品有 c[i] 个可用，这是多重背包问题。

与完全背包问题类似，也可以用“拆分物品”的思想把本问题转化为 0-1 背包问题：把第 i 种物品换成 $c[i]$ 个 0-1 背包问题中的物品，则得到了物品数为 $\sum c[i]$ 的 0-1 背包问题。时间复杂度是 $O(W \sum c[i])$。状态转移方程为：

$$f[i][j] = \max\{f[i-1][j-k*w[i]] + k*v[i], 0 \leq k \leq c[i], 0 \leq k*w[i] \leq j\}$$

伪代码如下：

```
for i = 1..N
    for j = W..w[i]
        K = min{j/w[i], c[i]}
        for k = 1..K
            d[j] = max{d[j], d[j-k*w[i]] + k*v[i]};
```

也可以写成：

```
for i = 1..N
    for k = 1..c[i]
        ZeroOneKnapsack(d[], i)
```

拆分物品也可以使用二进制的技巧，把第 i 种物品拆分成若干物品，其中每件物品都有一个系数，这个新物品的重量和价值均是原来的重量和价值乘以这个系数。系数分别为 $1, 2, 2^2, ..., 2^{k-1}, c[i] - 2^k + 1$，其中 k 是满足 $2^k - 1 < c[i]$ 的最大整数。例如，某种物品有 13 个，即 c[i]=13，则相应的 k=3，这种物品应该被拆分成系数分别 1,2,4,6 的四个物品。

这样处理单个物品的复杂度由 $O(c[i])$ 降到了 $O(\log c[i])$，伪代码如下：

```
// c, 物品系数
def ZeroOneKnapsack(d[], i, c)
    for j = W..w[i]
        d[j] = max(d[j], d[j-c*w[i]] + c*v[i])
def BoundedKnapsack(d[], i)
    if c[i]*w[i] >= W
        unbounded_knapsack(d[], i);
        return;

    k = 1;
    while k < c[i]
        zero_one_knapsack(d[], i, k);
        c[i] -= k;
        k *= 2;

    zero_one_knapsack(d[], i, c);
```

代码

bkp.c

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

#define MAXN 100
#define MAXW 100

int N, W;
int w[MAXN], v[MAXN], c[MAXN];

int d[MAXW + 1]; /* 滚动数组 */

/** c, 物品的系数 */
void zero_one_knapsack(int d[], const int i, const int c) {
    int j;
    const int neww = c * w[i];
    const int newv = c * v[i];
    for(j = W; j >= neww; --j) {
        const int tmp = d[j - neww] + newv;
        if(tmp > d[j]) d[j] = tmp;  /* 求最小用 <, 求最大用 > */
    }
}

void unbounded_knapsack(int d[], const int i) {
    int j;
    for(j = w[i]; j <= W; ++j) {
        const int tmp = d[j - w[i]] + v[i];
        if(tmp > d[j]) d[j] = tmp; /* 求最小用 <, 求最大用 > */
    }
}

/**
 * @brief 多重背包问题中, 处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @param[in] c 该物品的数量
 * @return 无
 */
void bounded_knapsack(int d[], const int i) {
    int k;
    for(k = 0; k < c[i]; ++k) {
        zero_one_knapsack(d, i, 1);
    }
}

/* 另一个版本, 拆分物品更加优化 */
void bounded_knapsack1(int d[], const int i) {
    int k;
    if(c[i] * w[i] >= W) {
        unbounded_knapsack(d, i);
        return;
    }

    k = 1;
    while(k < c[i]) {
        zero_one_knapsack(d, i, k);
        c[i] -= k;
        k *= 2;
    }
    zero_one_knapsack(d, i, c[i]);
}
```

```
void dp() {
    int i;
    memset(d, 0, sizeof(d)); /* 背包不一定要装满 */

    for(i = 0; i < N; ++i) bounded_knapsack1(d, i);
}

int main() {
    int i;
    int T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &W, &N);
        for(i = 0; i < N; ++i) scanf("%d %d %d", &w[i], &v[i], &c[i]);

        dp();
        printf("%d\n", d[W]);
    }
    return 0;
}
```

bkp.c

相关的题目

与本题相同的题目：

- HDOJ 2191 买大米, http://acm.hdu.edu.cn/showproblem.php?pid=2191

与本题相似的题目：

- None

14.9 序列型动态规划

对于所有动规题目，如果把状态转移图画出来，一定是一个有向无环图 (DAG)。再进一步细分类别，有序列型动态规划，棋盘型动态规划，树型动态规划等等。

14.9.1 最长上升子序列

描述

当一个序列严格递增时，我们称这个序列是上升的。对于一个给定的序列 $(a_1, a_2, ..., a_N)$，我们可以得到一些上升的子序列 $(a_{i1}, a_{i2}, ..., a_{iK})$，这里 $1 \leq i1 < i2 < ... < iK \leq N$。例如，对于序列 (1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如 (1, 7), (3, 4, 8) 等等，这些子序列中最长的长度是 4，比如子序列 (1, 3, 5, 8)。

对于给定的序列，求**最长上升子序列** (longest increasing subsequence) 的长度。

输入

第一行是序列的长度 $N(1 \leq N \leq 1000)$。第二行给出序列中的 N 个整数，这些整数的取值范围都在 0 到 10000。

输出

最长上升子序列的长度。

样例输入

```
7
1 7 3 5 9 4 8
```

样例输出

```
4
```

分析

设状态为 d[j]，表示以 a_j 为终点的最长上升子序列的长度。状态转移方程如下：

$$d[j] = \begin{cases} 1 & j = 1 \\ \max\{d[i]\} + 1 & 1 < i < j, a_i < a_j \end{cases}$$

代码

lis.c

```
#include<stdio.h>
#define MAXN 1001 // a[0] 未用

int N;
int a[MAXN];
int d[MAXN];

void dp() {
    int i, j;
    d[1] = 1;

    for (j = 2; j <= N; j++) { // 每次求以 aj 为终点的最长上升子序列的长度
        int max = 0;  // 记录 aj 左边的上升子序列的最大长度
        for (i = 1; i < j; i++)  if (a[i] <a[j] && max < d[i]) max = d[i];
        d[j] = max + 1;
    }
}

int main() {
    int i, max;
    scanf("%d",&N);
    for (i = 1; i <= N;i++) scanf("%d",&a[i]);

    dp();

    max = 0;
    for(i = 1; i <= N;i++) if (d[i] > max) max = d[i];
    printf("%d\n",max);
    return 0;
}
```

lis.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[①]第 198 页例题 10.3
- 百练 2757 最长上升子序列, http://poj.grids.cn/practice/2757/
- POJ 2533 Longest Ordered Subsequence, http://poj.org/problem?id=2533
- wikioi 1576, 最长严格上升子序列, http://www.wikioi.com/problem/1576/

与本题相似的题目：

- wikioi 1044 拦截导弹，http://www.wikioi.com/problem/1044/,
  参考代码 https://gist.github.com/soulmachine/6248489

[①]李文新, 程序设计导引及在线实践, 清华大学出版社,2007

14.9.2 嵌套矩形

描述

有 n 个矩形，每个矩形可以用 a, b 来描述，表示长和宽。矩形 $X(a, b)$ 可以嵌套在矩形 $Y(c, d)$ 中当且仅当 $a < c, b < d$ 或者 $b < c, a < d$（相当于旋转 X90 度）。例如（1,5）可以嵌套在（6,2）内，但不能嵌套在（3,4）中。你的任务是选出尽可能多的矩形排成一行，使得除最后一个外，每一个矩形都可以嵌套在下一个矩形内。

输入

第一行是一个正整数 $N(0 < N < 10)$，表示测试数据组数，每组测试数据的第一行是一个正正数 n，表示该组测试数据中含有矩形的个数 $(n <= 1000)$ 随后的 n 行，每行有两个数 $a, b(0 < a, b < 100)$，表示矩形的长和宽

输出

每组测试数据都输出一个数，表示最多符合条件的矩形数目，每组输出占一行

样例输入

```
1
10
1 2
2 4
5 8
6 10
7 9
3 1
5 8
12 10
9 7
2 2
```

样例输出

```
5
```

分析

本题实质上是求 DAG 中不固定起点的最长路径。

设 d[i] 表示从结点 i 出发的最长长度，状态转移方程如下：

$$d[i] = \max\{d[j] + 1 | (i, j) \in E\}$$

其中，E 为边的集合。最终答案是 d[i] 中的最大值。

代码

embedded_rectangles.c

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

#define MAXN 1000 // 矩形最大个数

int n; // 矩形个数
int G[MAXN][MAXN]; // 矩形包含关系
int d[MAXN]; // 表格

/**
 * @brief 备忘录法.
 * @param[in] i 起点
 * @return 以 i 为起点，能达到的最长路径
```

```
 */
int dp(const int i) {
    int j;
    int *ans= &d[i];
    if(*ans > 0) return *ans;

    *ans = 1;
    for(j = 0; j < n; j++) if(G[i][j]) {
        const int next = dp(j) + 1;
        if(*ans < next) *ans = next;
    }
    return *ans;
}

/**
 * @brief 按字典序打印路径.
 *
 * 如果多个 d[i] 相等, 选择最小的 i。
 *
 * @param[in] i 起点
 * @return 无
 */
void print_path(const int i) {
    int j;
    printf("%d ", i);
    for(j = 0; j < n; j++) if(G[i][j] && d[i] == d[j] + 1) {
        print_path(j);
        break;
    }
}

int main() {
    int N, i, j;
    int max, maxi;
    int a[MAXN],b[MAXN];

    scanf("%d", &N);
    while(N--) {
        memset(G, 0, sizeof(G));
        memset(d, 0, sizeof(d));

        scanf("%d", &n);
        for(i = 0; i < n; i++) scanf("%d%d", &a[i], &b[i]);

        for(i = 0; i < n; i++)
            for(j = 0; j < n; j++)
                if((a[i] > a[j] && b[i] > b[j]) ||
                    (a[i] > b[j] && b[i] > a[j])) G[i][j] = 1;

        max = 0;
        maxi = -1;
        for(i = 0; i < n; i++) if(dp(i) > max) {
            max = dp(i);
            maxi = i;
        }
        printf("%d\n", max);
        // print_path(maxi);
    }
    return 0;
}
```

embedded_rectangles.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[1]第 161 页 9.2.1 节
- NYOJ 16 嵌套矩形, http://acm.nyist.net/JudgeOnline/problem.php?pid=16

与本题相似的题目：

- None

14.9.3 线段覆盖 2

描述

数轴上有 $n(n \leq 1000)$ 条线段，线段的两端都是整数坐标，坐标范围在 $0 \sim 1000000$，每条线段有一个价值，请从 n 条线段中挑出若干条线段，使得这些线段两两不覆盖（端点可以重合）且线段价值之和最大。

输入

第一行一个整数 n，表示有多少条线段。
接下来 n 行每行三个整数, a_i, b_i, c_i，分别代表第 i 条线段的左端点 a_i，右端点 b_i（保证左端点 < 右端点）和价值 c_i。

输出

输出能够获得的最大价值

样例输入

```
3
1 2 1
2 3 2
1 3 4
```

样例输出

```
4
```

分析

先将线段排序。按照右端点从小到大排序。原因是循环结构中是 i 从 1 到 n, i 比较小的时候尽可能选右端点比较小的，这样才可以为后面的线段留下更大的空间。
设状态为 f[i]，表示前 i 条线段时，选上第 i 条线段，能获得的最大价值。状态转移方程如下：

$$f[i] = \max\{f[j]\} + c[i], 2 \leq i \leq n, 1 \leq j \leq i-1, \text{且 b[j]<=a[i]}$$

b[j]<=a[i] 表示 j 的右端点在 i 的左端点左边，即不重合。
输出 f[i] 数组中的最大值。

代码

lines_cover.c

```
/* http://www.wikioi.com/problem/3027/ */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAXN 1000
```

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

```
typedef struct line_t {
    int a, b, c;
} line_t;

int line_cmp(const void *a, const void *b) {
    line_t *la = (line_t*)a;
    line_t *lb = (line_t*)b;
    return la->b - lb->b;
}

int n;
line_t line[MAXN];
int f[MAXN];

void dp() {
    int i, j;
    qsort(line, n, sizeof(line_t), line_cmp);
    f[0] = line[0].c;

    for (i = 1; i < n; i++) {
        int max = 0;
        for (j = 0; j < i; j++) {
            if (line[j].b <= line[i].a) max = max > f[j] ? max : f[j];
        }
        f[i] = max + line[i].c;
    }
}


static int max_element(const int a[], int begin, int end) {
    int i;
    int max_value = INT_MIN;
    int max_pos = -1;
    for (i = begin; i < end; i++) {
        if (max_value < a[i]) {
            max_value = a[i];
            max_pos = i;
        }
    }
    return max_pos;
}


int main() {
    int i;
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d%d%d", &line[i].a, &line[i].b, &line[i].c);

    dp();

    printf("%d\n", f[max_element(f, 0, n)]);
    return 0;
}
```

lines_cover.c

相关的题目

与本题相同的题目：

- wikioi 3027 线段覆盖 2, http://www.wikioi.com/problem/3027/

与本题相似的题目：

- None

14.9.4 硬币问题

描述

有 n 种硬币，面值为别为 $v_1, v_2, v_3, \cdots, v_n$，每种都有无限多。给定非负整数 S，可以选取多少个硬币，使得面值和恰好为 S？输出硬币数目的最小值和最大值。$1 \leq n \leq 100, 1 \leq S \leq 10000, 1 \leq v_i \leq S$。

输入

第 1 行 n，第 2 行 S，第 3 到 $n+2$ 行为 n 种不同的面值。

输出

第 1 行为最小值，第 2 行为最大值。

样例输入

```
3
6
1
2
3
```

样例输出

```
2
6
```

分析

本题实质上是求 DAG 中固定终点的最长路径和最短路径。

把每种面值看作一个点，表示“还需要凑足的面值”，则初始状态为 S，目标状态为 0。若当前状态为 i，每使用一个硬币 j，状态便转移到 $i - v_j$。

设状态为 d[i]，表示从节点 i 出发的最长路径长度，则原问题的解是 d[S]。状态转移方程如下：

$$d[i] = \max\{d[j] + 1, (i, j) \in E\}$$

本题还可以看作是完全背包问题（见 §14.8.2节）：背包容量为 S，背包必须要装满，物品即硬币，每个硬币的费用为面值 v_i，价值均为 1。求背包中物品的最小价值和最大价值。

代码

版本 1，备忘录法。

coin_change.c

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

#define MAXN 100
#define MAXV 10000

/** 硬币面值的种类. */
int n;
/** 要找零的数目. */
int S;
/** 硬币的各种面值. */
int v[MAXN];
/** min[i] 表示面值之和为 i 的最短路径的长度, max 则是最长. */
int min[MAXV + 1], max[MAXV + 1];

/**
```

```
 * @brief 最短路径.
 * @param[in] s 面值
 * @return 最短路径长度
 */
int dp1(const int s) { // 最小值
    int i;
    int *ans = &min[s];
    if(*ans != -1) return *ans;
    *ans = 1<<30;
    for(i = 0; i < n; ++i) if(v[i] <= s) {
        const int tmp = dp1(s-v[i])+1;
        *ans = *ans < tmp ? *ans : tmp;
    }
    return *ans;
}

int visited[MAXV + 1];
/**
 * @brief 最长路径.
 * @param[in] s 面值
 * @return 最长路径长度
 */
int dp2(const int s) { //最大值
    int i;
    int *ans = &max[s];

    if(visited[s]) return max[s];
    visited[s] = 1;

    *ans = -1<<30;
    for(i = 0; i < n; ++i) if(v[i] <= s) {
        const int tmp = dp2(s-v[i])+1;
        *ans = *ans > tmp ? *ans : tmp;
    }
    return *ans;
}

void print_path(const int* d, const int s);

int main() {
    int i;
    scanf("%d%d", &n, &S);
    for(i = 0; i < n; ++i) scanf("%d", &v[i]);

    memset(min, -1, sizeof(min));
    min[0] = 0;
    printf("%d\n", dp1(S));
    // print_path(min, S);

    memset(max, -1, sizeof(max));
    memset(visited, 0, sizeof(visited));
    max[0] = 0; visited[0] = 1;
    printf("%d\n", dp2(S));
    // print_path(max, S);

    return 0;
}

/**
 * @brief 打印路径.
 * @param[in] d 上面的 min 或 min
 * @param[in] s 面值之和
 * @return 无
 */
```

```
void print_path(const int* d, const int s) {//打印的是边
    int i;
    for(i = 0; i < n; ++i) if(v[i] <= s && d[s-v[i]] + 1 == d[s]) {
        printf("%d ",i);
        print_path(d, s-v[i]);
        break;
    }
    printf("\n");
}
```

coin_change.c

版本 2，自底向上。

coin_change2.c

```
#include<stdio.h>

#define MAXN 100
#define MAXV 10000

int n, S, v[MAXN], min[MAXV + 1], max[MAXV + 1];
int min_path[MAXV], max_path[MAXV];

void dp() {
    int i, j;

    min[0] = max[0] = 0;
    for(i = 1; i <= S; ++i) {
        min[i] = MAXV;
        max[i] = -MAXV;
    }

    for(i = 1; i <= S; ++i) {
        for(j = 0; j < n; ++j) if(v[j] <= i) {
            if(min[i-v[j]] + 1 < min[i]) {
                min[i] = min[i-v[j]] + 1;
                min_path[i] = j;
            }
            if(max[i-v[j]] + 1 > max[i]) {
                max[i] = max[i-v[j]] + 1;
                max_path[i] = j;
            }
        }
    }
}

void print_path(const int *d, int s);

int main() {
    int i;
    scanf("%d%d", &n, &S);
    for(i = 0; i < n; ++i) scanf("%d", &v[i]);

    dp();
    printf("%d\n", min[S]);
    // print_path(min_path, S);
    printf("%d\n", max[S]);
    // print_path(max_path, S);
    return 0;
}

/**
 * @brief 打印路径.
 * @param[in] d 上面的 min_path 或 min_path
 * @param[in] s 面值之和
 * @return 无
```

```
 */
void print_path(const int *d, int s) {
    while(s) {
        printf("%d ", d[S]);
        S -= v[d[s]];
    }
    printf("\n");
}
```

coin_change2.c

版本 3，当作完全背包问题。

coin_change3.c

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

#define MAXN 100
#define MAXW 10000
 /* 无效值，不要用 0x7FFFFFFF，执行加运算后会变成负数 */
const int INF = 0xOFFFFFFF;

int N, W;
int w[MAXN], v[MAXN];

int min[MAXW + 1], max[MAXW + 1]; /* 滚动数组 */

int min_path[MAXW + 1], max_path[MAXW + 1];
void print_path(const int *d, int s);

/**
 * @brief 完全背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @return 无
 */
void unbounded_knapsack(int min[], int max[], const int i) {
    int j;
    for(j = w[i]; j <= W; ++j) {
        if(min[j - w[i]] + v[i] < min[j]) {
            min[j] = min[j - w[i]] + v[i];
            // min_path[j] = i;
        }
        if(max[j - w[i]] + v[i] > max[j]) {
            max[j] = max[j - w[i]] + v[i];
            // max_path[j] = i;
        }
    }
}

void dp() {
    int i, j;
    min[0] = 0;
    max[0] = 0;
    for(j = 1; j <= W; ++j) { /* 背包要装满 */
        min[j] = INF;
        max[j] = -INF;
    }

    for(i = 0; i < N; ++i) unbounded_knapsack(min, max, i);
}

int main() {
    int i;
    for (i = 0; i < MAXN; ++i) v[i] = 1;
    scanf("%d%d", &N, &W);
```

```
    for(i = 0; i < N; ++i) scanf("%d", &w[i]);

    dp();
    printf("%d\n", min[W]);
    // print_path(min_path, W);
    printf("%d\n", max[W]);
    // print_path(max_path, W);
    return 0;
}

/**
 * @brief 打印路径.
 * @param[in] d 上面的 min_path 或 min_path
 * @param[in] j 面值之和
 * @return 无
 */
void print_path(const int *d, int j) {
    while(j) {
        printf("%d ", d[j]);
        j -= w[d[j]];
    }
    printf("\n");
}
```

coin_change3.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[1]第 162 页例题 9-3
- tyvj 1214 硬币问题, http://www.tyvj.cn/problem_show.aspx?id=1214

与本题相似的题目：

- None

14.10 区间型动态规划

14.10.1 最优矩阵链乘

描述

一个 $m \times n$ 的矩阵乘以一个 $n \times p$ 的矩阵等于一个 $m \times p$ 的矩阵，运算量为 mnp。

矩阵乘法不满足分配律，但满足结合律，即 $A \times B \times C = (A \times B) \times C = A \times (B \times C)$。

假设 A、B、C 分别是 2×3、3×4、4×5 的矩阵，则 $(A \times B) \times C$ 的运算量为 $2 \times 3 \times 4 + 2 \times 4 \times 5 = 64$，$A \times (B \times C)$ 的运算量为 $3 \times 4 \times 5 + 2 \times 3 \times 5 = 90$，显然第一种运算顺序节省运算量。

给出 N 个矩阵组成的序列，设计一种方法把它们依次乘起来，使得总的运算量尽量小。

输入

对于每个矩阵序列，只给出它们的维度。每个序列由两个部分组成，第一行包含一个整数 N，表示矩阵的个数；接下来的 N 行，每行一对整数，分别表示矩阵的行数和列数。给出的顺序与矩阵链乘的顺序一致。最后一行 N 为 0，表示输入结束。N 不大于 10。

输出

假设矩阵命名为 $A_1, A_2, ..., A_N$。对每个测试用例输出一行，包含一个使用了小括号的表达式，清晰地指出乘法的先后顺序。每行输出以"Case x: " 为前缀，x 表示测试用例编号。如果有多个正确结果，只需要输出其中一个。

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

样例输入

```
3
1 5
5 20
20 1
3
5 10
10 20
20 35
6
30 35
35 15
15 5
5 10
10 20
20 25
0
```

样例输出

```
Case 1: (A1 x (A2 x A3))
Case 2: ((A1 x A2) x A3)
Case 3: ((A1 x (A2 x A3)) x ((A4 x A5) x A6))
```

分析

假设第 i 个矩阵 A_i 的维度是 $p_{i-1} \times p_i$, i=1..N。

设状态为 d[i][j]，表示子问题 $A_i, A_{i+1}, ..., A_j$ 的最优解，状态转移方程如下：

$$d[i][j] = \min\{d[i][k] + d[k+1][j] + p_{i-1}p_kp_j\}$$

代码

oams.c

```
#include <stdio.h>
#include <memory.h>
#include <limits.h>

#define INF INT_MAX
#define MAXN 10

int N;  /** 矩阵的个数. */
int p[MAXN + 1];  /** 矩阵 Ai 的维度是 p[i-1]xp[i]. */
int d[MAXN][MAXN];  /** 状态, d[i][j] 表示子问题 Ai~Aj 的最优解. */
int s[MAXN][MAXN]; /** 子问题 Ai~Aj 应该在 s[i][j] 处断开 */

/**
 * @brief 打印子问题 Ai~Aj 的解
 * @param[in] i Ai
 * @param[in] j Aj
 * @return 无
 */
void print(const int i, const int j) {
    if (i == j) {
        printf("A%d",i);
    } else {  /* i < j */
        printf("(");
        print(i, s[i][j]);
        printf(" x ");
        print(s[i][j]+1, j);
        printf(")");
    }
```

```
}

void dp() {
    int i, j, k, l; /* l 表示区间长度 */
    for (i = 1;i <= N; ++i) d[i][i]=0;

    for (l = 2; l <= N; ++l) {
        for (i = 1; i <= N - l + 1; ++i) {
            j = i + l -1;
            d[i][j] = INF;
            for (k = i; k < j; ++k) {
                if (d[i][j] > d[i][k] + d[k+1][j] + p[i-1] * p[k] * p[j]) {
                    d[i][j] = d[i][k] + d[k+1][j] + p[i-1] * p[k] * p[j];
                    s[i][j] = k;
                }
            }
        }
    }
}

int main() {
    int i;
    int cas = 1;
    while (scanf("%d", &N) && N > 0) {
        memset(s, 0, sizeof(s));
        for (i = 0; i < N; ++i)  scanf("%d %d",&p[i],&p[i+1]);

        dp();

        printf("Case %d: ", cas++);
        print(1, N);
        printf("\n");
    }
    return 0;
}
```

oams.c

相关的题目

与本题相同的题目：

- UVa 348 Optimal Array Multiplication Sequence, http://t.cn/zH2pchp
- ZOJ 1276 Optimal Array Multiplication Sequence, http://t.cn/zH2gW1P

与本题相似的题目：

- wikioi 1154, 能量项链, http://www.wikioi.com/problem/1154,
  参考代码 https://gist.github.com/soulmachine/6248459
- Matrix67 - 十个利用矩阵乘法解决的经典题目，http://www.matrix67.com/blog/archives/276

14.10.2 石子合并

描述

在一条直线上摆着 N 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

试设计一个算法，计算出将 N 堆石子合并成一堆的最小得分。

输入

第一行是一个数 $N, 1 \leq N \leq 40000$，表示堆数。

接下来的 N 行每行一个数，表示该堆的石子数目。

输出

一个整数，即 N 堆石子合并成一堆的最小得分。

样例输入

```
4
1
1
1
1
```

样例输出

```
8
```

分析

这题与前面的矩阵链乘非常相似，只是计算代价的方式不同。设状态为 $d(i,j)$，表示子问题第 i 堆到第 j 堆石子的最优解，状态转移方程如下：

$$d(i,j) = \min\{d(i,k) + d(k+1,j) + \text{sum}(i,j)\}$$

$\text{sum}(i,j)$ 表示从第 i 堆到第 j 堆的石子总数。代码见 https://gist.github.com/soulmachine/6195139

上面的动规算法可以用四边形不等式进行优化，将时间复杂度从 $O(N^3)$ 下降到 $O(N^2)$。代码见 https://TODO

无论如何，时间复杂度都是 $O(N^2)$，本题中 N 范围特别大，开不了这么大的二维数组。所以动规算法只能处理小规模的情况。下面介绍第三种解法，Garsia-Wachs 算法[①]。

假设我们只对 3 堆石子 a,b,c 进行合并, 先合并哪两堆, 能使得得分最小？有两种方案：

score1 = (a+b) + ((a+b)+c)

score2 = (b+c) + ((b+c)+a)

假设 score1 <= score2, 可以推导出 a <= c，反过来，只要 a 和 c 的关系确定，合并的顺序也确定。这就是 **2-递减性质**。

Garsia-Wachs 算法，就是基于这个性质，找出序列中满足 A[i-1] <= A[i+1] 最小的 i，合并 temp = A[i]+A[i-1]，接着往前面找是否有满足 A[j] > temp，把 temp 值插入 A[j] 的后面 (数组的右边)。循环这个过程一直到只剩下一堆石子结束。

例如,

13 9 5 7 8 6 14

首先，找第一个满足 A[i-1] <= A[i+1] 的三个连续点，本例中就是 5 7 8，5 和 7 合并得到 12，12 不是丢在原来的位置，而是将它插入到第一个比它大的元素的后面，得到

13 12 9 8 6 14

接着来，从左到右搜索三个连续点，找到 8 6 14，合并 8 和 6 的到 14，将 14 插入到适当位置，得到 14 13 12 9 14

找到 12 9 14，合并 12 和 9 得到 21，将 21 插入到适当位置，得到 21 14 13 14

找到 14 13 14，合并 14 和 13 得到 27，将 27 插入到适当位置，得到 27 21 14

因为 27<14，先合并 21 和 14，得到 35，最后合并 27 和 35，得到 62，就是最终答案。

为什么要将 temp 插入 A[j] 的后面, 可以理解为是为了保持 2-递减性质。从 A[j+1] 到 A[i-2] 看成一个整体 A[mid]，现在 A[j], A[mid], temp(A[i-1]+A[i])，因为 temp < A[j]，因此不管怎样都是 A[mid] 和 temp 先合并, 所以将 temp 值插入 A[j] 的后面不会影响结果。

代码

stone_merge.c

```
#include <stdio.h>
#define MAXN 55555

int N, A[MAXN];  /* 堆数，每堆的石头个数 */
int num, result;  /* 数组实际长度，结果 */

void combine(int k) {  /* 前提 A[k-1] < A[k+1] */
```

[①] Donald E. Knuth, The art of computer programming, volume 3, 2nd ed, p446

```
    int i, j;

    /* 合并 k-1 和 k */
    int temp=A[k] + A[k-1];
    result += temp;
    /* 紧缩 */
    for(i = k; i < num - 1; i++) A[i] = A[i+1];
    num--;
    /* 插入 temp 到合适位置 */
    for(j = k-1; j > 0 && A[j-1] < temp; j--) A[j] = A[j-1];
    A[j] = temp;

    /* why? */
    while(j >= 2 && A[j] >= A[j-2]) {
        int d = num - j;
        combine(j - 1);
        j = num - d;
    }
}

int main() {
    int i;

    scanf("%d",&N);
    if(N == 0)    return 0;
    for(i = 0; i < N; i++) scanf("%d", &A[i]);

    num=2;
    result=0;
    for(i = 2; i < N; i++) {
        A[num++] = A[i];
        while(num >= 3 && A[num-3] <= A[num-1]) combine(num-2);
    }
    while(num > 1) combine(num-1);
    printf("%d\n",result);
    return 0;
}
```

stone_merge.c

相关的题目

与本题相同的题目：

- WIKIOI 1048 石子归并（数据规模很小）, http://www.wikioi.com/problem/1048/
- WIKIOI 2298 石子合并（数据规模很大）, http://www.wikioi.com/problem/2298/
- POJ 1738 An old Stone Game, http://poj.org/problem?id=1738

与本题相似的题目：

- None

14.10.3 矩阵取数游戏

描述

帅帅经常跟同学玩一个矩阵取数游戏：对于一个给定的 $n * m$ 的矩阵，矩阵中的每个元素 a_{ij} 均为非负整数。游戏规则如下：

- 每次取数时须从每行各取走一个元素，共 n 个，m 次后取完矩阵所有元素；
- 每次取数时须从每行各取走一个元素，共 n 个，m 次后取完矩阵所有元素；每次取走的各个元素只能是该元素所在行的行首或行尾；
- 每次取数时须从每行各取走一个元素，共 n 个，m 次后取完矩阵所有元素；每次取数都有一个得分值，为每行取数的得分之和，每行取数的得分 = 被取走的元素值 $*2i$，其中 i 表示第 i 次取数（从 1 开始编号）；

- 每次取数时须从每行各取走一个元素，共 n 个，m 次后取完矩阵所有元素；游戏结束总得分为 m 次取数得分之和。

帅帅想请你帮忙写一个程序，对于任意矩阵，可以求出取数后的最大得分。

输入

第 1 行为两个用空格隔开的整数 n 和 $m, 1 \leq n, m \leq 90$。第 $2 \sim n+1$ 行为 $n * m$ 矩阵，其中每行有 m 个用单个空格隔开的非负整数 $a_{ij}, 0 \leq a_{ij} \leq 1000$。

输出

输出仅包含 1 行，为一个整数，即输入矩阵取数后的最大得分。

样例输入

```
2 3
1 2 3
3 4 2
```

样例输出

```
82
```

样例解释

第 1 次：第 1 行取行首元素，第 2 行取行尾元素，本次得分为 1*21+2*21=6
第 2 次：两行均取行首元素，本次得分为 2*22+3*22=20
第 3 次：得分为 3*23+4*23=56。总得分为 6+20+56=82

分析

首先，每行之间是互相独立的，因此可以分别求出每行的最大得分，最后相加起来。
设状态为 f(i,j)，表示单行的区间 [i,j] 的最大值，转移方程为

$$f(i,j) = \max\left\{f(i+1,j) + a(i) * 2^x, f(i,j-1) + a(j) * 2^x\right\}, 1 \leq x \leq m$$

等价于

$$f(i,j) = 2 * \max\left\{f(i+1,j) + a(i), f(i,j-1) + a(j)\right\}$$

同时，注意到，最大得分可能会非常大，因此需要用到大整数运算。预估一下最大得分，大概是 $1000 * 2^{80}$，用计算器算一下，有 28 位。

代码

wikioi1166_matrix_game.c

```c
#include <stdio.h>
#include <memory.h>
#include <limits.h>

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_RADIX 10000
#define RADIX_LEN 4
/* 1000*2^80 有 28 位 */
#define MAX_LEN (28/RADIX_LEN+1)  /* 整数的最大位数 */

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
```

```
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 一个 32 位整数转化为大整数 bigint.
 * @param[in] n 32 位整数
 * @param[out] x 大整数
 * @return 大整数
 */
int* int_to_bigint(int n, int x[]) {
    int i = 0;
    memset(x, 0, MAX_LEN * sizeof(int));
    while (n > 0) {
        x[i++] = n % BIGINT_RADIX;
        n /= BIGINT_RADIX;
    }
    return x;
}


/**
 * @brief 大整数比较.
 * @param[in] x x
 * @param[in] y y
 * @return x<y，返回-1，相等于返回 0，大于返回 1.
 */
int bigint_cmp(const int x[], const int y[]) {
    int i, lenx = 0, leny = 0;
    for (i = 0; i < MAX_LEN; i++) {
        if (x[i] != 0) lenx++;
        else break;
    }
    for (i = 0; i < MAX_LEN; i++) {
        if (y[i] != 0) leny++;
        else break;
    }
    if (lenx < leny) return -1;
    else if (lenx > leny) return 1;
    else {
        for (i = lenx - 1; i >= 0; i--) {
            if (x[i] == y[i]) continue;
            else if (x[i] > y[i]) return 1;
            else return -1;
        }
        return 0;
    }
}

/**
 * @brief 大整数加上普通整数.
```

```
 * @param[inout] x 大整数
 * @param[in] y 32 位整数
 * @return 大整数
 */
int* bigint_add(int x[], const int y) {
    int i;

    x[0] += y;
    for (i = 0; i < MAX_LEN; i++) {
        if (x[i] >= BIGINT_RADIX) {
            x[i+1] += x[i] / BIGINT_RADIX;
            x[i] %= BIGINT_RADIX;
        }
    }
    return x;
}


/**
 * @brief 两个大整数相加, in place.
 * @param[inout] x x=x+y
 * @param[in] y y
 * @return 无
 */
void bigint_add1(int x[], const int y[]) {
    int i;
    int c = 0;

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相加 */
        const int tmp = x[i] + y[i] + c;
        x[i] = tmp % BIGINT_RADIX;
        c = tmp / BIGINT_RADIX;
    }
}

/**
 * @brief 大整数乘法, x = x*y.
 * @param[inout] x x
 * @param[in] y y
 * @return 无
 */
void bigint_mul(int x[], const int y) {
    int i;
    int c = 0; /* 保存进位 */

    for (i = 0; i < MAX_LEN; i++) { /* 用 y, 去乘以 x 的各位 */
        const int tmp = x[i] * y + c;
        x[i] = tmp % BIGINT_RADIX;
        c = tmp / BIGINT_RADIX;
    }
}


#define MAX 80   /* n,m 的最大值 */

int n, m;
int matrix[MAX][MAX];  /* 输入矩阵 */
/* f[i][j] 表示某一行区间 [i,j] 的最大值, 转移方程
 * f[i][j] = f[i+1][j]+a[i]*2^x, f[i][j-1]+a[j]*2^x, 1<=x<=m
 * 等价于 f[i][j] = 2*max(f[i+1][j]+a[i], f[i][j-1]+a[j])
 */
int f[MAX][MAX][MAX_LEN];
int tmp[MAX_LEN];
```

```
/**
 * @brief 计算单行.
 * @param[in] a 某一行
 * @return 此行能获得的最大得分
 */
int* dp(int a[MAX], int l, int r) {
    if (l == r) {
        int_to_bigint(a[l] * 2, f[l][r]);
        return f[l][r];
    }
    if (f[l][r][0] >= 0) return f[l][r];

    if (l < r) {
        memcpy(f[l][r], dp(a, l+1, r), sizeof(f[l][r]));
        bigint_mul(bigint_add(f[l][r], a[l]), 2);
        memcpy(tmp, dp(a, l, r-1), sizeof(tmp));
        bigint_mul(bigint_add(tmp, a[r]), 2);

        if (bigint_cmp(f[l][r], tmp) < 0) memcpy(f[l][r], tmp, sizeof(tmp));
    }
    return f[l][r];
}

int main() {
    int i, j, result[MAX_LEN];
    scanf("%d%d", &n, &m);

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    memset(result, 0, sizeof(result));
    for (i = 0; i < n; i++) {
        memset(f, 0, sizeof(f));
        for (j = 0; j < m; j++) {
            int k;
            for (k = 0; k < m; k++) {
                f[j][k][0] = -1;
            }
        }
        bigint_add1(result, dp(matrix[i], 0, m-1));
    }
    bigint_print(result, MAX_LEN);
    return 0;
}
```

wikioi1166_matrix_game.c

相关的题目

与本题相同的题目：

- WIKIOI 1166 矩阵取数游戏, http://www.wikioi.com/problem/1166/

与本题相似的题目：

- None

14.11 棋盘型动态规划

14.11.1 数字三角形

描述

有一个由非负整数组成的三角形，第一行只有一个数，除了最下一行之外每个数的左下角和右下角各有一个数，如图 14-2所示。

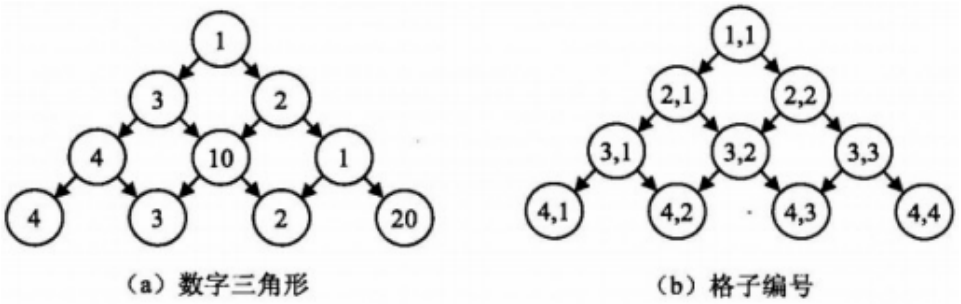


图 14-2 数字三角形问题

从第一行的数开始，每次可以往左下或右下走一格，直到走到最下行，把沿途经过的数全部加起来。如何走才能使得这个和最大?

输入

第一行是一个整数 $N(1 \leq N \leq 100)$，给出三角形的行数。接下来的 N 行给出数字三角形。三角形中的数全部是整数，范围在 0 到 100 之间。

输出

输出最大的和。

样例输入

```
5
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

样例输出

```
30
```

分析

这是一个动态决策问题，在每层有两种选择，左下或右下，因此一个 n 层的数字三角形有 2^n 条路线。

可以用回溯法，用回溯法求出所有可能的路线，就可以从中选出最优路线。但是由于有 2^n 条路线，回溯法很慢。

本题可以用动态规划来求解 (具有最有子结构和重叠子问题两个要素，后面会看到)。把当前位置 (i,j) 看成一个状态，然后定义状态 d[i][j] 为从位置 (i,j) 出发时能得到的最大和（包括格子 (i,j) 本身的值 a[i][j]）。在这个状态定义下，原问题的解是 d[0][0]。

下面来看看不同状态之间是怎样转移的。从位置 (i,j) 出发有两种决策，如果往左走，则走到 (i+1,j) 后需要求“从 (i+1,j) 出发后能得到的最大和”这一子问题，即 d[i+1][j]，类似地，往右走之后需要求 d[i+1][j+1]。应该选择 d[i+1][j] 和 d[i+1][j+1] 中较大的一个，因此可以得到如下的状态转移方程：

$$d[i][j] = a[i][j] + \max\{d[i+1][j], d[i+1][j+1]\}$$

代码

版本 1，备忘录法。

numbers_triangle1.c

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

#define MAXN 100

int n, a[MAXN][MAXN], d[MAXN][MAXN];

#define max(a,b) ((a)>(b)?(a):(b))

/**
 * @brief 求从位置 (i,j) 出发时能得到的最大和
 * @param[in] i 行
 * @param[in] j 列
 * @return 最大和
 */
int dp(const int i, const int j) {
    if(d[i][j] >= 0) {
        return d[i][j];
    } else {
        return d[i][j] = a[i][j] + (i == n-1 ? 0 : max(dp(i+1, j+1), dp(i+1, j)));
    }
}

int main() {
    int i, j;
    memset(d, -1, sizeof(d));

    scanf("%d", &n);
    for(i = 0; i < n; i++)
      for (j = 0; j <= i; j++) scanf("%d", &a[i][j]);

    printf("%d\n", dp(0, 0));
    return 0;
}
```

numbers_triangle1.c

版本 2，自底向上。

numbers_triangle2.c

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

#define MAXN 100

int n, a[MAXN][MAXN], d[MAXN][MAXN];

#define max(a,b) ((a)>(b)?(a):(b))

/**
 * @brief 自底向上计算所有子问题的最优解
 * @return 无
 */
void dp() {
    int i, j;
    for (i = 0; i < n; ++i) {
        d[n-1][i] = a[n-1][i];
    }
    for (i = n-2; i >= 0; --i)
      for (j = 0; j <= i; ++j)
        d[i][j] = a[i][j] + max(d[i+1][j], d[i+1][j+1]);
}
```

```
int main() {
    int i, j;
    memset(d, -1, sizeof(d));

    scanf("%d", &n);
    for(i = 0; i < n; i++)
      for (j = 0; j <= i; j++)
          scanf("%d", &a[i][j]);

    dp();

    printf("%d\n", d[0][0]);
    return 0;
}
```

numbers_triangle2.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①]
- POJ 1163 The Triangle, http://poj.org/problem?id=1163
- 百练 2760 数字三角形, http://poj.grids.cn/practice/2760/
- wikioi 1220 数字三角形, http://www.wikioi.com/problem/1220/

与本题相似的题目：

- None

14.11.2 过河卒

描述

如图，A 点有一个过河卒，需要走到目标 B 点。卒行走规则：可以向下、或者向右。同时在棋盘上的任一点有一个对方的马（如上图的 C 点），该马所在的点和所有跳跃一步可达的点称为对方马的控制点。例如上图 C 点上的马可以控制 9 个点（图中的 P1，P2 …P8 和 C）。卒不能通过对方马的控制点。

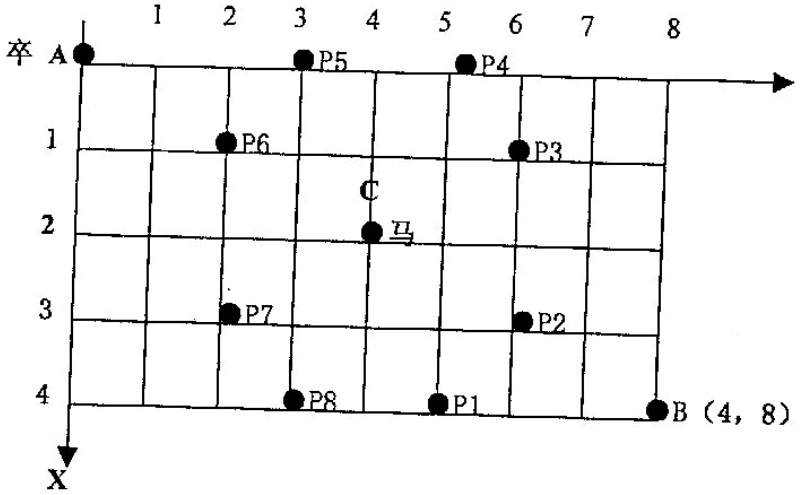


图 14-3 过河卒

棋盘用坐标表示，A 点（0，0）、B 点（n,m）(n,m 为不超过 20 的整数，并由键盘输入)，同样马的位置坐标是需要给出的（约定: C 不等于 A，同时 C 不等于 B）。现在要求你计算出卒从 A 点能够到达 B 点的路径的条数。

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

输入

B 点的坐标（n,m）以及对方马的坐标（X,Y）

输出

一个整数（路径的条数）

样例输入

```
6 6 3 2
```

样例输出

```
17
```

分析

这是一个棋盘形动态规划。

设状态为 f[i][j]，表示从 (0,0) 到 (i,j) 的路径的条数，状态转移方程为

$$f[i][j] = f[i-1][j] + f[i][j-1]$$

代码

river.c

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

#define MAX 21

int n, m;
int x, y; /* 马儿的坐标 */

/* 八个控制点，顺时针方向 */
int dx[] = {-2, -1, 1, 2, 2, 1, -1, -2};
int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};

int g[MAX][MAX];  /* 棋盘，马儿以及马儿的控制点为 1，其余为 0 */
int f[MAX][MAX];  /* f[i][j] = f[i-1][j] + f[i][j-1] */

void dp() {
    int i, j;
    memset(g, 0, sizeof(g));
    memset(f, 0, sizeof(f));

    g[x][y] = 1;
    for (i = 0; i < 8; i++) if(x+dx[i] >=0 && x+dx[i] <= n &&
            y+dy[i] >= 0 && y+dy[i] <= n){
        g[x+dx[i]][y+dy[i]] = 1;
    }

    f[0][0] = 1;  /* 起点 */
    /* 初始化边界 */
    for (i = 1; i <= n; i++) if (!g[i][0]) f[i][0] = f[i-1][0];
    for (j = 1; j <= m; j++) if (!g[0][j]) f[0][j] = f[0][j-1];

    for (i = 1; i <= n; i++) {
        for (j = 1; j <= m; j++) {
            if (!g[i][j]) f[i][j] = f[i-1][j] + f[i][j-1];
        }
    }
```

```
}

int main() {
    scanf("%d%d%d%d", &n, &m, &x, &y);
    dp();
    printf("%d\n", f[n][m]);
    return 0;
}
```

river.c

相关的题目

与本题相同的题目：

- wikioi 1010, 过河卒, http://www.wikioi.com/problem/1010/

与本题相似的题目：

- null

14.11.3 传纸条

描述

小渊和小轩是好朋友也是同班同学，他们在一起总有谈不完的话题。一次素质拓展活动中，班上同学安排做成一个 m 行 n 列的矩阵，而小渊和小轩被安排在矩阵对角线的两端，因此，他们就无法直接交谈了。幸运的是，他们可以通过传纸条来进行交流。纸条要经由许多同学传到对方手里，小渊坐在矩阵的左上角，坐标 $(1,1)$，小轩坐在矩阵的右下角，坐标 (m,n)。从小渊传到小轩的纸条只可以向下或者向右传递，从小轩传给小渊的纸条只可以向上或者向左传递。

在活动进行中，小渊希望给小轩传递一张纸条，同时希望小轩给他回复。班里每个同学都可以帮他们传递，但只会帮他们一次，也就是说如果此人在小渊递给小轩纸条的时候帮忙，那么在小轩递给小渊的时候就不会再帮忙。反之亦然。

还有一件事情需要注意，全班每个同学愿意帮忙的好感度有高有低（注意：小渊和小轩的好心程度没有定义，输入时用 0 表示），可以用一个 $0 \sim 100$ 的自然数来表示，数越大表示越好心。小渊和小轩希望尽可能找好心程度高的同学来帮忙传纸条，即找到来回两条传递路径，使得这两条路径上同学的好心程度只和最大。现在，请你帮助小渊和小轩找到这样的两条路径。

输入

输入的第一行有 2 个用空格隔开的整数 m 和 n，表示班里有 m 行 n 列 $(1 \leq m,n \leq 50)$。接下来的 m 行是一个 $m * n$ 的矩阵，矩阵中第 i 行 j 列的整数表示坐在第 i 行 j 列的学生的好心程度。每行的 n 个整数之间用空格隔开

输出

输出共一行，包含一个整数，表示来回两条路上参与传递纸条的学生的好心程度之和的最大值。

样例输入

```
3 3
0 3 9
2 8 5
5 7 0
```

样例输出

```
34
```

分析

这是一个棋盘形动态规划。用矩阵 `int g[MAX][MAX]` 存储输入数据，即每个学生的好心值。

题目等价为为从 $(1,1)$ 传两张纸条到 (m,n)。每个纸条都是由上面或左边传递过来的，所以有四种情况。设状态为 $f[i][j][k][l]$，表示纸条一在 (i,j), 纸条二在 (k,l) 的好心程度之和，状态转移方程为

$$\begin{aligned} f[i][j][k][l] = \max\{ \quad & f[i-1][j][k-1][l], \\ & f[i-1][j][k][l-1], \\ & f[i][j-1][k][l-1], \\ & f[i][j-1][k-1][l]\} \quad +g[i][j]+g[k][l] \end{aligned}$$

代码

deliver_note.c

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

#define MAX 51  /* m,n 的最大值是 50，应该有笔误，实际是 51 */

int m, n;           /* 行，列 */
int g[MAX][MAX];    /* 好心值 */
/** 题目等价为为从 (1, 1) 传两张纸条到 (m,n),
 * f[i][j][k][l] 表示纸条一在 (i,j), 纸条二在 (k,l) 的好心程度之和
 */
int f[MAX][MAX][MAX][MAX];

#define max(a,b) ((a)>(b)?(a):(b))

void dp() {
    int i, j, k, l;
    memset(f, 0, sizeof(f));
    /* 每个纸条都是由上面或左边传递过来的， 所以有四种情况 */
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < m; k++) {
                for (l = 0; l < n; l++) {
                    if(i > 0 && k > 0 && (i != k || j != l))
                        f[i][j][k][l] = max(f[i][j][k][l],
                                f[i-1][j][k-1][l] + g[i][j]+g[k][l]);
                    if(i > 0 && l < n && (i-1 != k || j != l-1))
                        f[i][j][k][l] = max(f[i][j][k][l],
                                f[i-1][j][k][l-1] + g[i][j]+g[k][l]);
                    if(j < n && l < n && (i != k || j != l))
                        f[i][j][k][l] = max(f[i][j][k][l],
                                f[i][j-1][k][l-1] + g[i][j]+g[k][l]);
                    if(j < n && k > 0 && (i != k-1 || j-1 != l))
                        f[i][j][k][l] = max(f[i][j][k][l],
                                f[i][j-1][k-1][l] + g[i][j]+g[k][l]);
                }
            }
        }
    }
}

int main() {
    int i, j;
    scanf("%d%d", &m, &n);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &g[i][j]);
```

```
            }
        }
    }
    dp();
    printf("%d\n", f[m-1][n-1][m-1][n-1]);
    return 0;
}
```

deliver_note.c

TODO: 本题可以优化为三维

相关的题目

与本题相同的题目：

- wikioi 1169, 传纸条, http://www.wikioi.com/problem/1169/

与本题相似的题目：

- null

14.11.4 骑士游历

描述

设有一个 $n * m$ 的棋盘 $(2 \leq n \leq 50, 2 \leq m \leq 50)$，如图 14-4所示，在棋盘上有一个中国象棋马。规定：

- 马只能走日字
- 马只能向右跳

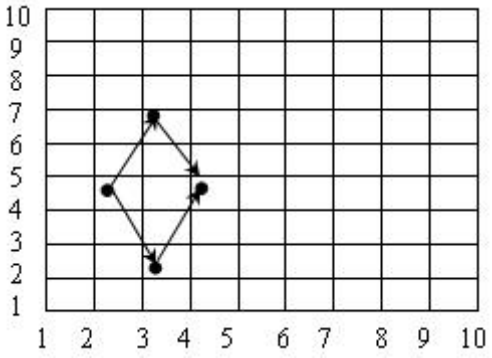


图 14-4 骑士游历

问给定起点 (x1,y1) 和终点 (x2,y2)，求出马从 (x1,y1) 出发到 (x2,y2) 的合法路径条数。

输入

第一行 2 个整数 n 和 m，第二行 4 个整数 x1,y1,x2,y2

输出

合法路径条数

样例输入

```
30 30
1 15 3 15
```

样例输出

```
2
```

分析

这是一道棋盘型动态规划。

设状态为 f[i][j]，表示从起点 (x1,y1) 到 (i,j) 的合法路径条数，状态转移方程为

$$f[i][j] = f[i-1][j-2] + f[i-1][j+2] + f[i-2][j-1] + f[i-2][j+1]$$

注意，合法路径条数有可能非常大，32 位整数存不下，需要用 64 位整数。

代码

horse.c

```
#include <stdio.h>
#include <memory.h>
#include <stdint.h>

#define MAX  51  /* n,m 最大值，本题范围又有笔误，应该是 51 */

int n, m;
int x1, y1, x2, y2;
/*f[i][j] 表示从 (x1,y1) 到 (i,j) 的合法路径的条数 */
int64_t f[MAX+1][MAX+1];

void dp() {
    int i, j;
    memset(f, 0, sizeof(f));
    f[x1][y1] = 1;
    for (i = x1+1; i <= x2; i++) {
        for (j = 1; j <= m; j++) {
            f[i][j] = f[i-1][j-2] + f[i-1][j+2] + f[i-2][j-1] + f[i-2][j+1];
        }
    }

}

int main() {
    scanf("%d%d", &n, &m);
    scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
    dp();
    printf("%lld\n", f[x2][y2]);
    return 0;
}
```

horse.c

相关的题目

与本题相同的题目：

- wikioi 1219, 骑士游历, http://www.wikioi.com/problem/1219/

与本题相似的题目：

- null

14.12 划分型动态规划

14.12.1 乘积最大

描述

设有一个长度为 N 的数字串，要求选手使用 K 个乘号将它分成 $K+1$ 个部分，找出一种分法，使得这 $K+1$ 个部分的乘积能够为最大。

举个例子：有一个数字串：312，当 $N=3$，$K=1$ 时会有以下两种分法：

- 3*12=36
- 31*2=62

这时，符合题目要求的结果是：31*2=62。

输入

输入共有两行：
第一行共有 2 个自然数 $N, K(6 \leq N \leq 40, 1 \leq K \leq 6)$
第二行是一个长度为 N 的数字串。

输出

输出最大乘积

样例输入

```
4 2
1231
```

样例输出

```
62
```

分析

首先，本题可以用区间型动态规划的思路解决。设状态为 f[i][j][k]，表示 i 到 j 这一段用 k 个乘号的最大乘积，状态转移方程如下：

$$f[i][j][k] = \max\left\{f[i][s][t] * f[s+1][j][k-t-1]\right\}, i \leq s < j, 0 \leq t < k$$

复杂度是 $O(N^3K^2)$，代码如下：

```
/** wikioi 1017 乘积最大, http://www.wikioi.com/problem/1017 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define MAXN 40
#define MAXK 6

int N, K;
char str[MAXN];

/** f[i][j][k] 表示 i 到 j 这一段用 k 个乘号的最大乘积. */
int64_t f[MAXN][MAXN][MAXK+1];


#define max(a,b) ((a)>(b)?(a):(b))

/** 计算 str[l,r] 字符串的值，int64 可以过，不用高精度. */
int64_t num(int l, int r) {
    int64_t ret = 0;
    int i;
    for (i = l; i <= r; i++) {
        ret = ret * 10 + str[i] - '0';
    }
    return ret;
}

/**
 * 区间型动态规划，复杂度是 O(n^3*k^2).
 *
```

```
 * f[i][j][k] = max(f[i][s][t] * f[s+1][j][k-t-1]),
 * i <= s < j , 0 <= t < k
 */
void dp() {
    int i, j, k, s, t;
    memset(f, 0, sizeof(f));
    for (i = 0; i < N; i++) {
        for (j = i; j < N; j++) {
            f[i][j][0] = num(i, j);
        }
    }

    for (i = 0; i < N; i++) {
        for (j = i; j < N; j++) {
            for (k = 1; k <= K; k++) {
                for (s = i; s < j; s++) {
                    for (t = 0; t < k; t++) {
                        f[i][j][k] = max(f[i][j][k],
                                f[i][s][t] * f[s + 1][j][k - t - 1]);
                    }
                }
            }
        }
    }
}

int main() {
    scanf("%d%d", &N, &K);
    scanf("%s", str);
    dp();
    printf("%lld\n", f[0][N-1][K]);
    return 0;
}
```

尽管上面的代码可以 AC 了，但还有更高效的思路。设状态为 f[k][j]，表示在区间 [0,j] 放入 k 个乘号的最大乘积，状态转移方程如下：

$$f[k][j] = \max\{f[k][j], f[k-1][i] * num(i+1, j), 0 \le i < j\}$$

复杂度是 $O(N^2K)$。代码如下。

代码

maximal_product.c

```
/** wikioi 1017 乘积最大, http://www.wikioi.com/problem/1017 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define MAXN 40
#define MAXK 6

int N, K;
char str[MAXN];

/** f[k][j] 在区间 [0,j] 放入 k 个乘号的最大乘积. */
int64_t f[MAXK+1][MAXN];


#define max(a,b) ((a)>(b)?(a):(b))

/** 计算 str[l,r] 字符串的值, int64 可以过, 不用高精度. */
int64_t num(int l, int r) {
    int64_t ret = 0;
    int i;
```

```
    for (i = l; i <= r; i++) {
        ret = ret * 10 + str[i] - '0';
    }
    return ret;
}

/** 划分型型动态规划，复杂度是 O(n^3*k^2). */
void dp() {
    int i, j, k;
    memset(f, 0, sizeof(f));
    for (j = 0; j < N; j++) {
        f[0][j] = num(0, j);
    }

    for (k = 1; k <= K; k++) {
        for (j = 0; j < N; j++) {
            for (i = 0; i < j; i++) {
                f[k][j] = max(f[k][j], f[k-1][i]*num(i+1, j));
            }
        }
    }
}

int main() {
    scanf("%d%d", &N, &K);
    scanf("%s", str);
    dp();
    printf("%lld\n", f[K][N-1]);
    return 0;
}
```

maximal_product.c

相关的题目

与本题相同的题目：

- wikioi 1017, 乘积最大, http://www.wikioi.com/problem/1017/

与本题相似的题目：

- null

14.12.2 数的划分

描述

将整数 n 分成 k 份，且每份不能为空，任意两种划分方案不能相同 (不考虑顺序)。

例如：$n=7$，$k=3$，下面三种划分方案被认为是相同的。

```
1 1 5
1 5 1
5 1 1
```

问有多少种不同的分法。

输入

$n, k(6 < n \leq 200, 2 \leq k \leq 6)$

输出

一个整数，即不同的分法。

样例输入

```
7 3
```

样例输出

```
4
```

分析

设状态为 f[k][i]，表示将整数 i 分成 k 份的方案数。有两种情况，一种划分中有至少一个 1，另外一种划分中所有的数都大于 1。对于第一种划分，去掉一个 1，就变成了一个于子问题 f[n-1][k-1]，对于第二种划分，把划分中的每个数都减去 1，就变成了一个子问题 f[k][n-k]。因此，状态转移方程如下：

$$f[k][i] = f[k-1][i-1] + f[k][i-k]$$

代码

integer_partition.c

```
/** wikioi 1039 数的划分, http://www.wikioi.com/problem/1039 */
#include <stdio.h>
#include <memory.h>

#define MAXN 300
#define MAXK 10

int N, K;

/** f[k][i] 表示将整数 i 分成 k 份的方案数. */
int f[MAXK+1][MAXN+1];

/** 划分型动态规划, 复杂度是 O(K*N). */
void dp() {
    int i, k;
    memset(f, 0, sizeof(f));
    for (i = 1; i <= N; i++) {
        f[1][i] = 1;
        if (i <= K) f[i][i] = 1;
    }

    for (k = 2; k <= K; k++) {
        for (i = k+1; i <= N; i++) {
            f[k][i] = f[k-1][i-1] + f[k][i-k];
        }
    }
}

int main() {
    scanf("%d%d", &N, &K);
    dp();
    printf("%d\n", f[K][N]);
    return 0;
}
```

integer_partition.c

相关的题目

与本题相同的题目：

- wikioi 1039, 数的划分, http://www.wikioi.com/problem/1039/

与本题相似的题目：

- None

14.13 树型动态规划

14.13.1 访问艺术馆

描述

皮尔是一个出了名的盗画者，他经过数月的精心准备，打算到艺术馆盗画。艺术馆的结构，每条走廊要么分叉为二条走廊，要么通向一个展览室。皮尔知道每个展室里藏画的数量，并且他精确地测量了通过每条走廊的时间，由于经验老道，他拿下一副画需要 5 秒的时间。你的任务是设计一个程序，计算在警察赶来之前 (警察到达时皮尔回到了入口也算)，他最多能偷到多少幅画。

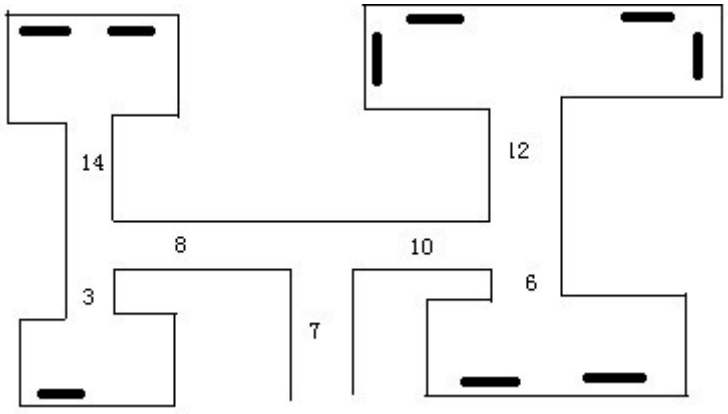


图 14-5 艺术馆

输入

第 1 行是警察赶到得时间 s($s \leq 600$)，以秒为单位。第 2 行描述了艺术馆得结构，是一串非负整数，成对地出现：每一对得第一个数是走过一条走廊得时间，第 2 个数是它末端得藏画数量；如果第 2 个数是 0，那么说明这条走廊分叉为两条另外得走廊。走廊的数目 ≤ 100。数据按照深度优先得次序给出，请看样例

输出

输出偷到得画得数量

样例输入

```
60
7 0 8 0 3 1 14 2 10 0 12 4 6 2
```

样例输出

```
2
```

分析

设状态为 f[root][time]，表示在 root 节点，剩下 time 时间内能偷到的画数，则状态转移方程如下：

$$f[root][time] = \max\{f[left][t] + f[right][ct - t], ct = time - T[root], 0 \leq t \leq ct\}$$

代码

art_gallery.c

```
/* wikioi 1163 访问艺术馆, http://www.wikioi.com/problem/1163/ */
#include<stdio.h>
#include <memory.h>
```

```
#define max(a,b) ((a)>(b)?(a):(b))
#define MAXN  1010  /* 走廊的最大数目 */

/* 输入数据，警察到达时间，走廊时间，藏画数量 */
int LIMIT, T[MAXN], P[MAXN];
/* 二叉树，节点个数，左孩子，右孩子 */
int n, L[MAXN], R[MAXN]; /* 0 位置未用 */

/* f[root][time] 表示在 root 节点，剩下 time 时间内能偷到的画数 */
int f[MAXN][MAXN];

/**
 * @brief 输入的数据是前序遍历序列，依次读入，递归建树
 * @return 根节点
 */
int build_tree() {
    n++;
    scanf("%d%d", &T[n], &P[n]);
    T[n] *= 2; /* 走廊花费时间要 *2，因为还要出来 */
    int now = n; /* 当前节点 */
    if (P[now] == 0) {
        L[now] = build_tree();  /* 建立节点 now 的左子树 */
        R[now] = build_tree();  /* 建立节点 now 的右子树 */
    }
    return now;
}

/**
 * 备忘录法.
 * @param[in] root 开始节点
 * @param[in] 剩余时间
 * @return 偷到得画的数量
 */
int dp(int root, int time) {
    int t;
    if (f[root][time] != -1)
        return f[root][time];
    if (!time)
        return f[root][time] = 0;

    const int ct = time - T[root]; /* 留给左右子节点的时间 */

    /* 到达叶子节点，即展览室 */
    if (!L[root] && !R[root]) {
        /* 不能 tt / 5，因为时间够，但是可能画不够 */
        if (P[root] * 5 <= ct)
            return f[root][time] = P[root];
        else
            return f[root][time] = ct / 5;
    }

    f[root][time] = 0;
    for (t = 0; t <= ct; t++) {
        int lp = dp(L[root], t);
        int rp = dp(R[root], ct - t);
        f[root][time] = max(f[root][time], lp + rp);
    }
    return f[root][time];
}

int main() {
    scanf("%d", &LIMIT);
    build_tree();
    memset(f, -1, sizeof(f));
```

```
    printf("%d\n", dp(1, LIMIT));
    return 0;
}
```

art_gallery.c

相关的题目

与本题相同的题目：

- wikioi 1163, 访问艺术馆, http://www.wikioi.com/problem/1163/

与本题相似的题目：

- None

14.13.2 没有上司的舞会

描述

Ural 大学有 N 个职员，编号为 $1 \sim N$。他们有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。每个职员有一个快乐指数。现在有个周年庆宴会，要求与会职员的快乐指数最大。但是，没有职员愿和直接上司一起与会。

输入

第一行一个整数 $N(1 \leq N \leq 6000)$。
接下来 N 行，第 $i+1$ 行表示 i 号职员的快乐指数 $R_i(-128 \leq R_i \leq 127)$。
接下来 $N-1$ 行，每行输入一对整数 L, K。表示 K 是 L 的直接上司。
最后一行输入 0,0，表示结束。

输出

输出最大的快乐指数

样例输入

```
7
1
1
1
1
1
1
1
1 3
2 3
6 4
7 4
4 5
3 5
0 0
```

样例输出

```
5
```

分析

设状态为 f[k][0] 和 f[k][1]，f[k][0] 表示第 i 个人不参加时的最大值，f[k][1] 表示第 k 个人参加的最大值，则状态转移方程如下：

$$
\begin{aligned}
f[k][0] &= \sum_{l} \max\{f[l][0], f[l][1]\}\text{，其中 k 是 l 的直接上司} \\
f[k][1] &= \sum_{l} f[l][0] + r[k]
\end{aligned}
$$

代码

ball.c

```
/* wikioi 1380 没有上司的舞会, http://www.wikioi.com/problem/1380 */
#include<stdio.h>
#include<string.h>

#define MAXN 6001 /* 0 位置未用 */
#define max(a,b) ((a)>(b)?(a):(b))
typedef char bool;

int n;
/* 快乐指数 */
int r[MAXN];

/* 静态链表节点. */
typedef struct edge_t {
    int v;
    int prev; /* 上一条边, 倒着串起来 */
} edge_t;

/* 多个静态链表在一个数组里, 实质上是树的孩子表示法 */
edge_t edge[MAXN-1];

/* head[root] 指向表头, 即最后一条边 */
int head[MAXN];

/* has_boss[i] 表示第 i 个人是否有上司 */
bool has_boss[MAXN];

int f[MAXN][2];

int cnt;  /* 边个数-1 */
void add_edge(int u, int v) {
    edge[cnt].v = v;
    edge[cnt].prev = head[u];
    head[u] = cnt++;
}

void dp(int k) {
    int p;
    f[k][1] = r[k];
    f[k][0] = 0;
    for (p = head[k]; p != -1; p = edge[p].prev) {
        int l = edge[p].v;
        dp(l);
        f[k][1] += f[l][0];
        f[k][0] = f[k][0] + max(f[l][1],f[l][0]);
    }
}

int solve() {
    int k;
    memset(f, 0, sizeof(f));
```

```
    for (k = 1; k <= n; k++) if (!has_boss[k]) {
        break;
    }

    dp(k);
    return max(f[k][0], f[k][1]);
}

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

    cnt = 0;
    memset(has_boss, 0, sizeof(has_boss));
    memset(head, -1, sizeof(head));
    int x, y;
    for (i = 1; i < n; i++) {
        scanf("%d%d", &x, &y);
        has_boss[x] = 1;
        add_edge(y, x);
    }
    scanf("%d%d", &x, &y);

    printf("%d\n", solve());
    return 0;
}
```

ball.c

相关的题目

与本题相同的题目：

- wikioi 1380, 没有上司的舞会, http://www.wikioi.com/problem/1380/

与本题相似的题目：

- None

14.14 最大子矩形

在一个给定的矩形网格中有一些障碍点，要找出网格内部不包含任何障碍点，且边界与坐标轴平行的最大子矩形。
遇到求矩形面积，一般把左上角设置为坐标原点，这与数学中的坐标系不同。
解决方法参考“浅谈用极大化思想解决最大子矩形问题”，http://wenku.baidu.com/view/728cd5126edb6f1aff001fbb.html
方法一是一种暴力枚举法，方法二是一种动规法。

14.14.1 奶牛浴场

描述

由于 john 建造了牛场围栏，激起了奶牛的愤怒，奶牛的产奶量急剧减少。为了讨好奶牛，john 决定在牛场中建造一个大型浴场。但是 john 的奶牛有一个奇怪的习惯，每头奶牛都必须在牛场中的一个固定的位置产奶，而奶牛显然不能在浴场中产奶，于是，john 希望所建造的浴场不覆盖这些产奶点。这回，他又要求助于 clevow 了。你还能帮助 clevow 吗?

john 的牛场和规划的浴场都是矩形。浴场要完全位于牛场之内，并且浴场的轮廓要与牛场的轮廓平行或者重合。浴场不能覆盖任何产奶点，但是产奶点可以位于浴场的轮廓上。

clevow 当然希望浴场的面积尽可能大了，所以你的任务就是帮她计算浴场的最大面积。

输入

输入文件的第一行包含两个整数 L 和 W($1 \leq L, W \leq 30000$)，分别表示牛场的长和宽。文件的第二行包含一个整数 n($0 \leq n \leq 5000$)，表示产奶点的数量。以下 n 行每行包含两个整数 x 和 y，表示一个产奶点的坐标。所有产奶点都位于牛场内，即：$0 \leq x \leq l, 0 <\leq y \leq w$。

输出

输出文件仅一行，包含一个整数 S，表示浴场的最大面积。

样例输入

```
10 10
4
1 1
9 1
1 9
9 9
```

样例输出

```
80
```

分析

这里使用方法二，需要先做一定的预处理。由于第二种算法复杂度与牛场的面积有关，而题目中牛场的面积很大（30000×30000），因此需要对数据进行离散化处理。离散化后矩形的大小降为 S×S，所以时间复杂度为 O(S2)，空间复杂度为 O(S)。需要注意的是，为了保证算法能正确执行，把 (0,0) 和 (m,n) 设置为产奶点，相当于加上了一个“虚拟边界”。

代码

cow_bath.c

```
/**
 * OJ: https://vijos.org/p/1055
 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXN 5001

#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))

int L, W; /* 长，竖向 x 坐标，宽，横向 y 坐标 */
int n; /* n 个产奶点 */
int x[MAXN], y[MAXN]; /* 产奶点坐标 */

int int_cmp(const void *a, const void *b) {
    const int *ia = (const int*) a;
    const int *ib = (const int*) b;
    return *ia - *ib;
}

/* 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 "binary_search.c"

/**
 * @brief 求最大子矩形
 *
 * 参考 http://wenku.baidu.com/view/728cd5126edb6f1aff001fbb.html 的第二种方法
```

```
 *
 * @param[in] L 长度，纵向
 * @param[in] W 宽度，横向
 * @param[in] x 纵向 x 坐标
 * @param[in] y 横向 y 坐标
 * @param[in] n 障碍点的个数
 * @return 最大子矩形的面积
 */
int max_submatrix(const int L, const int W,
        const int x[], const int y[], int n) {
    int *a = (int*) malloc((n + 1) * sizeof(int));
    int *b = (int*) malloc((n + 1) * sizeof(int));
    int la = n, lb = n;
    int i, j;
    int lm, rm; // 左边界，右边界
    int result = 0, temp;
    int **v;  // 新的 01 矩阵，1 表示障碍点
    int *h;  // 高度
    int *l;  // 左边界
    int *r;  // 右边界

    memcpy(a, x, n * sizeof(int));
    memcpy(b, y, n * sizeof(int));
    a[la++] = 0;
    b[lb++] = 0;
    a[la++] = L;
    b[lb++] = W;

    // 去重
    qsort(a, la, sizeof(int), int_cmp);
    qsort(b, lb, sizeof(int), int_cmp);
    for (j = 1, i = 1; i < la; i++) { // 去重
        if (a[i] != a[i - 1])
            a[j++] = a[i];
    }
    la = j;
    for (j = 1, i = 1; i < lb; i++) { // 去重
        if (b[i] != b[i - 1])
            b[j++] = b[i];
    }
    lb = j;
    h = (int*) malloc(lb * sizeof(int));
    l = (int*) malloc(lb * sizeof(int));
    r = (int*) malloc(lb * sizeof(int));
    //******************************//

    // 计算 v 矩阵
    v = (int**) malloc(la * sizeof(int*));
    for (i = 0; i < la; i++) {
        v[i] = (int*) calloc(lb, sizeof(int));
    }
    for (i = 0; i < n; i++) {
        int ia, ib;
        ia = binary_search(a, la, x[i]);
        ib = binary_search(b, lb, y[i]);
        v[ia][ib] = 1; //标记障碍点
    }

    // 初始化
    for (i = 0; i < lb; i++) {
        l[i] = 0;
        r[i] = W;
        h[i] = 0;
    }
```

```
    for (i = 1; i < la; i++) { // 从上到下
        lm = 0;
        for (j = 0; j < lb; j++) { // 从左到右计算 l[j]
            if (!v[i - 1][j]) {  //如果上一个不是障碍点
                h[j] = h[j] + a[i] - a[i - 1]; //高度累加
                // l[i][j]=max(l[i-1][j] , 左边第一个障碍点 (i-1,j) 的位置)
                l[j] = max(l[j], lm);
            } else {  //如果上一个点是障碍点
                h[j] = a[i] - a[i - 1]; //高度重新计算
                l[j] = 0;
                r[j] = W;
                lm = b[j]; //更新 (i-1,j) 左边第一个障碍点的位置
            }
        }
        rm = W;
        for (j = lb - 1; j >= 0; j--) {  //  从右到左计算 r[j]
            //r[i][j]=min(r[i-1][j] , (i-1,j) 右边第一个障碍点的位置)
            r[j] = min(r[j], rm);
            temp = h[j] * (r[j] - l[j]);
            result = max(result, temp);  //计算最优解
            if (v[i - 1][j]) //如果该点是障碍点，更新 (i-1,j) 右边第一个障碍点的位置
                rm = b[j];
        }
    }
    // 计算横条的面积
    for (i = 1; i < la; i++) {
        temp = W * (a[i] - a[i - 1]);
        result = max(result, temp);
    }

    free(a);
    free(b);
    for (i = 0; i < la; i++) {
        free(v[i]);
    }
    free(v);
    free(h);
    free(l);
    free(r);
    return result;
}

int main() {
    int i;
    while (scanf("%d%d", &L, &W) == 2) {
        scanf("%d", &n);
        for (i = 0; i < n; i++) {
            scanf("%d%d", &x[i], &y[i]);
        }

        printf("%d\n", max_submatrix(L, W, x, y, n));
    }
    return 0;
}
```

cow_bath.c

相关的题目

与本题相同的题目：

- Vijos 1055 奶牛浴场, https://vijos.org/p/1055

与本题相似的题目：

- LeetCode Maximal Rectangle, http://leetcode.com/onlinejudge#question_85，
  参考代码 https://gist.github.com/soulmachine/b20a15009450016038d9
- POJ 3494 Largest Submatrix of All 1's, http://poj.org/problem?id=3494

14.14.2 最大全 1 子矩阵

描述

给定一个 $m \times n$ 的 01 矩阵，求最大的全 1 子矩阵。

输入

输入包含多组测试用例。每组测试用例第一行包含两个整数 m 和 n($1 \leq m, n \leq 2000$)，接下来是 m 行数据，每行 n 个元素。

输出

对每个测试用例，输出最大全 1 子矩阵的 1 的个数。如果输入的矩阵是全 0，则输出 0.

样例输入

```
2 2
0 0
0 0
4 4
0 0 0 0
0 1 1 0
0 1 1 0
0 0 0 0
```

样例输出

```
0
4
```

分析

注意，上一题算的是面积，这一题算的是个数，在某些细节上处理不同。

代码

largest_rectangle.c

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

#define max(a,b)  (a > b ? a : b)
#define min(a,b)  (a < b ? a : b)

#define  MAXN 2001

int matrix[MAXN][MAXN];

int lagest_rectangle(/*int **matrix, */int m, int n) {
    int i, j;
    int *H = (int*) malloc(n * sizeof(int));  // 高度
    int *L = (int*) malloc(n * sizeof(int));  // 左边界
    int *R = (int*) malloc(n * sizeof(int));  // 右边界
    int ret = 0;
```

```
    memset(H, 0, n * sizeof(int));
    memset(L, 0, n * sizeof(int));
    for (i = 0; i < n; i++) R[i] = n;

    for (i = 0; i < m; ++i) {
        int left = 0, right = n;
        // calculate L(i, j) from left to right
        for (j = 0; j < n; ++j) {
            if (matrix[i][j] == 1) {
                ++H[j];
                L[j] = max(L[j], left);
            } else {
                left = j + 1;
                H[j] = 0;
                L[j] = 0;
                R[j] = n;
            }
        }
        // calculate R(i, j) from right to left
        for (j = n - 1; j >= 0; --j) {
            if (matrix[i][j] == 1) {
                R[j] = min(R[j], right);
                ret = max(ret, H[j] * (R[j] - L[j]));
            } else {
                right = j;
            }
        }
    }

    return ret;
}

int main() {
    int m, n;
    int i, j;
    while (scanf("%d%d", &m, &n) > 0) {
        for (i = 0; i < m; i++) {
            for (j = 0; j < n; j++) {
                scanf("%d", &matrix[i][j]);
            }
        }

        printf("%d\n", lagest_rectangle(m, n));
    }
    return 0;
}
```

largest_rectangle.c

相关的题目

与本题相同的题目：

- POJ 3494 Largest Submatrix of All 1's, http://poj.org/problem?id=3494
- LeetCode Maximal Rectangle, http://leetcode.com/onlinejudge#question_85，
  参考代码 https://gist.github.com/soulmachine/b20a15009450016038d9

与本题相似的题目：

- Vijos 1055 奶牛浴场, https://vijos.org/p/1055

14.15 Triangle

描述

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
[2],
[3,4],
[6,5,7],
[4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note: Bonus point if you are able to do this using only $O(n)$ extra space, where n is the total number of rows in the triangle.

分析

设状态为 $f(i,j)$，表示从从位置 (i,j) 出发，路径的最小和，则状态转移方程为

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

代码

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

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

相关题目

- 无

14.16 Maximum Subarray

描述

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array `[-2,1,-3,4,-1,2,1,-5,4]`, the contiguous subarray `[4,-1,2,1]` has the largest `sum = 6`.

分析

最大连续子序列和，非常经典的题。

当我们从头到尾遍历这个数组的时候，对于数组里的一个整数，它有几种选择呢？它只有两种选择：1、加入之前的 SubArray；2. 自己另起一个 SubArray。那什么时候会出现这两种情况呢？

如果之前 SubArray 的总体和大于 0 的话，我们认为其对后续结果是有贡献的。这种情况下我们选择加入之前的 SubArray

如果之前 SubArray 的总体和为 0 或者小于 0 的话，我们认为其对后续结果是没有贡献，甚至是有害的（小于 0 时）。这种情况下我们选择以这个数字开始，另起一个 SubArray。

设状态为 `f[j]`，表示以 `S[j]` 结尾的最大连续子序列和，则状态转移方程如下：

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

解释如下：

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

其他思路：

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

动规

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

思路 5

```
// LeetCode, Maximum Subarray
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    public:
    int maxSubArray(int A[], int n) {
        return mcss(A, n);
    }
    private:
    // 思路 5，求最大连续子序列和
    static int mcss(int A[], int n) {
        int i, result, cur_min;
        int *sum = new int[n + 1];  // 前 n 项和

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

相关题目

- Binary Tree Maximum Path Sum，见 §??

14.17 Palindrome Partitioning II

描述

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

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

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

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

分析

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

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

这是一个二维函数，实际写代码比较麻烦。

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

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

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

定义状态 `P[i][j] = true if [i,j]` 为回文，那么

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

代码

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

相关题目

- Palindrome Partitioning，见 §??

14.18 Maximal Rectangle

描述

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

分析

无

代码

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

        const int m = matrix.size();
        const int n = matrix[0].size();
        vector<int> H(n, 0);
        vector<int> L(n, 0);
        vector<int> R(n, n);

        int ret = 0;
        for (int i = 0; i < m; ++i) {
            int left = 0, right = n;
            // calculate L(i, j) from left to right
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == '1') {
                    ++H[j];
                    L[j] = max(L[j], left);
                } else {
                left = j+1;
                H[j] = 0; L[j] = 0; R[j] = n;
            }
        }
        // calculate R(i, j) from right to left
        for (int j = n-1; j >= 0; --j) {
            if (matrix[i][j] == '1') {
                R[j] = min(R[j], right);
                ret = max(ret, H[j]*(R[j]-L[j]));
            } else {
            right = j;
        }
    }
}
return ret;
}
};
```

相关题目

- 无

14.19 Best Time to Buy and Sell Stock III

描述

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

分析

设状态 $f(i)$，表示区间 $[0,i](0 \le i \le n-1)$ 的最大利润，状态 $g(i)$，表示区间 $[i,n-1](0 \le i \le n-1)$ 的最大利润，则最终答案为 $\max\{f(i)+g(i)\}, 0 \le i \le n-1$。

允许在一天内买进又卖出，相当于不交易，因为题目的规定是最多两次，而不是一定要两次。

将原数组变成差分数组，本题也可以看做是最大 m 子段和，$m=2$，参考代码：https://gist.github.com/soulmachine/5906637

代码

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

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

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

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

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

        return max_profit;
    }
};
```

相关题目

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

14.20 Interleaving String

描述

Given $s1, s2, s3$, find whether $s3$ is formed by the interleaving of $s1$ and $s2$.

For example, Given: `s1 = "aabcc", s2 = "dbbca"`,

When `s3 = "aadbbcbcac"`, return true.

When `s3 = "aadbbbaccc"`, return false.

分析

设状态 f[i][j]，表示 s1[0,i] 和 s2[0,j]，匹配 s3[0, i+j]。如果 s1 的最后一个字符等于 s3 的最后一个字符，则 f[i][j]=f[i-1][j]；如果 s2 的最后一个字符等于 s3 的最后一个字符，则 f[i][j]=f[i][j-1]。因此状态转移方程如下：

```
f[i][j] = (s1[i - 1] == s3 [i + j - 1] && f[i - 1][j])
|| (s2[j - 1] == s3 [i + j - 1] && f[i][j - 1]);
```

递归

```
// LeetCode, Interleaving String
// 递归，会超时，仅用来帮助理解
class Solution {
    public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s3.length() != s1.length() + s2.length())
        return false;

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

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

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

动规

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

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

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

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

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

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

```
    }
};
```

动规 + 滚动数组

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

        if (s1.length() < s2.length())
        return isInterleave(s2, s1, s3);

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

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

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

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

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

相关题目

- 无

14.21 Scramble String

描述

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

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

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

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

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

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

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

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

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

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

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

分析

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

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

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

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

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

递归

```
// LeetCode, Interleaving String
// 递归，会超时，仅用来帮助理解
// 时间复杂度 O(n^6)，空间复杂度 O(1)
class Solution {
    public:
    bool isScramble(string s1, string s2) {
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
    private:
    typedef string::iterator Iterator;
    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

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

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

        return false;
    }
};
```

动规

```
// LeetCode, Interleaving String
// 动规，时间复杂度 O(n^3)，空间复杂度 O(n^3)
class Solution {
    public:
```

```
    bool isScramble(string s1, string s2) {
        const int N = s1.size();
        if (N != s2.size()) return false;

        // f[n][i][j], 表示长度为 n, 起点为 s1[i] 和
        // 起点为 s2[j] 两个字符串是否互为 scramble
        bool f[N + 1][N][N];
        fill_n(&f[0][0][0], (N + 1) * N * N, false);

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

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

递归 + 剪枝

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

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

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

        return false;
    }
};
```

备忘录法

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

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

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

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

        return false;
    }

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

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

备忘录法

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

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

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

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

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

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

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

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

        return false;
    }

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

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

相关题目

- 无

14.22 Minimum Path Sum

描述

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

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

分析

跟 Unique Paths (见 §??) 很类似。

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

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

备忘录法

```
// LeetCode, Minimum Path Sum
// 备忘录法
class Solution {
    public:
    int minPathSum(vector<vector<int> > &grid) {
        const int m = grid.size();
```

```
        const int n = grid[0].size();
        this->f = vector<vector<int> >(m, vector<int>(n, -1));
        return dfs(grid, m-1, n-1);
    }
    private:
    vector<vector<int> > f;  // 缓存

    int dfs(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX; // 越界，终止条件，注意，不是 0

        if (x == 0 && y == 0) return grid[0][0]; // 回到起点，收敛条件

        return min(getOrUpdate(grid, x - 1, y),
        getOrUpdate(grid, x, y - 1)) + grid[x][y];
    }

    int getOrUpdate(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX; // 越界，注意，不是 0
        if (f[x][y] >= 0) return f[x][y];
        else return f[x][y] = dfs(grid, x, y);
    }
};
```

动规

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

        int f[m][n];
        f[0][0] = grid[0][0];
        for (int i = 1; i < m; i++) {
            f[i][0] = f[i - 1][0] + grid[i][0];
        }
        for (int i = 1; i < n; i++) {
            f[0][i] = f[0][i - 1] + grid[0][i];
        }

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

动规 + 滚动数组

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

        int f[n];
        fill(f, f+n, INT_MAX); // 初始值是 INT_MAX，因为后面用了 min 函数。
```

```
            f[0] = 0;

            for (int i = 0; i < m; i++) {
                f[0] += grid[i][0];
                for (int j = 1; j < n; j++) {
                    // 左边的 f[j]，表示更新后的 f[j]，与公式中的 f[i[[j] 对应
                    // 右边的 f[j]，表示老的 f[j]，与公式中的 f[i-1][j] 对应
                    f[j] = min(f[j - 1], f[j]) + grid[i][j];
                }
            }
            return f[n - 1];
        }
    };
```

相关题目

- Unique Paths, 见 §??
- Unique Paths II, 见 §??

14.23 Edit Distance

描述

Given two words `word1` and `word2`, find the minimum number of steps required to convert `word1` to `word2`. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

分析

设状态为 `f[i][j]`，表示 `A[0,i]` 和 `B[0,j]` 之间的最小编辑距离。设 `A[0,i]` 的形式是 `str1c`，`B[0,j]` 的形式是 `str2d`，

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

动规

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

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

动规 + 滚动数组

```
    // LeetCode, Edit Distance
    // 二维动规 + 滚动数组
    // 时间复杂度 O(n*m), 空间复杂度 O(n)
    class Solution {
        public:
        int minDistance(const string &word1, const string &word2) {
            if (word1.length() < word2.length())
            return minDistance(word2, word1);

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

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

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

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

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

                    upper_left = upper;
                }
            }

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

相关题目

- 无

14.24 Decode Ways

描述

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.

For example, Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

分析

跟 Climbing Stairs (见 §2.1.19) 很类似，不过多加一些判断逻辑。

代码

```
// LeetCode, Decode Ways
// 动规，时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
    public:
    int numDecodings(const string &s) {
        if (s.empty() || s[0] == '0') return 0;

        int prev = 0;
        int cur = 1;
        // 长度为 n 的字符串，有 n+1 个阶梯
        for (size_t i = 1; i <= s.size(); ++i) {
            if (s[i-1] == '0') cur = 0;

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

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

相关题目

- Climbing Stairs, 见 §2.1.19

14.25 Distinct Subsequences

描述

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example: S = "rabbbit", T = "rabbit"

Return 3.

分析

设状态为 $f(i,j)$，表示 T[0,j] 在 S[0,i] 里出现的次数。首先，无论 S[i] 和 T[j] 是否相等，若不使用 S[i]，则 $f(i,j)=f(i-1,j)$；若 S[i]==T[j]，则可以使用 S[i]，此时 $f(i,j)=f(i-1,j)+f(i-1,j-1)$。

代码

```
// LeetCode, Distinct Subsequences
// 二维动规 + 滚动数组
// 时间复杂度 O(m*n)，空间复杂度 O(n)
class Solution {
    public:
    int numDistinct(const string &S, const string &T) {
        vector<int> f(T.size() + 1);
        f[0] = 1;
        for (int i = 0; i < S.size(); ++i) {
            for (int j = T.size() - 1; j >= 0; --j) {
                f[j + 1] += S[i] == T[j] ? f[j] : 0;
            }
        }

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

相关题目

- 无

14.26 Word Break

描述

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

For example, given

s = "leetcode",

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

Return true because "leetcode" can be segmented as "leet code".

分析

设状态为 $f(i)$，表示 s[0,i] 是否可以分词，则状态转移方程为

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

深搜

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

```
            return false;
        }
    };
```

动规

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

相关题目

- Word Break II, 见 §??

14.27 Word Break II

描述

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

s = `"catsanddog"`,

dict = `["cat", "cats", "and", "sand", "dog"]`.

A solution is `["cats and dog", "cat sand dog"]`.

分析

在上一题的基础上，要返回解本身。

代码

```
    // LeetCode, Word Break II
    // 动规，时间复杂度 O(n^2)，空间复杂度 O(n^2)
    class Solution {
        public:
        vector<string> wordBreak(string s, unordered_set<string> &dict) {
            // 长度为 n 的字符串有 n+1 个隔板
            vector<bool> f(s.length() + 1, false);
            // prev[i][j] 为 true，表示 s[j, i) 是一个合法单词，可以从 j 处切开
            // 第一行未用
            vector<vector<bool> > prev(s.length() + 1, vector<bool>(s.length()));
            f[0] = true;
```

```
            for (size_t i = 1; i <= s.length(); ++i) {
                for (int j = i - 1; j >= 0; --j) {
                    if (f[j] && dict.find(s.substr(j, i - j)) != dict.end()) {
                        f[i] = true;
                        prev[i][j] = true;
                    }
                }
            }
            vector<string> result;
            vector<string> path;
            gen_path(s, prev, s.length(), path, result);
            return result;

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

相关题目

- Word Break, 见 §??

第 15 章

图

无向图的节点定义如下：

```
// 无向图的节点
struct UndirectedGraphNode {
    int label;
    vector<UndirectedGraphNode *> neighbors;
    UndirectedGraphNode(int x) : label(x) {};
};
```

稠密图适合用邻接矩阵来表示。

am_graph.cpp

```
/** 顶点数最大值. */
const int MAX_NV = 100;

/** 边的权值类型, 可以为 int, float, double. */
typedef int graph_weight_t;
const graph_weight_t GRAPH_INF = INT_MAX;

/**
 * @struct 图, 用邻接矩阵 (Adjacency Matrix).
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接矩阵, 存放边的信息, 如权重等
    graph_weight_t matrix[MAX_NV][MAX_NV];
};
```

am_graph.cpp

稀疏图适合用邻接表来表示。

al_graph.cpp

```
/** 边的权值类型, 可以为 int, float, double. */
typedef int graph_weight_t;

/** 顶点的编号, 可以为 char, int, string 等. */
typedef char graph_vertex_id_t;

/**
 * @struct 图, 用邻接表 (Adjacency List).
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接表, 存放边的信息, 如权重等
    map<graph_vertex_id_t, map<graph_vertex_id_t, graph_weight_t> > matrix;
};
```

al_graph.cpp

15.1 图的深搜

图的深度优先搜索的代码框架如下：

graph_dfs.cpp
```
/**
 * @brief 图的深度优先搜索代码框架，搜索边.
 * @param[in] g 图
 * @param[in] u 出发顶点
 * @param[in] visited 边的访问历史记录
 * @return 无
 * @remark 在使用的时候，为了降低递归的内存占用量，可以把
 * g, visited 抽出来作为全局变量
 */
void dfs(const graph_t &g, int u, bool visited[][MAX_NV]) {
    for(int v = 0;  v < g.nv; v++) if(g.matrix[u][v] && !visited[u][v]) {
        visited[u][v] = visited[v][u] = true; // 无向图用这句
        // visited_edges[u][v] = true; // 有向图用这句
        dfs(g, v, visited);
        // 这里写逻辑代码
        // printf("%d %d\n", u, v);
    }
}

/**
 * @brief 图的深度优先搜索代码框架，搜索顶点.
 * @param[in] g 图
 * @param[in] u 出发顶点
 * @param[in] visited 顶点的访问历史记录
 * @return 无
 * @remark 在使用的时候，为了降低递归的内存占用量，可以把
 * g, visited 抽出来作为全局变量
 */
void dfs(const graph_t &g, int u, bool visited[MAX_NV]) {
    visited[u] = true;
    for(int v = 0;  v < g.nv; v++) if(g.matrix[u][v] && !visited[v]) {
        dfs(g, v, visited);
        // 这里写逻辑代码
        // printf("%d %d\n", u, v);
    }
}
```
graph_dfs.cpp

15.1.1 Satellite Photographs

描述

Farmer John purchased satellite photos of $W \times H$ pixels of his farm ($1 \leq W \leq 80, 1 \leq H \leq 1000$) and wishes to determine the largest 'contiguous' (connected) pasture. Pastures are contiguous when any pair of pixels in a pasture can be connected by traversing adjacent vertical or horizontal pixels that are part of the pasture. (It is easy to create pastures with very strange shapes, even circles that surround other circles.)

Each photo has been digitally enhanced to show pasture area as an asterisk ('*') and non-pasture area as a period ('.'). Here is a 10×5 sample satellite photo:

```
..*.....**
.**..*****
.*...*....
..****.***
..****.***
```

This photo shows three contiguous pastures of 4, 16, and 6 pixels. Help FJ find the largest contiguous pasture in each of his satellite photos.

输入

Line 1: Two space-separated integers: W and H

Lines 2..H+1: Each line contains W "*" or "." characters representing one raster line of a satellite photograph.

输出

Line 1: The size of the largest contiguous field in the satellite photo.

样例输入

```
10 5
..*.....**
.**..*****
.*...*....
..****.***
..****.***
```

样例输出

```
16
```

分析

这是一个平面的二维地图，把地图上的每个点当成隐式图上的一个顶点，每个顶点有上下左右四个邻接点。在这个隐式图上进行深搜。

代码

satellite_photographs.c

```
/* POJ 3051 Satellite Photographs, http://poj.org/problem?id=3051 */
#include <stdio.h>
#include <string.h>

#define MAXH 1000
#define MAXW 80

int H, W; /* H 行 W 列 */
char map[MAXH+2][MAXW+2];/* 上下左右加一圈'.' 可以防止越界 */

int count;

void dfs(int x, int y) {
    /* 加了一圈'.' 可以防止越界，因此不需要判断越界 */
    if (map[x][y] == '.') return;

    map[x][y] = '.';  /* 标记 (x,y) 已访问过，起到去重作用 */
    count++;
    dfs(x + 1, y);
    dfs(x - 1, y);
    dfs(x, y + 1);
    dfs(x, y - 1);
}

int main() {
    int i, j, max;
    memset(map, '.', sizeof(map));

    scanf("%d%d", &W, &H); /* H 是行数，W 是列数 */
    for(i = 1; i <= H; ++i) {
        char line[MAXW+1];
        scanf("%s", line);
        strncpy(&map[i][1], line, W);
    }
```

```
    max = 0;
    for (i = 1; i <= H; i++) {
        for (j = 1; j <= W; j++) {
            if (map[i][j] == '*') {
                count = 0;
                dfs(i, j);
            }
            if (count > max) max = count;
        }
    }
    printf("%d\n", max);
    return 0;
}
```

satellite_photographs.c

相关的题目

与本题相同的题目：

- POJ 3051 Satellite Photographs, http://poj.org/problem?id=3051

与本题相似的题目：

- POJ 3620 Avoid The Lakes, http://poj.org/problem?id=3620
  参考代码 https://gist.github.com/soulmachine/6761537

15.1.2 John's trip

Description

Little Johnny has got a new car. He decided to drive around the town to visit his friends. Johnny wanted to visit all his friends, but there was many of them. In each street he had one friend. He started thinking how to make his trip as short as possible. Very soon he realized that the best way to do it was to travel through each street of town only once. Naturally, he wanted to finish his trip at the same place he started, at his parents' house.

The streets in Johnny's town were named by integer numbers from 1 to $n, n < 1995$. The junctions were independently named by integer numbers from 1 to $m, m <= 44$. No junction connects more than 44 streets. All junctions in the town had different numbers. Each street was connecting exactly two junctions. No two streets in the town had the same number. He immediately started to plan his round trip. If there was more than one such round trip, he would have chosen the one which, when written down as a sequence of street numbers is lexicographically the smallest. But Johnny was not able to find even one such round trip.

Help Johnny and write a program which finds the desired shortest round trip. If the round trip does not exist the program should write a message. Assume that Johnny lives at the junction ending the street appears first in the input with smaller number. All streets in the town are two way. There exists a way from each street to another street in the town. The streets in the town are very narrow and there is no possibility to turn back the car once he is in the street

Input

Input file consists of several blocks. Each block describes one town. Each line in the block contains three integers $x; y; z$, where $x > 0$ and $y > 0$ are the numbers of junctions which are connected by the street number z. The end of the block is marked by the line containing $x = y = 0$. At the end of the input file there is an empty block, $x = y = 0$.

Output

Output one line of each block contains the sequence of street numbers (single members of the sequence are separated by space) describing Johnny's round trip. If the round trip cannot be found the corresponding output block contains the message "Round trip does not exist."

Sample Input

```
1 2 1
2 3 2
3 1 6
1 2 5
2 3 3
3 1 4
0 0
1 2 1
2 3 2
1 3 3
2 4 4
0 0
0 0
```

Sample Output

```
1 2 3 5 4 6
Round trip does not exist.
```

分析

欧拉回路。

如果能从图的某一顶点出发，每条边恰好经过一次，这样的路线称为**欧拉道路** (Eulerian Path)。如果还能够回到起点，这样的路线称为**欧拉回路** (Eulerian Circuit)。

对于无向图 G，当且仅当 G 是连通的，且最多有两个奇点，则存在欧拉道路。如果有两个奇点，则必须从其中一个奇点出发，到另一个奇点终止。

如果没有奇点，则一定存在一条欧拉回路。

对于有向图 G，当且仅当 G 是连通的，且每个点的入度等于出度，则存在欧拉回路。

如果有两个顶点的入度与出度不相等，且一个顶点的入度比出度小 1，另一个顶点的入度比出度大 1，此时，存在一条欧拉道路，以前一个顶点为起点，以后一个顶点为终点。

代码

round_trip.cpp

```
// POJ 1041 John's trip, http://poj.org/problem?id=1041
#include <iostream>
#include <cstring>
#include <algorithm>
#include <stack>

using namespace std;

const int MAX_NV = 45;
const int MAX_NE = 1996;

/**
 * @struct 图，邻接矩阵的变种.
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // G[点][边] = 点，这样是为了能方便让边 lexicographically 输出
    int matrix[MAX_NV][MAX_NE];
};

graph_t G;

bool visited[MAX_NE];  // 边是否已访问
int degree[MAX_NV];    // 点的度
```

```
stack<int> s;  // 栈，用于输出

void stack_print(stack<int> &s) {
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;
}

void euler(int u) {
    for (int e = 1; e <= G.ne; e++) {
        if (!visited[e] && G.matrix[u][e]) { //若相邻边未访问过
            visited[e] = true;
            euler(G.matrix[u][e]);
            s.push(e);
        }
    }
}

int main() {
    int x, y, z, start;
    while ((cin >> x >> y) && x && y) {
        memset(visited, false, sizeof(visited));
        memset(degree, 0, sizeof(degree));
        memset(&G, 0, sizeof(G));

        start = x < y ? x : y;
        cin >> z;
        G.ne = max(G.ne, z);
        G.nv = max(G.nv, max(x, y));
        G.matrix[x][z] = y;
        G.matrix[y][z] = x;
        ++degree[x];
        ++degree[y];

        while ((cin >> x >> y) && x && y) {
            cin >> z;
            G.ne = max(G.ne, z);
            G.nv = max(G.nv, max(x, y));
            G.matrix[x][z] = y;
            G.matrix[y][z] = x;
            ++degree[x];
            ++degree[y];
        }

        /* 欧拉回路形成的条件之一，判断结点的度是否为偶数 */
        bool flag = true;
        for (int i = 1; i <= G.nv; i++) {
            if (degree[i] & 1) {
                flag = false;
                break;
            }
        }

        if (!flag) {
            cout << "Round trip does not exist." << endl;
        } else {
            euler(start);
            stack_print(s);
        }
    }
    return 0;
```

```
}
```

round_trip.cpp

相关的题目

与本题相同的题目：

- POJ 1041 John's trip, http://poj.org/problem?id=1041

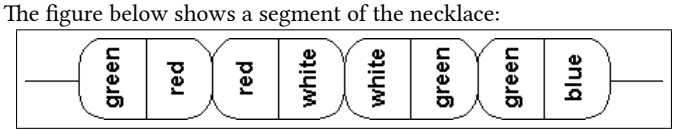
与本题相似的题目：

- 《算法竞赛入门经典》[1] 第 111 页 6.4.4 节
- UVa 10054 The Necklace, http://t.cn/zRwqcRp
- UVa 10129 Play on Words, http://t.cn/zTInBDX

15.1.3 The Necklace

描述

My little sister had a beautiful necklace made of colorful beads. Two successive beads in the necklace shared a common color at their meeting point. The figure below shows a segment of the necklace:



But, alas! One day, the necklace was torn and the beads were all scattered over the floor. My sister did her best to recollect all the beads from the floor, but she is not sure whether she was able to collect all of them. Now, she has come to me for help. She wants to know whether it is possible to make a necklace using all the beads she has in the same way her original necklace was made and if so in which order the bids must be put.

Please help me write a program to solve the problem.

Input

The input contains T test cases. The first line of the input contains the integer T.

The first line of each test case contains an integer $N(5 \leq N \leq 1000)$ giving the number of beads my sister was able to collect. Each of the next N lines contains two integers describing the colors of a bead. Colors are represented by integers ranging from 1 to 50.

Output

For each test case in the input first output the test case number as shown in the sample output. Then if you apprehend that some beads may be lost just print the sentence "some beads may be lost" on a line by itself. Otherwise, print N lines with a single bead description on each line. Each bead description consists of two integers giving the colors of its two ends. For $1 \leq i \leq N_1$, the second integer on line i must be the same as the first integer on line i + 1. Additionally, the second integer on line N must be equal to the first integer on line 1. Since there are many solutions, any one of them is acceptable.

Print a blank line between two successive test cases.

Sample Input

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

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

```
2 1
2 2
3 4
3 1
2 4
```

Sample Output

```
Case #1
some beads may be lost

Case #2
2 1
1 3
3 4
4 2
2 2
```

分析

欧拉回路。

注意顶点可以有自环。

代码

eulerian_circuit.c

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

#define MAXN 51  // 顶点最大个数

int G[MAXN][MAXN];
int visited_vertices[MAXN];
int visited_edges[MAXN][MAXN];
int count[MAXN]; // 顶点的度

void dfs(const int u) {
    int v;
    visited_vertices[u] = 1;
    for(v = 0;  v < MAXN; v++) if(G[u][v] && !visited_vertices[v]) {
        dfs(v);
    }
}

/*
 * @brief 欧拉回路，允许自环和重复边
 * @param[in] u 起点
 * @return 无
 */
void euler(const int u){
    int v;
    for(v = 0; v < MAXN; ++v) if(G[u][v]){
        --G[u][v]; --G[v][u]; // 这个技巧，即有 visited 的功能，又允许重复边
        euler(v);
        // 逆向打印，或者存到栈里再打印
        printf("%d %d\n", u, v);
    }
}

int main() {
    int T, N, a, b;
    int i;
    int cases=1;
```

```
    scanf("%d",&T);
    while(T--) {
        int flag = 1; // 结点的度是否为偶数
        int flag2 = 1; // 图是否是连通的

        memset(G, 0, sizeof(G));
        memset(count, 0, sizeof(count));

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

        printf("Case #%d\n", cases++);

        // 欧拉回路形成的条件之一，判断结点的度是否为偶数
        for(i=0; i<MAXN; ++i) {
            if(count[i] & 1){
                flag = 0;
                break;
            }
        }
        // 检查图是否连通
        if(flag) {
            memset(visited_vertices, 0, sizeof(visited_vertices));
            memset(visited_edges, 0, sizeof(visited_edges));

            for(i=0; i< MAXN; ++i)
                if(count[i]) {
                    dfs(i);
                    break;
                }
            for(i=0; i< MAXN; ++i){
                if(count[i] && !visited_vertices[i]) {
                    flag2 = 0;
                    break;
                }
            }
        }
        if (flag && flag2) {
            for(i = 0; i < MAXN; ++i) if(count[i]){
                euler(i);
                break;
            }
        } else {
            printf("some beads may be lost\n");
        }

        if(T > 0) printf("\n");
    }
    return 0;
}
```

eulerian_circuit.c

相关的题目

与本题相同的题目：

- UVa 10054 The Necklace, http://t.cn/zRwqcRp

与本题相似的题目：

- 《算法竞赛入门经典》[1] 第 111 页 6.4.4 节
- POJ 1041 John's trip, http://poj.org/problem?id=1041
- UVa 10129 Play on Words, http://t.cn/zTlnBDX

15.2 图的广搜

15.3 最小生成树

"最小"指的是边的权值之和最小。

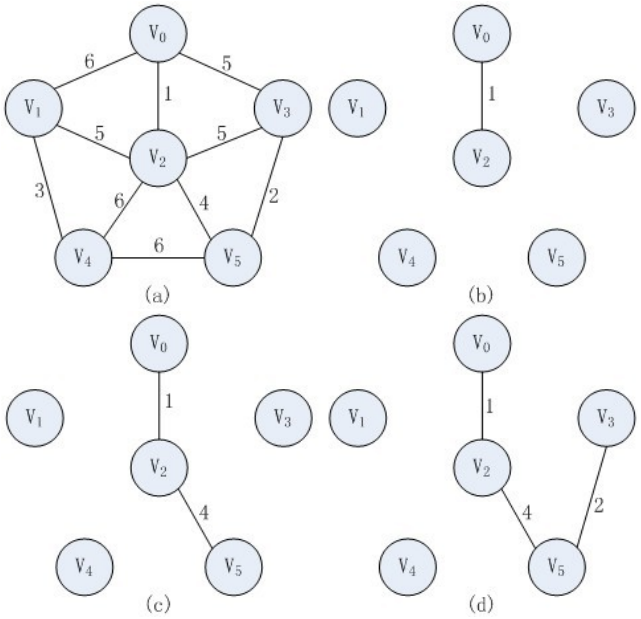
构造最小生成树 (Minimum Spanning Tree, MST) 有多种算法。其中多数算法利用了最小生成树的一个性质（简称为 MST 性质）：假设 $N=(V,E)$ 是一个连通网，U 是顶点集 V 的一个非空子集。若 (u,v) 是一条具有最小权值的边，其中 $u\in U, v\in V-U$，则必存在一颗包含边 (u,v) 的最小生成树。

Prim 算法和 Kruskal 算法是两个利用 MST 性质构造最小生成树的算法。它们都属于贪心法。

15.3.1 Prim 算法

假设 $N=(V,E)$ 是一个连通网，TE 是 N 上最小生成树中边的集合。算法从 $U=u_0(u_0\in V), TE=\{\}$ 开始，重复执行下述操作：在所有 $u\in U, v\in V-U$ 的边 $(u,v)\in E$ 中找一条代价最小的边 (u_0,v_0) 并入集合 TE，同时 v_0 并入 U，直至 $U=V$ 为止。此时 TE 中必有 $n-1$ 条边，则 $T=(V,TE)$ 为 N 的最小生成树。为实现这个算法需附设一个数组 `closedge`，以记录从 U 到 $V-U$ 具有最小代价的边。对每个顶点 $v_i\in V-U$，在辅助数组中存在一个相应分量 `closedge[i-1]`，它包括两个域，其中 `lowcost` 存储该边上的权。显然，$closedge[i].lowcost=\min\{cost(u,v_i), u\in U\}$。`adjvex` 域存储该边依附的在 U 中的顶点。

图 15-1所示为按 Prim 算法构造网的一棵最小生成树的过程，在构造过程中辅助数组中各分量值的变化如表15-1所示。



[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

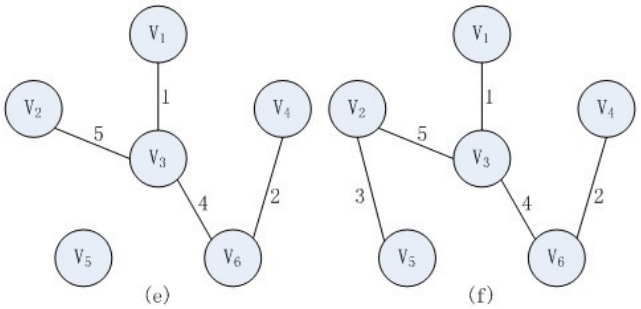


图 15-1 Prim 算法构造最小生成树的过程

表 15-1 构造最小生成树过程中辅助数组的变化

i / closedge	1	2	3	4	5	U	U-V	k
adjvex	v_0	v_0	v_0			v_0	$\{v_1, v_2, v_3, v_4, v_5\}$	2
lowcost	6	1	5					
adjvex	v_2		v_1	v_2	v_2	$\{v_0, v_2\}$	$\{v_1, v_3, v_4, v_5\}$	5
lowcost	5	0	5	6	4			
adjvex	v_2		v_6	v_2		$\{v_0, v_2, v_5\}$	$\{v_1, v_3, v_4\}$	3
lowcost	5	0	2	6	0			
adjvex	v_2			v_2		$\{v_0, v_2, v_5, v_3\}$	$\{v_1, v_4\}$	1
lowcost	5	0	0	6	0			
adjvex				v_1		$\{v_0, v_2, v_5, v_3, v_1\}$	$\{v_4\}$	4
lowcost	0	0	0	3	0			
adjvex						$\{v_0, v_2, v_5, v_3, v_1, v_4\}$	$\{\}$	
lowcost	0	0	0	0	0			

代码

am_graph_prim1.cpp

```
#include <iostream>
#include <climits>  /* for INT_MAX */

using namespace std;

/** 顶点数最大值. */
const int MAX_NV = 100;

/** 边的权值类型, 可以为 int, float, double. */
typedef int graph_weight_t;
const graph_weight_t GRAPH_INF = INT_MAX;

/**
 * @struct 图, 用邻接矩阵 (Adjacency Matrix).
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接矩阵, 存放边的信息, 如权重等
    graph_weight_t matrix[MAX_NV][MAX_NV];
};

graph_t g;
```

```
struct closedge_t {
    int adjvex; /* 弧头，属于 U */
    /* 边 adjvex-> 本下标 的权值，-GRAPH_INF 表示已经加入 U */
    graph_weight_t lowcost;
};

/*
 * @brief 在 V-E 集合中寻找最小的边
 * @param[in] closedge MST 中的边，起点为 adjvex，终点为本下标
 * @param[in] n closedge 数组的长度
 * @return 找到了则返回弧尾的下标，V-U 为空集则返回-1，表示终止
 */
static int min_element(const closedge_t closedge[], int n) {
    int min_value = GRAPH_INF;
    int min_pos = -1;
    for (int i = 0; i < n; i++)
        if (closedge[i].lowcost > -GRAPH_INF) {
            if (min_value > closedge[i].lowcost) {
                min_value = closedge[i].lowcost;
                min_pos = i;
            }
        }
    return min_pos;
}

/**
 * @brief Prim 算法，求图的最小生成树.
 * @param[in] g 图对象的指针
 * @return MST 的边的权值之和
 */
graph_weight_t prim(const graph_t &g) {
    graph_weight_t sum = 0; /* 权值之和 */
    int u = 0; /* 从 0 号顶点出发 */
    const int n = g.nv;
    /* closedge[n]，记录从顶点集 U 到 V-U 的边 */
    closedge_t* const closedge = new closedge_t[n];

    /* 辅助数组初始化 */
    for (int i = 0; i < n; i++) if (i != u) {
        closedge[i].adjvex = u;
        closedge[i].lowcost = g.matrix[u][i];
    }
    closedge[u].lowcost = -GRAPH_INF; /* 初始，U={u} */

    for (int i = 0; i < n; i++) if (i != u) { /* 其余的 n-1 个顶点 */
        /* 求出 TE 的下一个顶点 k */
        const int k = min_element(closedge, n);
        /* 输出此边 closedge[k].adjvex --> k */
        cout << (char)('A' + closedge[k].adjvex) << " - " << (char)('A' + k)
                << " : "<< g.matrix[closedge[k].adjvex][k] << endl;
        sum += g.matrix[closedge[k].adjvex][k];
        // sum += closedge[k].lowcost;  // 等价
        closedge[k].lowcost = -GRAPH_INF;  /* 顶点 k 并入 U，表示此边加入 TE */
        /* 更新 k 的邻接点的值，不相邻为无穷大 */
        for (int j = 0; j < n; j++) {
            const graph_weight_t w = g.matrix[k][j];
            if (w < closedge[j].lowcost) {
                closedge[j].adjvex = k;
                closedge[j].lowcost = w;
            }
        }
    }
    delete[] closedge;
    return sum;
```

```
}

/** 读取输入，构建图. */
void read_graph() {
    int m, n;

    /* 读取节点和边的数目 */
    cin >> m >> n;
    g.nv = m;
    g.ne = n;

    /* 初始化图，所有节点间距离为无穷大 */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            g.matrix[i][j] = GRAPH_INF;
        }
    }

    /* 读取边信息 */
    for (int k = 0; k < n; k++) {
        char chx, chy;
        int w;
        cin >> chx >> chy >> w;
        const int i = chx - 'A';
        const int j = chy - 'A';
        g.matrix[i][j] = w;
        g.matrix[j][i] = w;
    }
}

/* test
输入数据:
7 11
A B 7
A D 5
B C 8
B D 9
B E 7
C E 5
D E 15
D F 6
E F 8
E G 9
F G 11

输出:
A - D : 5
D - F : 6
A - B : 7
B - E : 7
E - C : 5
E - G : 9
Total:39
*/
int main() {
    read_graph();
    cout << "Total : " << prim(g) << endl;
    return 0;
}
```

am_graph_prim1.cpp

算法分析

假设网中有 n 个顶点，则第一个进行初始化的循环语句的频度为 n，第二个循环语句的频度为 $n-1$。其中有两个内循环：其一是在 `closedge[v].lowcost` 中求最小值，其频度为 $n-1$；其二是重新选择具有最小代价的边，其频度为 n。因此 Prim 算法的时间复杂度为 $O(n^2)$，与网中边数无关，因此适用于求边稠密的图的最小生成树。

Prim 算法的另一种实现是使用小根堆，其流程是：小根堆中存储一个端点在生成树中，另一个端点不在生成树的边，每次从小根堆的堆顶可选出权值最小的边 (u,v)，将其从堆中推出，加入生成树中。然后将新出现的所有一个端点在生成树中，一个端点不在生成树的边都插入小根堆中。下一轮迭代中，下一条满足要求的边又上升到堆顶。如此重复 $n-1$ 次，最后建立起该图的最小生成树。该算法的 C 代码实现如下。

代码

am_graph_prim2.cpp

```
#include <iostream>
#include <climits>  /* for INT_MAX */
#include <queue>
#include <algorithm>

using namespace std;

/** 顶点数最大值. */
const int MAX_NV = 100;

/** 边的权值类型，可以为 int, float, double. */
typedef int graph_weight_t;
const graph_weight_t GRAPH_INF = INT_MAX;

/**
 * @struct 图，用邻接矩阵 (Adjacency Matrix).
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接矩阵，存放边的信息，如权重等
    graph_weight_t matrix[MAX_NV][MAX_NV];
};

graph_t g;

/**
 * @struct 边
 */
struct edge_t{
    int u;  // from
    int v;  // to
    graph_weight_t w;  // 权值

    bool operator>(const edge_t &other) const {
        return w > other.w;
    }
};

/**
  * @brief Prim 算法，求图的最小生成树.
  * @param[in] g 图对象的指针
  * @return MST 的边的权值之和
  */
int prim(const graph_t &g){
    graph_weight_t sum = 0; // 权值之和
    priority_queue<edge_t, vector<edge_t>, greater<edge_t> > q;
    const int n = g.nv;
    bool* used = new bool[n];  // 判断顶点是否已经加入最小生成树
```

```
    std::fill(used, used + n, false);

    int count = 1;  // MLE 当前的边数
    int u = 0;      // 从 0 号顶点出发
    used[u] = true;    // 开始顶点加入 U(所以 count 初始为 1)
    while (count < n) {
        for(int v = 0; v < n; v++) if(!used[v]) { // 若 v 不在生成树，(u,v) 加入堆
            edge_t e = {u, v, g.matrix[u][v]};
            q.push(e);
        }
        while(!q.empty() && count < n) {
            const edge_t e = q.top(); q.pop();  // 从堆中退出最小权值边，存入 e
            if(!used[e.v]) {
                // 输出生成树 TE 的边，即此边加入 TE
                cout << (char)('A' + e.u) << " - " << (char)('A' + e.v) <<
                        " : " << g.matrix[e.u][e.v] << endl;
                sum += g.matrix[e.u][e.v];
                u = e.v;
                used[u] = true; // u 并入到生成树的顶点集合 U
                count++;
                break;
            }
        }
    }

    delete[] used;
    return sum;
}

// ...
```

am_graph_prim2.cpp

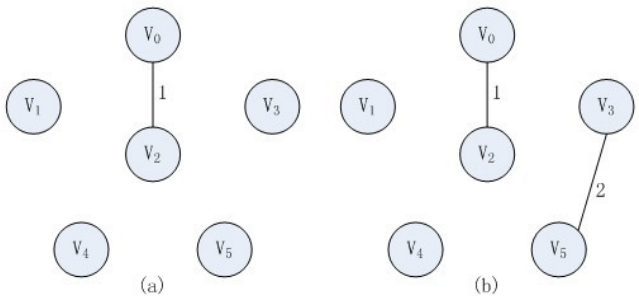
算法分析

该算法迭代次数为 $O(n)$，每次迭代将平均 e/n 条边插入最小堆中，e 条边从堆中删除，堆的插入和删除操作时间复杂度均为 $O(\log_2 e)$，则总的时间复杂度为 $O(e \log_2 e)$。

15.3.2 Kruskal 算法

假设连通网 $N = V, E$，则令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T = (V,)$，图中每个顶点自成一个连通分量。在 E 中选择代价最小的边，若该边依附的顶点落在 T 中不同的连通分量上，则将此边加入到 T 中，否则舍去此边而选择下一条代价最小的边。依次类推，直至 T 中所有顶点都在同一连通分量上为止。

图15-2所示为 Kruskal 算法构造一棵最小生成树的过程。



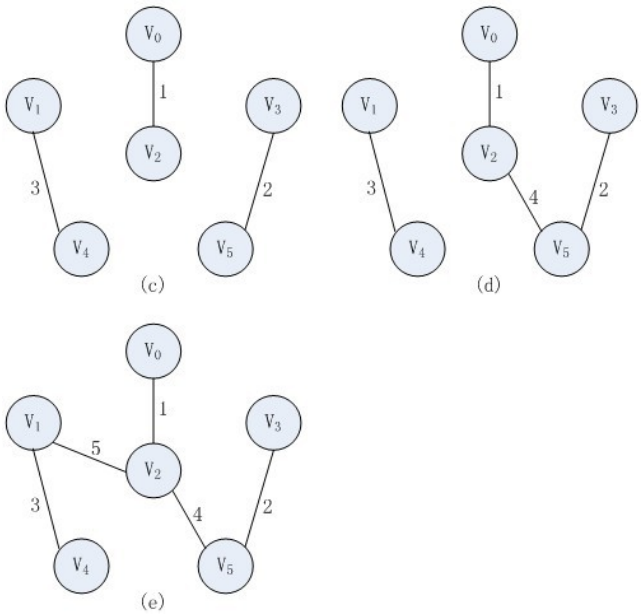


图 15-2 Kruskal 算法构造最小生成树的过程

下面是 Kruskal 算法的 C 语言实现。

代码

kruskal.cpp

```
#include <iostream>
#include <climits>  /* for INT_MAX */
#include <queue>
#include <algorithm>

using namespace std;

/* 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 */

const int MAX_NV = 11; /* 顶点数最大值 */
const int MAX_NE = 100;  /* 最大边数 */
/** 边的权值类型. */
typedef int graph_weight_t;

/** 图的边. */
struct edge_t{
    int u;  /** 顶点编号 */
    int v;  /** 顶点编号 */
    graph_weight_t w;  /** 权值 */

    bool operator>(const edge_t &other) const {
        return w > other.w;
    }
};

edge_t edges[MAX_NE];


/*
  * @brief Kruskal 算法，堆 + 并查集.
  * @param[in] edges 边的数组
  * @param[in] n 边数，一定要大于或等于 (顶点数-1)
  * @param[in] m 顶点数
  * @return MST 的边的权值之和
```

```
  */
graph_weight_t kruskal(const edge_t edges[], int n, int m) {
    graph_weight_t sum = 0;
    priority_queue<edge_t, vector<edge_t>,
                                greater<edge_t> > q;
    ufs_t *s = ufs_create(MAX_NV);
    if (n < m - 1) return -1;

    /* 把所有边插入堆中 */
    for (int i = 0; i < n; i++) {
        q.push(edges[i]);
    }

    for (int i = 0; i < n; i++) {
        /* 从堆中退出最小权值边 */
        const edge_t e = q.top(); q.pop();
        /* 取两顶点所在集合的根 */
        const int u = ufs_find(s, e.u);
        const int v = ufs_find(s, e.v);
        if (u != v) { /* 不是同一集合，说明不连通 */
            ufs_union(s, u, v); /* 合并，连通成一个分量 */
            /* 输出生成树 TE 的边，即此边加入 TE */
            cout << (char)('A' + e.u) << " - " << (char)('A' + e.v) << endl;
            sum += e.w;
        }
    }

    ufs_destroy(s);
    return sum;
}

bool operator<(const edge_t &e1, const edge_t &e2) {
    return e1.w < e2.w;
}

/** Kruskal 算法，快排 + 并查集. */
graph_weight_t kruskal1(edge_t edges[], int n, int m) {
    graph_weight_t sum = 0;
    ufs_t *s = ufs_create(MAX_NV);  /* 并查集，0 位置未用  */
    if (n < m - 1) return -1;

    std::sort(edges, edges + n);

    for (int i = 0; i < n; i++) {
        /* 从堆中退出最小权值边，存入 ed */
        const edge_t e = edges[i];
        /* 取两顶点所在集合的根 */
        const int u = ufs_find(s, e.u);
        const int v = ufs_find(s, e.v);
        if (u != v) { /* 不是同一集合，说明不连通 */
            ufs_union(s, u, v); /* 合并，连通成一个分量 */
            /* 输出生成树 TE 的边，即此边加入 TE */
            cout << (char)('A' + e.u) << " - " << (char)('A' + e.v) << endl;
            sum += e.w;
        }
    }
    ufs_destroy(s);
    return sum;
}

/* test
输入数据:
7 11
A B 7
```

```
A D 5
B C 8
B D 9
B E 7
C E 5
D E 15
D F 6
E F 8
E G 9
F G 11

输出:
A - D
C - E
D - F
B - E
A - B
E - G
Total : 39
*/
int main() {
    int m, n;
    /* 读取顶点数，边数目 */
    cin >> m >> n;

    /* 读取边信息 */
    for (int i = 0; i < n; i++) {
        char chx, chy;
        int w;
        cin >> chx >> chy >> w;

        edges[i].u = chx - 'A';
        edges[i].v = chy - 'A';
        edges[i].w = w;
    }

    /* 求解最小生成树 */
    cout << "Total : " << kruskal(edges, n, m) << endl;
    return 0;
}
```

kruskal.cpp

算法分析

如果采用邻接矩阵作为图的存储结构，则在建立小根堆时需要检测图的邻接矩阵，这需要 $O(n^2)$ 的时间。此外，需要将 e 条边组成初始的小根堆。如果直接从空堆开始，依次插入各边，需要 $O(e\log_2 e)$ 的时间。在构造最小生成树的过程中，需要进行 $O(e)$ 次出堆操作 `heap_remove()`、$2e$ 次并查集的 `ufs_find()` 操作以及 $n-1$ 次 `ufs_union()` 操作，计算时间分别为 $O(e\log_2 e)$、$O(\log_2 n)$ 和 $O(n)$，所以总时间为 $O(n^2+e\log_2 e)$。

如果采用邻接表作为图的存储结构，则在建立小根堆时需要检测图的邻接表，这需要 $O(n+e)$ 的时间。为建成初始的小根堆，需要 $O(e\log_2 e)$ 的时间。在构造最小生成树的过程中，需要进行 $O(e)$ 次出堆操作 `heap_remove()`、$2e$ 次并查集的 `ufs_find()` 操作以及 $n-1$ 次 `ufs_union()` 操作，计算时间分别为 $O(e\log_2 e)$、$O(e\log_2 n)$ 和 $O(n)$，所以总时间为 $O(n+e\log_2 e)$。

15.3.3 Highways

描述

一个名叫 Flatopia 的岛国地势非常平坦。不幸的是 Flatopia 的公共高速公路系统很差劲。Flatopia 的政府也意识到了这个问题，已经建造了许多高速公路用来连接比较重要的城镇。不过，仍然有一些城镇没有接入高速公路。因此，很有必要建造更多的高速公路，让任意两个城镇之间可以通过高速公路连接。

Flatopia 的城镇从 1 到 N 编号，城镇 i 的位置由笛卡尔坐标 (x_i, y_i) 表示。每条高速公路仅连接两个城镇。所有的高速公路都是直线，因此它们的长度就等于两个城镇之间的欧氏距离。所有的高速公路是双向的，高速公路之间可以相交，但是司机只能在公路的端点（也即城镇）换道。

Flatopia 政府希望能最小化建造高速公路的代价。由于 Flatopia 地势平坦，一条高速公路的代价正比于它的长度。因此，应该让高速公路的总长度最小。

输入

输入由两部分组成。第一部分描述所有的城镇，第二部分描述所有已经建造好的高速公路。

第一行包含一个整数 $N(1 \leq N \leq 750)$，表示城镇的数目。接下来的 N 行每行包含一对整数，x_i 和 y_i，由空格隔开，表示第 i 个城镇的坐标。坐标的绝对值不会超过 10000。每个城镇的坐标都不重叠。

接下来一行包含一个整数 $M(0 \leq M \leq 1000)$，表示已经存在的高速公路的数目。接下来的 M 行每行包含一对整数，给出了一对城镇编号，表示这两个城镇被一条高速公路连接起来。每两个城镇之间最多被一条高速公路连接。

输出

输出所有需要新建的高速公路。每行一个高速公路，用一对城镇编号表示。

如果不需要新建高速公路，输出为空。

样例输入

```
9
1 5
0 0
3 2
4 5
5 1
0 4
5 2
1 2
5 3
3
1 3
9 7
1 2
```

样例输出

```
1 6
3 7
4 9
5 7
8 3
```

分析

很明显，最小生成树。

题中的网络是一个完全图，任意两个城镇之间都有边，权值是两点间的距离。因此 Prim 算法比 Kruskal 算法效率更高。

对于已经存在的高速公路，令它们权值为 0，可以保证它们一定会被选中。

因为题目只需要输出新建的高速公路的两个端点，不需要输出最小生成树的长度，所以计算距离的时候不用 sqrt，也就不用 double 了。

代码

highways.cpp

```
// POJ 1751 Highways, http://poj.org/problem?id=1751
```

```
// 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 "am_graph_prim1.cpp"  // 见“图-> 最小生成树->Prim 算法”这节

// 1. 修改范围
const int MAX_NV = 750;
// 2. 重写 read_graph()
// 3. 重写 main()
// 4. 修改 mgraph_prim() 里的 printf，权值大于 0 才打印出来
if (g.matrix[closedge[k].adjvex][k] > 0)
    cout << closedge[k].adjvex+1 << " " << k+1 << endl;

// 输入数据
int n, m, x[MAX_NV], y[MAX_NV];

/*
 * @brief 两点之间的距离.
 *
 * 因为题目只需要输出新建的高速公路的两个端点，不需要输出最小生成
 * 树的长度，能比较大小即可，所以用距离的平方，简化计算。
 *
 * @param[in] i 编号为 i+1 的城镇
 * @param[in] j 编号为 j+1 的城镇
 *
 * @return 欧氏距离的平方
 */
static int distance(int i,int j) {
    return (x[i]-x[j]) * (x[i]-x[j]) + (y[i]-y[j]) * (y[i]-y[j]);
}

/** 读取输入，构建图. */
void read_graph() {
    int i, j;
    cin >> n;
    g.nv = n;
    g.ne = n * (n - 1) / 2;

    for (i = 0; i < n; i++)
        cin >> x[i] >> y[i];
    for (i = 0; i < n; i++)
        for (j = i; j < n; j++)
            g.matrix[i][j] = g.matrix[j][i] = distance(i, j);

    cin >> m;
    for (i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g.matrix[a - 1][b - 1] = g.matrix[b - 1][a - 1] = 0;
    }
}

int main() {
    read_graph();
    prim(g);
    return 0;
}
```

highways.cpp

相关的题目

与本题相同的题目：

- POJ 1751 Highways, http://poj.org/problem?id=1751

与本题相似的题目：

- POJ 2485 Highways, http://poj.org/problem?id=2485
- POJ 1861 Network, http://poj.org/problem?id=1861
- POJ 2395 Out of Hay, http://poj.org/problem?id=2395
- POJ 2377 Bad Cowtractors, http://poj.org/problem?id=2377
- POJ 2421 Constructing Roads, http://poj.org/problem?id=2421
- POJ 1679 The Unique MST, http://poj.org/problem?id=1679
- POJ 1258 Agri-Net, http://poj.org/problem?id=1258
- POJ 1251 Jungle Roads, http://poj.org/problem?id=1251
- POJ 3625 Building Roads, http://poj.org/problem?id=3625
- POJ 1789 Truck History, http://poj.org/problem?id=1789

15.3.4 最优布线问题

描述

学校需要将 n 台计算机连接起来，不同的 2 台计算机之间的连接费用可能是不同的。为了节省费用，我们考虑采用间接数据传输结束，就是一台计算机可以间接地通过其他计算机实现和另外一台计算机连接。

为了使得任意两台计算机之间都是连通的（不管是直接还是间接的），需要在若干台计算机之间用网线直接连接，现在想使得总的连接费用最省，让你编程计算这个最小的费用。

输入

输入第一行为两个整数 $n, m(2 \leq n \leq 100000, 2 \leq m \leq 100000)$，表示计算机总数，和可以互相建立连接的连接个数。接下来 m 行，每行三个整数 a, b, c 表示在机器 a 和机器 b 之间建立连接的话费是 c。(题目保证一定存在可行的连通方案, 数据中可能存在权值不一样的重边，但是保证没有自环)

输出

输出只有一行一个整数，表示最省的总连接费用。

样例输入

```
3 3
1 2 1
1 3 2
2 3 1
```

样例输出

```
2
```

分析

本题是非常直白的 kruskal 算法，可以直接使用第 §15.3.2节的样例代码。

代码

wiring.c

```
// wikioi 1231 最优布线问题, http://www.wikioi.com/problem/1231/
// 1. 修改范围
const int MAX_NV = 100001; // 顶点数最大值
const int MAX_NE = 100000;  // 最大边数
// 2. 注释掉 kruskal() 里的 printf()
// 3. sum 类型改为 long long, kruskal() 返回值改为 long long
// 4. main() 里的 printf 改为 %lld
// 5. main() 里输入边时, 顶点由字母改为整数
```

```
// 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 "kruskal.cpp"  // 见“图-> 最小生成树->Kruskal 算法”这节
```
wiring.c

相关的题目

与本题相同的题目：

- wikioi 1231 最优布线问题, http://www.wikioi.com/problem/1231/

与本题相似的题目：

- None

15.4 最短路径

15.4.1 单源最短路径——Dijkstra 算法

假设 S 为已求得最短路径的点的集合，则可证明：下一条最短路径（设其终点为 x）或者是弧 (v,x)，或者是中间只经过 S 中的顶点而最后到达顶点 x 的路径。

Dijkstra 算法流程如下：

1. S 为已找到从 v 出发的最短路径的终点的集合，它的初始状态为空集。`dist[i]` 存放的是 v 到 v_i 的最短路径长度，根据前面所述性质，`dist[i]=min{dist[i],weight(v,`v_i`)}`。`path[i]` 存放的是最短路径上指向 v_i 的弧尾顶点。那么从 v 出发到图上其余 v_i 的最短路径长度的初值为：

$$\text{dist}[i] = \text{weight}(v, v_i), v_i \in V$$

2. 选择 v_j，使得

$$\text{dist}[j] = \min\{\text{dist}[j], \text{weight}(v, v_j) | v_j \in V - S\}$$

将 v_j 加入到 S，

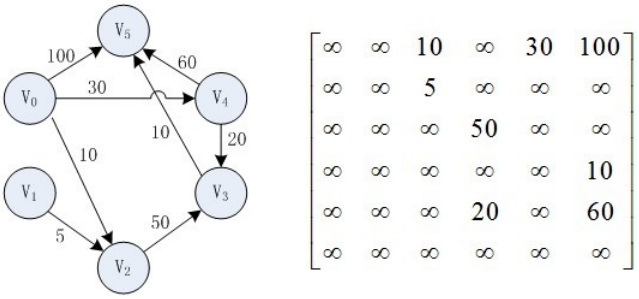
$$S = S \cup v_j$$

3. 修改从 v 出发到集合 $V-S$ 上任一顶点 v_k 可达的最短路径长度，并记录下这条边。

```
if(dist[j] + weight(j, k) < dist[k]) {
    dist[k] = dist[j] + weight(j, k);
    path[k] = j; /* 修改到 k 的最短路径 */
}
```

4. 重复 2，3 共 $n-1$ 次。

例如，对图15-3所示的有向图及其邻接矩阵运行运行 Dijkstra 算法，



$$\begin{bmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

图 15-3 有向图及其邻接矩阵

运算过程中 v_0 到其余个顶点的最短路近，`dist[]` 向量的变化情况如表15-2所示（从一列到下一列只需要更新新加入点的邻接点）。

表 15-2 Dijkstra 算法过程中 dist[] 向量的变化情况

终点	i=1	i=2	i=3	i=4	i=5
v_1	∞	∞	∞	∞	∞
v_2	**10** $(\mathbf{v_0}, \mathbf{v_2})$				
v_3	∞	60 (v_0, v_2, v_3)	**50** $(\mathbf{v_0}, \mathbf{v_4}, \mathbf{v_5})$		
v_4	30 (v_0, v_4)	**30** $(\mathbf{v_0}, \mathbf{v_4})$			
v_5	100 (v_0, v_5)	100 (v_0, v_5)	90 (v_0, v_4, v_5)	**60** $(\mathbf{v_0}, \mathbf{v_4}, \mathbf{v_3}, \mathbf{v_5})$	
v_j	v_2	v_4	v_3	v_5	
S	(v_0, v_2)	(v_0, v_2, v_4)	(v_0, v_2, v_3, v_4)	$(v_0, v_2, v_3, v_4, v_5)$	

Dijkstra 算法的 C++ 语言实现如下。

代码

al_graph_dijkstra.cpp

```
#include <iostream>
#include <queue>
#include <map>
#include <utility>
#include <climits>

using namespace std;

/** 边的权值类型，可以为 int, float, double. */
typedef int graph_weight_t;

/** 顶点的编号，可以为 char, int, string 等. */
typedef char graph_vertex_id_t;

/**
 * @struct 图，用邻接表 (Adjacency List).
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接表，存放边的信息，如权重等
    map<graph_vertex_id_t, map<graph_vertex_id_t, graph_weight_t> > matrix;
};


/**
 * @brief Dijkstra 算法求单源最短路径.
 * @param[in] g 图
 * @param[in] start 起点
 * @param[out] dist dist[v] 存放的是起点到 v 的最短路径长度
 * @param[out] father father[v] 存放的是最短路径上指向 v 的上一个顶点
 * @return 无
 */
void dijkstra(const graph_t &g, graph_vertex_id_t start,
        map<graph_vertex_id_t, graph_weight_t> &distance,
        map<graph_vertex_id_t, graph_vertex_id_t> &father) {
    /* 起点到顶点的最短距离. */
    typedef pair<graph_weight_t, graph_vertex_id_t> to_dist_t;
```

```
    // 小根堆
    priority_queue<to_dist_t, vector<to_dist_t>, greater<to_dist_t> > q;
    distance[start] = 0;  // 自己到自己，距离为 0
    q.push(to_dist_t(0, start));
    while (!q.empty()) {
        const to_dist_t u = q.top(); q.pop();
        if (g.matrix.find(u.second) == g.matrix.end()) continue;

        for (map<graph_vertex_id_t, graph_weight_t>::const_iterator iter =
                g.matrix.at(u.second).begin();
                iter != g.matrix.at(u.second).end(); ++iter) {
            const graph_vertex_id_t v = iter->first;
            const graph_weight_t w = iter->second;

            if (!distance.count(v) || distance[u.second] + w < distance[v]) {
                distance[v] = distance[u.second] + w;
                father[v] = u.second;
                q.push(to_dist_t(distance[v], v));
            }
        }
    }
    return;
}

/**
 * @brief 打印从起点到终点的最短路径
 * @param[in] father Dijkstra 计算好的 father 数组
 * @param[in] end 终点
 * @return 无
 */
void print_path(const map<graph_vertex_id_t, graph_vertex_id_t> &father,
        graph_vertex_id_t end) {
    if (!father.count(end)) {
        cout << end;
    } else {
        print_path(father, father.at(end));
        cout << "->" << end;
    }
}

/** 读取输入，构建图. */
void read_graph(graph_t &g) {
    cin >> g.ne;
    for (int i = 0; i < g.ne; ++i) {
        graph_vertex_id_t from, to;
        graph_weight_t weight;
        cin >> from >> to >> weight;
        g.matrix[from][to] = weight;
    }
}

/* test

输入数据:
8
A C 10
A E 30
A F 100
B C 5
C D 50
D F 10
E D 20
E F 60
```

```
输出:
A->C
A->E->D
A->E
A->E->D->F
*/
int main() {
    graph_t g;
    map<graph_vertex_id_t, graph_weight_t> distance;
    map<graph_vertex_id_t, graph_vertex_id_t> father;

    read_graph(g);

    dijkstra(g, 'A', distance, father);

    // 输出所有路径
    for(map<graph_vertex_id_t, graph_vertex_id_t>::const_iterator iter =
            father.begin(); iter != father.end();  ++iter) {
        if (iter->first != 'A') {
            print_path(father, iter->first);
            cout << endl;
        }
    }
    return 0;
}
```

al_graph_dijkstra.cpp

算法分析

该算法包含了两个并列的 for 循环，第一个 for 循环做辅助数组的初始化工作，计算时间为 $O(n)$，第二个 for 循环是二重嵌套循环，进行最短路径的求解工作，由于对图中几乎每个顶点都要做计算，每个顶点的又要对集合 S 内的顶点进行检测，对集合 $V-S$ 内中的顶点进行修改，所以运算时间复杂度为 $O(n^2)$。算法总的时间复杂度为 $O(n^2)$。

15.4.2 第 k 短路/长路

求给定无向图中从 s 到 t 得第 k 短路/长路。

用每个点到 t 得最短路径长度，用启发搜索进行判断剪枝，当第 k 次到达 t 的节点是就求出了第 k 短路。具体实现用堆可足够优化。

15.4.3 每点最短路径——Floyd 算法

Floyd 算法的基本思想是：假设求从定点 v_i 到 v_j 的最短路径。初始时，若 v_i 与 v_j 之间存在边，则最短路径长度为此边的权值；若不存在边，则最短路径长度为无穷大。以后逐步在路径中加入顶点 $k(k=0,1,...,n-1)$ 作为中间顶点，如果加入中间顶点后，得到的路径比原来的路径长度减少了，则以新路径代替原路径。

首先比较 (v_i,v_j) 和 (v_i,v_0,v_j) 的路径长度，取较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。如果 (v_i,v_0,v_j) 较短，则取 (v_i,v_0,v_j) 作为最短路径。假如在路径上再增加一个顶点 v_1，也就是说，如果 $(v_i,...,v_1)$ 和 $(v_1,...,v_j)$ 分别是当前找到的中间定点的序号不大于 0 的最短路径，那么 $(vi,...,v1,...,vj)$ 就有可能是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径，将它和已经得到的从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径相比较，选出较短者作为从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径。再增加一个顶点 v_2，继续进行试探，依此类推。一般的，若 $(v_i,...,v_k)$ 和 $(v_k,...,v_j)$ 分别是从 v_i 到 v_k 和从 v_k 到 v_j 的中间定点的序号不大于 $k-1$ 的最短路径，则将 $(v_i,...,v_k,...,v_j)$ 和已经得到的从 v_i 到 v_j 的中间顶点的序号不大于 $k-1$ 的最短路径相比，较短者便是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径。这样，在经过 n 次比较后，最后求得的必是从 v_i 到 v_j 的最短路径。

现定义一个 n 阶方阵序列，

$$D^{(-1)},D^{(0)},D^{(1)},...,,D^{(k)},...,,D^{(n-1)}$$

其中，

$$\begin{aligned} D^{(-1)}[i][j] &= \text{g->matrix}[i][j], \\ D^{(k)}[i][j] &= \min\left\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\right\}, 0 \le k \le n-1 \end{aligned}$$

上述公式中，$D^{(k)}[i][j]$ 是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径的长度；$D^{(n-1)}[i][j]$ 是从 v_i 到 v_j 的最短路径的长度。

例如，对图15-4所示的有向图及其邻接矩阵运行 Floyd 算法，

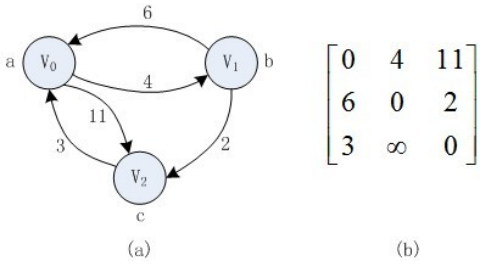


图 15-4 有向图及其邻接矩阵

运算过程中矩阵 D 的变化如表15-3所示。

表 15-3 Floyd 算法过程中方阵和最短路径的变化

D	$\mathbf{D^{(0)}}$			$\mathbf{D^{(1)}}$			$\mathbf{D^{(2)}}$			$\mathbf{D^{(3)}}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0	0	4	11	0	4	11	0	4	6	0	4	6
1	6	0	2	6	0	2	6	0	2	5	0	2
2	3	∞	0	3	7	0	3	7	0	3	7	0
P	$\mathbf{P^{(0)}}$			$\mathbf{P^{(1)}}$			$\mathbf{P^{(2)}}$			$\mathbf{P^{(3)}}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0		A B	A C		AB	A C		AB	AB C		AB	AB C
1	B A		B C	B A		B C	B A		BC	BC A		BC
2	C A			C A	CA B		C A	CA B		CA	CA B	

Floyd 算法的 C 语言实现如下。

代码

am_graph_floyd.cpp

```
#include <iostream>
#include <climits>  /* for INT_MAX */

using namespace std;

const int MAX_NV = 100; // 顶点数的最大值
/** 边的权值. */
typedef int graph_weight_t;
const int GRAPH_INF = INT_MAX / 2;   // 确保加法不溢出

/**
 *@struct 图，用邻接矩阵.
```

```
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接矩阵, 存放边的权重
    graph_weight_t matrix[MAX_NV][MAX_NV];
};


graph_t g;
/** dist[i][j] 是顶点 i 和 j 之间最短路径长度 */
graph_weight_t dist[MAX_NV][MAX_NV];
/** path[i][j] 是最短路径上 i 和 j 之间的顶点 */
int path[MAX_NV][MAX_NV];


/*
  * @brief Floyd 算法求每点之间最短路径.
  * @param[in] g 图对象的指针
  * @param[out] dist dist[i][j] 是顶点 i 和 j 之间最短路径长度
  * @param[out] path path[i][j] 是最短路径上 i 和 j 之间的顶点
  * @return 无
  */
void floyd(const graph_t &g,
       graph_weight_t dist[][MAX_NV],
       int path[][MAX_NV]) {
    int i, j, k;
    const int n = g.nv;

    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(i != j) {
                dist[i][j] = g.matrix[i][j];
                path[i][j] = i;
            } else {
                dist[i][j] = 0;
                path[i][j] = -1;
            }
        }
    }
    for(k = 0; k < n; k++) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                // i 到 j 的路径上加入顶点 k 可以缩短路径长度
                if(dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    path[i][j] = k;
                }
            }
        }
    }
}

/*
 * @brief 打印从 u 到 v 的最短路径
 * @param[in] u 起点
 * @param[in] v 终点
 * @param[in] path Floyd 计算好的 path
 * @return 无
 */
static void print_path_r(int u, int v, const int path[][MAX_NV]) {
    if (path[u][v] == -1) {
        cout << (char)('A' + u);
    } else {
```

```
        print_path_r(u, path[u][v], path);
        cout << "->" << (char)('A' + v);
    }
}

/**
 * @brief 打印 u 到其他所有点的最短路径
 * @param[in] path Dijkstra 计算好的 path
 * @param[in] n path 的长度
 * @return 无
 */
void print_path(const graph_t &g, const int path[][MAX_NV]) {
    for (int i = 0; i < g.nv; i++) {
        for (int j = 0; j < g.nv; j++) {
            if (i != j) {
                print_path_r(i, j, path);
                cout << endl;
            }
        }
        cout << endl;
    }
}

/**
 * @brief 读取输入，构建图.
 */
void read_graph(graph_t &g) {
    cin >> g.nv >> g.ne;

    // 初始化图，所有节点间距离为无穷大
    for (int i = 0; i < g.nv; i++) {
        for (int j = 0; j < g.nv; j++) {
            g.matrix[i][j] = GRAPH_INF;
        }
    }

    for (int k = 0; k < g.ne; k++) { // 读取边信息
        char chx, chy;
        graph_weight_t w;

        cin >> chx >> chy >> w;
        g.matrix[chx - 'A'][chy - 'A'] = w;
    }
}

/* test

输入数据:
3 5
A B 4
A C 11
B A 6
B C 2
C A 3

输出:
A->B
A->B->C

B->C->A
B->C

C->A
C->A->B
```

```
*/
int main() {
    read_graph(g);
    /* 求两两之间的最短路径 */
    floyd(g, dist, path);
    print_path(g, path);
    return 0;
}
```

am_graph_floyd.cpp

算法分析

该算法中有两个并列的 for 循环，第一个循环是个二重循环，用于初始化方阵 D；第二个循环是个三重循环，逐步生成 $D^{(0)}, D^{(1)}, ..., D^{(n-1)}$。所以算法总的时间复杂度为 $O(n^3)$。

Dijkstra 算法权值不能为负，Floyd 权值可以为负，但环路之和不能为负。

15.4.4 HDU 2544 最短路

描述

在每年的校赛里，所有进入决赛的同学都会获得一件很漂亮的 t-shirt。但是每当我们的工作人员把上百件的衣服从商店运回到赛场的时候，却是非常累的！所以现在他们想要寻找最短的从商店到赛场的路线，你可以帮助他们吗?

输入

输入包括多组数据。每组数据第一行是两个整数 $N, M(N \leq 100, M \leq 10000)$，$N$ 表示成都的大街上有几个路口，标号为 1 的路口是商店所在地，标号为 N 的路口是赛场所在地，M 则表示在成都有几条路。$N = M = 0$ 表示输入结束。接下来 M 行，每行包括 3 个整数 $A, B, C(1 \leq A, B \leq N, 1 \leq C \leq 1000)$, 表示在路口 A 与路口 B 之间有一条路，我们的工作人员需要 C 分钟的时间走过这条路。输入保证至少存在 1 条商店到赛场的路线。

输出

对于每组输入，输出一行，表示工作人员从商店走到赛场的最短时间

样例输入

```
2 1
1 2 3
3 3
1 2 5
2 3 5
3 1 2
0 0
```

样例输出

```
3
2
```

分析

单源最短路径，用 Dijkstra 算法，将第 §15.4.1节中的代码稍加修改即可。

注意，街道是双向的，所以给边赋值时要对称赋值。

代码

hdu_2544.cpp

```
// HDU 2544 最短路, http://acm.hdu.edu.cn/showproblem.php?pid=2544
/** 边的权值类型. */
typedef int graph_weight_t;

/** 顶点的编号, 可以为 char, int, string 等. */
typedef int graph_vertex_id_t;

// 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 "al_graph_dijkstra.cpp"  // 见“图-> 最短路径” 这节

/** 读取输入, 构建图. */
void read_graph(graph_t &g) {
    cin >> g.nv >> g.ne;
    for (int i = 0; i < g.ne; ++i) {
        graph_vertex_id_t from, to;
        graph_weight_t weight;
        cin >> from >> to >> weight;
        g.matrix[from][to] = weight;
        g.matrix[to][from] = weight;  // 街道是双向的
    }
}


int main() {
    while(true) {
        graph_t g;
        map<graph_vertex_id_t, graph_weight_t> distance;
        map<graph_vertex_id_t, graph_vertex_id_t> father;

        read_graph(g);
        if (g.nv == 0 && g.ne == 0) break;

        dijkstra(g, 1, distance, father);
        cout << distance[g.nv] << endl;
    }

    return 0;
}
```

hdu_2544.cpp

相关的题目

与本题相同的题目：

- HDU 2544 最短路, http://acm.hdu.edu.cn/showproblem.php?pid=2544

与本题相似的题目：

- POJ 2253 Frogger, http://poj.org/problem?id=2253
- POJ 3268 Silver Cow Party, http://poj.org/problem?id=3268
- POJ 1797 Heavy Transportation, http://poj.org/problem?id=1797
- POJ 1847 Tram, http://poj.org/problem?id=1847

15.4.5 POJ 1125 Stockbroker Grapevine

描述

Stockbrokers are known to overreact to rumours. You have been contracted to develop a method of spreading disinformation amongst the stockbrokers to give your employer the tactical edge in the stock market. For maximum effect, you have to spread the rumours in the fastest possible way.

Unfortunately for you, stockbrokers only trust information coming from their "Trusted sources" This means you have to take into account the structure of their contacts when starting a rumour. It takes a certain amount of time for a specific stockbroker to pass the rumour on to each of his colleagues. Your task will be to write a program that tells you which stockbroker to choose as your starting point for the rumour, as well as the time it will take for the rumour to spread throughout the stockbroker community. This duration is measured as the time needed for the last person to receive the information.

输入

Your program will input data for different sets of stockbrokers. Each set starts with a line with the number of stockbrokers. Following this is a line for each stockbroker which contains the number of people who they have contact with, who these people are, and the time taken for them to pass the message to each person. The format of each stockbroker line is as follows: The line starts with the number of contacts (n), followed by n pairs of integers, one pair for each contact. Each pair lists first a number referring to the contact (e.g. a '1' means person number one in the set), followed by the time in minutes taken to pass a message to that person. There are no special punctuation symbols or spacing rules.

Each person is numbered 1 through to the number of stockbrokers. The time taken to pass the message on will be between 1 and 10 minutes (inclusive), and the number of contacts will range between 0 and one less than the number of stockbrokers. The number of stockbrokers will range from 1 to 100. The input is terminated by a set of stockbrokers containing 0 (zero) people.

输出

For each set of data, your program must output a single line containing the person who results in the fastest message transmission, and how long before the last person will receive any given message after you give it to this person, measured in integer minutes.

It is possible that your program will receive a network of connections that excludes some persons, i.e. some people may be unreachable. If your program detects such a broken network, simply output the message "disjoint". Note that the time taken to pass the message from person A to person B is not necessarily the same as the time taken to pass it from B to A, if such transmission is possible at all.

样例输入

```
3
2 2 4 3 5
2 1 2 3 6
2 1 2 2 2
5
3 4 4 2 8 5 3
1 5 8
4 1 6 4 10 2 7 5 2
0
2 2 5 1 5
7
2 2 6 7 1
0
3 1 5 2 8 4 7
0
2 2 9 4 10
2 3 8 4 7
2 5 8 6 3
7
2 2 6 7 8
0
3 1 5 2 8 4 7
0
2 2 9 4 10
2 3 8 4 7
2 5 8 6 3
0
```

样例输出

```
3 2
3 10
1 12
7 17
```

分析

用 Floyd 算法求出每点之间的最短路径，输出距离最大的。

代码

poj_1125.cpp

```
// poj 1125 Stockbroker Grapevine, http://poj.org/problem?id=1125

// 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 "am_graph_floyd.cpp"  // 见“图-> 最短路径”这节

// 1. 修改范围
const int MAX_NV = 100; // 顶点数的最大值
// 2. 添加 max_len 数组
/** 存放每个点的最短路径的最长者 */
graph_weight_t max_len[MAX_NV];
// 3. 重写 read_graph()
// 4. 重写 main()

/** 读取输入，构建图. */
void read_graph(graph_t &g) {
    cin >> g.nv;
    g.ne = 0;

    /* 初始化图，所有节点间距离为无穷大 */
    for (int i = 0; i < g.nv; i++) {
        for (int j = 0; j < g.nv; j++) {
            g.matrix[i][j] = GRAPH_INF;
        }
    }

    /* 读取边信息 */
    for (int i = 0; i < g.nv; i++) {
        int m;
        cin >> m;
        g.ne += m;
        while (m--) {
            int j;
            graph_weight_t w;
            cin >> j >> w;
            --j;
            g.matrix[i][j] = w;
        }
    }
}

int main() {
    while (true) {
        read_graph(g);
        if (g.nv == 0) break;

        /* 求两两之间的最短路径 */
        floyd(g, dist, path);

        /* 找最短路径的最长着 */
        for (int i = 0; i < g.nv; i++) {
```

```
            max_len[i] = dist[i][distance(&dist[i][0],
                    max_element(&dist[i][0], &dist[i][0] + g.nv))];
        }
        /* 找 max_len 的最小者   */
        const int min_pos = distance(&max_len[0],
                min_element(&max_len[0], &max_len[0] + g.nv));

        if (max_len[min_pos] == GRAPH_INF) {
            cout << "disjoint" << endl;
        } else {
            cout << min_pos + 1 << " " << max_len[min_pos] << endl;
        }
    }
    return 0;
}
```

poj_1125.cpp

相关的题目

与本题相同的题目：

- POJ 1125 Stockbroker Grapevine, http://poj.org/problem?id=1125

与本题相似的题目：

- POJ 3615 Cow Hurdles, http://poj.org/problem?id=3615
- POJ 3660 Cow Contest, http://poj.org/problem?id=3660
- POJ 2502 Subway, http://poj.org/problem?id=2502
- HDU 3631 Shortest Path, http://acm.hdu.edu.cn/showproblem.php?pid=3631

15.5 拓扑排序

由某个集合上的一个偏序得到该集合上的一个全序，这个操作称为**拓扑排序**。

拓扑序列的特点是：若有向边 $< V_i, V_j >$ 是途中的弧，则在序列中顶点 V_i 必须排在顶点 V_j 之前。

如果用有向图表示一个工程，顶点表示活动，用有向边 $< v_i, v_j >$ 表示活动必须先于活动进行。这种有向图叫做顶点表示活动的网络 (Activity On Vertext Network)，简称 **AOV 网络**。

检测 AOV 网络是否存在环的方法是对 AOV 网络构造其顶点的拓扑有序序列。拓扑排序的基本步骤是：

1. 在有向图中选一个没有前驱的顶点且输出之；
2. 从图中删除该顶点和所有以它为尾的弧线。

重复以上两步，直至全部顶点输出，或当前图中不存在无前驱的顶点为止（这种情况说明图中存在环）。

拓扑排序的 C 语言实现如下。

代码

am_graph_topo_sort.cpp

```
#include <iostream>
#include <climits>  // for INT_MAX
#include <stack>

using namespace std;

/** 顶点数的最大值 */
const int MAX_NV = 100;
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
const graph_weight_t GRAPH_INF = INT_MAX;

/**
 *@struct
```

```
 *@brief 邻接矩阵.
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接矩阵, 存放边的信息, 如权重等
    graph_weight_t matrix[MAX_NV][MAX_NV];
};

graph_t g;
/** 拓扑排序的结果. */
int topological[MAX_NV];

/*
  * @brief 拓扑排序.
  * @param[in] g 图对象的指针
  * @param[out] topological 保存拓扑排序的结果
  * @return 无环返回 true, 有环返回 false
  */
bool topo_sort(const graph_t &g, int topological[]) {
    const int n = g.nv;
    int *in_degree = new int[n](); // in_degree[i] 是顶点 i 的入度

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (g.matrix[i][j] < GRAPH_INF)
                in_degree[j]++;
        }
    }

    stack<int> s;
    for(int i = 0; i < n; i ++) {
        if(in_degree[i] == 0)
            s.push(i);
    }

    int count = 0; /* 拓扑序列的元素个数 */
    while(!s.empty()) {
        const int u = s.top(); s.pop();
        topological[count++] = u;
        for (int i = 0; i < n; i++) if (g.matrix[u][i] < GRAPH_INF) {
            if(--in_degree[i] == 0) s.push(i);
        }
    }

    delete[] in_degree;
    if(count != n) { /* 有环 */
        return false;
    } else { /* 无环 */
        return true;
    }
}

/** 读取输入, 构建图. */
void read_graph() {
    /* 读取节点和边的数目 */
    cin >> g.nv >> g.ne;

    /* 初始化图, 所有节点间距离为无穷大 */
    for (int i = 0; i < g.nv; i++) {
        for (int j = 0; j < g.nv; j++) {
            g.matrix[i][j] = GRAPH_INF;
        }
    }
```

```
    /* 读取边信息 */
    for (int k = 0; k < g.ne; k++) {
        char chx, chy;
        graph_weight_t w;
        cin >> chx >> chy >> w;
        g.matrix[chx - 'A'][chy - 'A'] = w;
    }
}

/* test

输入数据:
6 8
A C 10
A E 30
A F 100
B C 5
C D 50
D 5 10
E D 20
E F 60

输出:
B A E F C D
*/
int main() {
    read_graph();
    /* 拓扑排序 */
    topo_sort(g, topological);
    for (int i = 0; i < g.nv; i++) {
        cout << (char)('A' + topological[i]) << " ";
    }
    return 0;
}
```

am_graph_topo_sort.cpp

算法分析

对有 n 个顶点和 e 条边的 AOV 网络而言，求各顶点的入度所需时间为 $O(e)$，建立零入度顶点栈所需时间为 $O(n)$；在拓扑排序过程中，若有向图无环，每个顶进一次栈出一次栈，顶点入度减 1 的操作共执行了 e 次。所以总的时间复杂度为 $O(n+e)$。

当有向图中无环时，也可以利用深度优先搜索进行拓扑排序。因为图中无环，深度优先遍历不会死循环。进行深度优先遍历时，最先退出 DFS 函数的顶点即为出度为零的顶点，是拓扑有序序列的最后一个顶点。由此，按退出 DFS 函数的先后次序记录下来的顶点序列即为逆向的拓扑有序序列。

15.5.1 POJ 1094 Sorting It All Out

描述

An ascending sorted sequence of distinct values is one in which some form of a less-than operator is used to order the elements from smallest to largest. For example, the sorted sequence A, B, C, D implies that $A < B, B < C$ and $C < D$. in this problem, we will give you a set of relations of the form $A < B$ and ask you to determine whether a sorted order has been specified or not.

输入

Input consists of multiple problem instances. Each instance starts with a line containing two positive integers n and m. the first value indicated the number of objects to sort, where $2 \leq n \leq 26$. The objects to be sorted will be the first n characters of

the uppercase alphabet. The second value m indicates the number of relations of the form $A < B$ which will be given in this problem instance. Next will be m lines, each containing one such relation consisting of three characters: an uppercase letter, the character "<" and a second uppercase letter. No letter will be outside the range of the first n letters of the alphabet. Values of $n = m = 0$ indicate end of input.

输出

For each problem instance, output consists of one line. This line should be one of the following three:

Sorted sequence determined after xxx relations: $yyy...y$.

Sorted sequence cannot be determined.

Inconsistency found after xxx relations.

where xxx is the number of relations processed at the time either a sorted sequence is determined or an inconsistency is found, whichever comes first, and $yyy...y$ is the sorted, ascending sequence.

样例输入

```
4 6
A<B
A<C
B<C
C<D
B<D
A<B
3 2
A<B
B<A
26 1
A<Z
6 6
A<F
B<D
C<E
F<D
D<E
E<F
0 0
```

样例输出

```
Sorted sequence determined after 4 relations: ABCD.
Inconsistency found after 2 relations.
Sorted sequence cannot be determined.
Inconsistency found after 6 relations.
```

分析

根据题目的要求，我们要每输入一次就要进行一次拓扑排序 `topological_sort()`，这样才能做到不成功（即发现有环）时，能知道是哪步不成功，并且给出输出。

还有要注意的就是如果我们可以提前判断结果了，但后面还有输入没完成，那么我们必须继续完成输入，不然剩下的输入会影响下一次 case 的输入。

代码

poj_1094.cpp

```
// POJ 1094 Sorting It All Out, http://poj.org/problem?id=1094
#include <iostream>
#include <climits>  // for INT_MAX
#include <stack>
```

```
using namespace std;

/** 顶点数的最大值 */
const int MAX_NV  = 26;
/** 边的权值. */
typedef int graph_weight_t;
const int GRAPH_INF = INT_MAX;

/**
 *@struct 邻接矩阵.
 */
struct graph_t {
    int nv; // 顶点数
    int ne; // 边数
    // 邻接矩阵, 存放边的信息, 如权重等
    graph_weight_t matrix[MAX_NV][MAX_NV];
};

graph_t g;
/** 拓扑排序的结果. */
int topological[MAX_NV];

/**
  * @brief 拓扑排序.
  * @param[in] g 图对象的指针
  * @param[out] topological 保存拓扑排序的结果
  * @return 无环返回 1, 有环返回 0, 条件不足返回 -1
  */
int topo_sort(const graph_t *g, int topological[]) {
    const int n = g->nv;

    /* in_degree[i] 是顶点 i 的入度 */
    int *in_degree = new int[n]();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (g->matrix[i][j] < GRAPH_INF)
                in_degree[j]++;
        }
    }

    stack<int> s;
    for(int i = 0; i < n; i ++) {
        if(in_degree[i] == 0) {
            s.push(i);
        }
    }

    int count = 0; // 拓扑序列的元素个数
    bool insufficient = false;  // 条件不足
    while(!s.empty()) {
        // 栈内应该始终只有一个元素
        if (s.size() > 1) insufficient = true;
        // 删除顶点 u
        const int u = s.top(); s.pop();
        topological[count++] = u;
        --in_degree[u];  // 变成 -1, 表示已经输出
        // 更新入度
        for (int i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INF) {
            --in_degree[i];
        }
        // 选择入度为 0 的顶点
        for (int i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INF) {
            if (in_degree[i] == 0) s.push(i);
        }
```

```
    }

    delete[] in_degree;
    if(count < n) { // 有环
        return 0;
    } else { // 无环
        if (insufficient) {  // 有孤立点，说明条件不足
            return -1;
        } else {
            return 1;
        }
    }
}

int main() {
    int m;  // m 不一定是边的数目，因为输入边可能有重复

    /* 读取节点和边的数目 */
    while ((cin >> g.nv >> m) && g.nv > 0 && m > 0) {
        bool finished = false;  // 排序完成，结束，发现有环，可以提前结束

        // 初始化图，所有节点间距离为无穷大
        for (int i = 0; i < g.nv; i++)
            for (int j = 0; j < g.nv; j++)
                g.matrix[i][j] = GRAPH_INF;

        // 读取边信息
        for (int k = 0; k < m; k++) {
            string s;
            cin >> s;
            g.matrix[s[0] - 'A'][s[2] - 'A'] = 1;

            if (finished) continue;    // 完成，则 continue，消耗输入

            // 是否有环，0 表示有环
            const int ok = topo_sort(&g, topological);

            if (ok == 0) {  // 有环存在
                cout << "Inconsistency found after " << k+1 <<
                        " relations." << endl;
                finished = true;  // 提前结束，记住要继续消耗输入
            }
            if (ok == 1 && k) {
                cout << "Sorted sequence determined after " << k+1
                        << " relations: ";
                for (int i = 0; i < g.nv; i++) {
                    cout << (char)('A' + topological[i]);
                }
                cout << "." << endl;
                finished = true;
            }
            // ok==-1, continue
        }
        if (!finished) {
            cout << "Sorted sequence cannot be determined." << endl;
        }
    }
    return 0;
}
```

poj_1094.cpp

相关的题目

与本题相同的题目：

- POJ 1094 Sorting It All Out, http://poj.org/problem?id=1094

与本题相似的题目：

- POJ 3267 The Cow Lexicon, http://poj.org/problem?id=3267
- POJ 3687 Labeling Balls, http://poj.org/problem?id=3687

15.6 关键路径

用有向边上的权值表示活动的持续时间，用顶点表示时间，这样的有向图叫做边表示的活动网络 (Activity On Edge Network)，简称 **AOE 网络**。

路径最长的路径叫做**关键路径** (Critical Path)。假设开始点为 v_1，从 v_1 到 v_i 的最长路径长度叫做事件 v_i 的最早发生时间。这个事件决定了所有以 v_i 为尾的弧所表示的活动的最早开始时间。我们用 $e(i)$ 表示活动 a_i 的最早开始时间。还可以定义一个活动的最迟开始时间 $l(i)$，这是在不推迟整个工程完成的前提下，活动 a_i 最迟必须开始进行的时间。两者之差 $l(i)-e(i)$ 意味着完成活动 a_i 的时间余量。我们把 $l(i)=e(i)$ 的活动叫做关键活动。

设活动 a_i 由弧 $<j,k>$ 表示，为了求得活动的 $e(i)$ 和 $l(i)$，首先应求得事件的最早发生时间 $ve(j)$ 和最迟发生时间 $vl(j)$，其持续时间记为 $dut(<j,k>)$，则有如下关系

$$\begin{aligned} e(i) &= ve(j) \\ l(i) &= vl(k)-dut(<j,k>) \end{aligned}$$

求 $ve(j)$ 和 $vl(k)$ 需分两步进行：

1. 从 $ve(0)=0$ 开始向前递推

$$ve(j)=\max\{ve(i)+dut(<i,j>)\},<i,j>\in T$$

其中 T 是所有以顶点 j 为弧头的边的集合。

2. 从 $vl(n-1)=ve(n-1)$ 起向后递推

$$vl(j)=\min\{vl(k)-dut(<j,k>)\},<j,k>\in S$$

其中 S 是所有以顶点 j 为弧尾的边的集合。

例如，对图15-5(a) 所示 AOE 网络的计算过程如表15-4所示，可见 a_2、a_5 和 a_7 为关键活动，组成一条从起点到终点的关键路径，如图15-5(b) 所示。

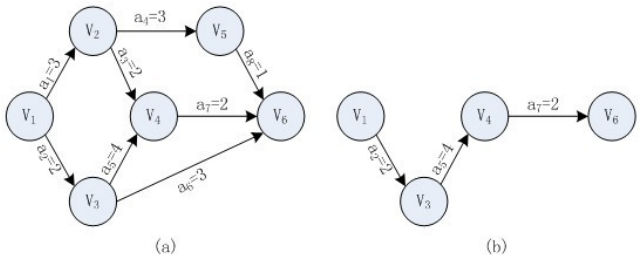


图 15-5 有向图及其邻接矩阵

表 15-4 图15-5(a) 所示 AOE 网络的关键路径的计算过程

顶点	ve		vl	活动	e	l	l-e
v_1	0		0	a_1	0	1	1
v_2	3		4	a_2	0	0	0
v_3	2		2	a_3	3	4	1
v_4	6	↓	6	a_4	3	4	1
v_5	6		7	a_5	2	2	0
v_6	8		8	a_6	2	5	3
				a_7	6	6	0
				a_8	6	7	1

邻接矩阵上的关键路径的 C 语言实现如下。

代码

am_graph_critical_path.cpp

```
#include <iostream>
#include <climits>  /* for INT_MAX */
#include <stack>
#include <algorithm>

using namespace std;

/** 顶点数的最大值 */
const int MAX_NV = 100;
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
const graph_weight_t GRAPH_INF = INT_MAX;

/**
 *@struct 邻接矩阵.
 */
struct graph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_NV][MAX_NV];
};

graph_t g;
/** 拓扑排序的结果. */
int topological[MAX_NV];
/** 关键路径，其余顶点为-1. */
int path[MAX_NV];

/*
  * @brief 按照拓扑排序的顺序，计算所有顶点的最早发生时间 ve.
  * @param[in] g 图对象的指针
  * @param[out] topological 保存拓扑排序的结果
  * @param[out] ve 所有事件的最早发生时间
  * @return 无环返回 true，有环返回 false
  */
static bool toposort_ve(const graph_t &g, int topological[],
        graph_weight_t ve[]) {
    const int n = g.nv;

    /* in_degree[i] 是顶点 i 的入度 */
    int *in_degree = new int[n]();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (g.matrix[i][j] < GRAPH_INF)
                in_degree[j]++;
        }
    }

    stack<int> s;
    for(int i = 0; i < n; i ++) {
        if(in_degree[i] == 0) {
            s.push(i);
        }
    }
    fill(ve, ve + n, 0);

    int count = 0; // 拓扑序列的元素个数
    while(!s.empty()) {
```

```
        // 删除顶点 u
        const int u = s.top(); s.pop();
        topological[count++] = u;
        --in_degree[u];  // 变成 -1，表示已经输出
        // 更新入度
        for (int i = 0; i < n; i++) if (g.matrix[u][i] < GRAPH_INF) {
            --in_degree[i];
        }
        // 更新邻接点的 ve
        for (int i = 0; i < n; i++) if (g.matrix[u][i] < GRAPH_INF) {
            if (ve[i] < ve[u] + g.matrix[u][i])
                ve[i] = ve[u] + g.matrix[u][i];
        }
        // 选择入度为 0 的顶点
        for (int i = 0; i < n; i++) if (g.matrix[u][i] < GRAPH_INF) {
            if (in_degree[i] == 0) s.push(i);
        }
    }

    delete[] in_degree;

    if(count < n) return false; // 有环
    else          return true;  // 无环
}

/**
  * @brief 求关键路径，第一个顶点为起点，最后一个顶点为终点.
  * @param[in] g 图对象的指针
  * @param[out] ve 所有事件的最早发生时间
  * @param[inout] path 关键路径
  * @return 无环返回关键路径的顶点个数，有环返回 0
  */
int critical_path(const graph_t &g, int path[MAX_NV]) {
    int count = 0;    // 关键路径的顶点个数
    graph_weight_t *ve = new graph_weight_t[g.nv];
    graph_weight_t *vl = new graph_weight_t[g.nv];

    if (!toposort_ve(g, topological, ve)) return 0;  // 有环

    for (int i = 0; i < MAX_NV; i++) path[i] = -1;
    // 初始化 vl 为最大
    for (int i = 0; i < g.nv; i++) vl[i] = ve[g.nv-1];

    // 逆序计算 vl
    for (int i = g.nv-1; i >=0; i--) {
        const int k = topological[i];
        for (int j = 0; j < g.nv; j++) {
            if (g.matrix[j][k] < GRAPH_INF) {
                if (vl[j] > vl[k] - g.matrix[j][k])
                    vl[j] = vl[k] - g.matrix[j][k];
            }
        }
    }
    for (int i = 0; i < g.nv; i++) {
        for (int j = 0; j < g.nv; j++) {
            const int e = ve[i];
            const int l = vl[j] - g.matrix[i][j];
            if (e == l) {
                if (i == 0) {
                    path[count++] = i;
                    path[count++] = j;
                } else {
                    path[count++] = j;
                }
```

```
            }
        }
    }

    delete[] ve;
    delete[] vl;
    return count;
}

/** 读取输入，构建图. */
void read_graph() {
    cin>> g.nv >> g.ne;

    // 初始化图，所有节点间距离为无穷大
    for (int i = 0; i < g.nv; i++)
        for (int j = 0; j < g.nv; j++)
            g.matrix[i][j] = GRAPH_INF;

    for (int k = 0; k < g.ne; k++) {  // 读取边信息
        char chx, chy;
        graph_weight_t w;
        cin >> chx >> chy >> w;
        g.matrix[chx - 'A'][chy - 'A'] = w;
    }
}

/* test

输入数据:
6 8
A B 3
A C 2
C D 4
B D 2
C F 3
B E 3
E F 1
D F 2

输出: A C D F

*/
int main() {
    read_graph();
    /* 拓扑排序 */
    const int count = critical_path(g, path);
    for (int i = 0; i < count; i++) {
        cout << (char)('A' + path[i]) << " ";
    }
    return 0;
}
```

am_graph_critical_path.cpp

算法分析

一次正向，复杂度为 $O(n^2)$，一次逆向，复杂度为 $O(n^2)$，因此，该算法的复杂度为 $O(n^2)$。

15.7 Clone Graph

描述

Clone an undirected graph. Each node in the graph contains a `label` and a list of its `neighbours`.

OJ's undirected graph serialization: Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbour of the node. As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

1. First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
2. Second node is labeled as 1. Connect node 1 to node 2.
3. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

```
1
/ \
/   \
0 --- 2
/ \
\_/
```

分析

广度优先遍历或深度优先遍历都可以。

DFS

```
// LeetCode, Clone Graph
// DFS, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
    public:
    UndirectedGraphNode *cloneGraph(const UndirectedGraphNode *node) {
        if(node == nullptr) return nullptr;
        // key is original node, value is copied node
        unordered_map<const UndirectedGraphNode *,
        UndirectedGraphNode *> copied;
        clone(node, copied);
        return copied[node];
    }
    private:
    // DFS
    static UndirectedGraphNode* clone(const UndirectedGraphNode *node,
    unordered_map<const UndirectedGraphNode *,
    UndirectedGraphNode *> &copied) {
        // a copy already exists
        if (copied.find(node) != copied.end()) return copied[node];

        UndirectedGraphNode *new_node = new UndirectedGraphNode(node->label);
        copied[node] = new_node;
        for (auto nbr : node->neighbors)
        new_node->neighbors.push_back(clone(nbr, copied));
        return new_node;
    }
};
```

BFS

```
// LeetCode, Clone Graph
// BFS, 时间复杂度 O(n), 空间复杂度 O(n)
class Solution {
    public:
    UndirectedGraphNode *cloneGraph(const UndirectedGraphNode *node) {
        if (node == nullptr) return nullptr;
        // key is original node, value is copied node
        unordered_map<const UndirectedGraphNode *,
```

```
            UndirectedGraphNode *> copied;
            // each node in queue is already copied itself
            // but neighbors are not copied yet
            queue<const UndirectedGraphNode *> q;
            q.push(node);
            copied[node] = new UndirectedGraphNode(node->label);
            while (!q.empty()) {
                const UndirectedGraphNode *cur = q.front();
                q.pop();
                for (auto nbr : cur->neighbors) {
                    // a copy already exists
                    if (copied.find(nbr) != copied.end()) {
                        copied[cur]->neighbors.push_back(copied[nbr]);
                    } else {
                    UndirectedGraphNode *new_node =
                    new UndirectedGraphNode(nbr->label);
                    copied[nbr] = new_node;
                    copied[cur]->neighbors.push_back(new_node);
                    q.push(nbr);
                }
            }
        }
        return copied[node];
    }
};
```

相关题目

- 无

第 16 章
细节实现题

这类题目不考特定的算法，纯粹考察写代码的熟练度。

16.1 Reverse Integer

描述

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

Have you thought about this?

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

Throw an exception? Good, but what if throwing an exception is not an option? You would then have to re-design the function (ie, add an extra parameter).

分析

短小精悍的题，代码也可以写的很短小。

代码

```
//LeetCode, Reverse Integer
// 时间复杂度 O(logn)，空间复杂度 O(1)
class Solution {
public:
    int reverse (int x) {
        int r = 0;

        for (; x; x /= 10)
            r = r * 10 + x % 10;

        return r;
    }
};
```

相关题目

- Palindrome Number, 见 §??

16.2 Palindrome Number

描述

Determine whether an integer is a palindrome. Do this without extra space.

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

分析

首先想到，可以利用上一题，将整数反转，然后与原来的整数比较，是否相等，相等则为 Palindrome 的。可是 reverse() 会溢出。

正确的解法是，不断地取第一位和最后一位（10 进制下）进行比较，相等则取第二位和倒数第二位，直到完成比较或者中途找到了不一致的位。

代码

```
//LeetCode, Palindrome Number
// 时间复杂度 O(1)，空间复杂度 O(1)
class Solution {
public:
    bool isPalindrome(int x) {
        if (x < 0) return false;
        int d = 1; // divisor
        while (x / d >= 10) d *= 10;

        while (x > 0) {
            int q = x / d;  // quotient
            int r = x % 10;   // remainder
            if (q != r) return false;
            x = x % d / 10;
            d /= 100;
        }
        return true;
    }
};
```

相关题目

- Reverse Integer, 见 §??
- Valid Palindrome, 见 §3.12.2

16.3 Insert Interval

描述

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1: Given intervals `[1,3],[6,9]`, insert and merge `[2,5]` in as `[1,5],[6,9]`.

Example 2: Given `[1,2],[3,5],[6,7],[8,10],[12,16]`, insert and merge `[4,9]` in as `[1,2],[3,10],[12,16]`.

This is because the new interval `[4,9]` overlaps with `[3,5],[6,7],[8,10]`.

分析

无

代码

```
struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) { }
    Interval(int s, int e) : start(s), end(e) { }
};

//LeetCode, Insert Interval
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {
            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            } else if (newInterval.start > it->end) {
                it++;
                continue;
            } else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

相关题目

- Merge Intervals，见 §??

16.4 Merge Intervals

描述

Given a collection of intervals, merge all overlapping intervals.

For example, Given `[1,3],[2,6],[8,10],[15,18]`, return `[1,6],[8,10],[15,18]`

分析

复用一下 Insert Intervals 的解法即可，创建一个新的 interval 集合，然后每次从旧的里面取一个 interval 出来，然后插入到新的集合中。

代码

```
struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) { }
    Interval(int s, int e) : start(s), end(e) { }
};

//LeetCode, Merge Interval
//复用一下 Insert Intervals 的解法即可
// 时间复杂度 O(n1+n2+...)，空间复杂度 O(1)
```

```
class Solution {
public:
    vector<Interval> merge(vector<Interval> &intervals) {
        vector<Interval> result;
        for (int i = 0; i < intervals.size(); i++) {
            insert(result, intervals[i]);
        }
        return result;
    }
private:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {
            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            } else if (newInterval.start > it->end) {
                it++;
                continue;
            } else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

相关题目

- Insert Interval，见 §??

16.5 Minimum Window Substring

描述

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

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

Minimum window is `"BANC"`.

Note:

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

分析

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

代码

```
// LeetCode, Minimum Window Substring
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    string minWindow(string S, string T) {
        if (S.empty()) return "";
```

```
        if (S.size() < T.size()) return "";

        const int ASCII_MAX = 256;
        int appeared_count[ASCII_MAX];
        int expected_count[ASCII_MAX];
        fill(appeared_count, appeared_count + ASCII_MAX, 0);
        fill(expected_count, expected_count + ASCII_MAX, 0);

        for (size_t i = 0; i < T.size(); i++) expected_count[T[i]]++;

        int minWidth = INT_MAX, min_start = 0;  // 窗口大小，起点
        int wnd_start = 0;
        int appeared = 0;  // 完整包含了一个 T
        //尾指针不断往后扫
        for (size_t wnd_end = 0; wnd_end < S.size(); wnd_end++) {
            if (expected_count[S[wnd_end]] > 0)  {  // this char is a part of T
                appeared_count[S[wnd_end]]++;
                if (appeared_count[S[wnd_end]] <= expected_count[S[wnd_end]])
                    appeared++;
            }
            if (appeared == T.size()) {  // 完整包含了一个 T
                // 收缩头指针
                while (appeared_count[S[wnd_start]] > expected_count[S[wnd_start]]
                        || expected_count[S[wnd_start]] == 0) {
                    appeared_count[S[wnd_start]]--;
                    wnd_start++;
                }
                if (minWidth > (wnd_end - wnd_start + 1)) {
                    minWidth = wnd_end - wnd_start + 1;
                    min_start = wnd_start;
                }
            }
        }

        if (minWidth == INT_MAX) return "";
        else return S.substr(min_start, minWidth);
    }
};
```

相关题目

- 无

16.6 Multiply Strings

描述

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

分析

高精度乘法。

常见的做法是将字符转化为一个 int，一一对应，形成一个 int 数组。但是这样很浪费空间，一个 int32 的最大值是 $2^{31}-1=2147483647$，可以与 9 个字符对应，由于有乘法，减半，则至少可以与 4 个字符一一对应。一个 int64 可以与 9 个字符对应。

代码 1

```
// LeetCode, Multiply Strings
// @author 连城 (http://weibo.com/lianchengzju)
```

```
// 一个字符对应一个 int
// 时间复杂度 O(n*m), 空间复杂度 O(n+m)
typedef vector<int> bigint;

bigint make_bigint(string const& repr) {
    bigint n;
    transform(repr.rbegin(), repr.rend(), back_inserter(n),
            [](char c) { return c - '0'; });
    return n;
}

string to_string(bigint const& n) {
    string str;
    transform(find_if(n.rbegin(), prev(n.rend()),
            [](char c) { return c > '\0'; }), n.rend(), back_inserter(str),
            [](char c) { return c + '0'; });
    return str;
}

bigint operator*(bigint const& x, bigint const& y) {
    bigint z(x.size() + y.size());

    for (size_t i = 0; i < x.size(); ++i)
        for (size_t j = 0; j < y.size(); ++j) {
            z[i + j] += x[i] * y[j];
            z[i + j + 1] += z[i + j] / 10;
            z[i + j] %= 10;
        }

    return z;
}

class Solution {
public:
    string multiply(string num1, string num2) {
        return to_string(make_bigint(num1) * make_bigint(num2));
    }
};
```

代码 2

```
// LeetCode, Multiply Strings
// 9 个字符对应一个 int64_t
// 时间复杂度 O(n*m/81), 空间复杂度 O((n+m)/9)
/** 大整数类. */
class BigInt {
public:
    /**
     * @brief 构造函数, 将字符串转化为大整数.
     * @param[in] s 输入的字符串
     * @return 无
     */
    BigInt(string s) {
        vector<int64_t> result;
        result.reserve(s.size() / RADIX_LEN + 1);

        for (int i = s.size(); i > 0; i -= RADIX_LEN) {  // [i-RADIX_LEN, i)
            int temp = 0;
            const int low = max(i - RADIX_LEN, 0);
            for (int j = low; j < i; j++) {
                temp = temp * 10 + s[j] - '0';
            }
            result.push_back(temp);
        }
```

```
        elems = result;
    }
    /**
     * @brief 将整数转化为字符串.
     * @return 字符串
     */
    string toString() {
        stringstream result;
        bool started = false; // 用于跳过前导 0
        for (auto i = elems.rbegin(); i != elems.rend(); i++) {
            if (started) { // 如果多余的 0 已经都跳过, 则输出
                result << setw(RADIX_LEN) << setfill('0') << *i;
            } else {
                result << *i;
                started = true; // 碰到第一个非 0 的值, 就说明多余的 0 已经都跳过
            }
        }

        if (!started) return "0"; // 当 x 全为 0 时
        else return result.str();
    }

    /**
     * @brief 大整数乘法.
     * @param[in] x x
     * @param[in] y y
     * @return 大整数
     */
    static BigInt multiply(const BigInt &x, const BigInt &y) {
        vector<int64_t> z(x.elems.size() + y.elems.size(), 0);

        for (size_t i = 0; i < y.elems.size(); i++) {
            for (size_t j = 0; j < x.elems.size(); j++) { // 用 y[i] 去乘以 x 的各位
                //  两数第 i, j 位相乘, 累加到结果的第 i+j 位
                z[i + j] += y.elems[i] * x.elems[j];

                if (z[i + j] >= BIGINT_RADIX) { //  看是否要进位
                    z[i + j + 1] += z[i + j] / BIGINT_RADIX; //  进位
                    z[i + j] %= BIGINT_RADIX;
                }
            }
        }
        while (z.back() == 0) z.pop_back();  // 没有进位, 去掉最高位的 0
        return BigInt(z);
    }

private:
    typedef long long int64_t;
    /** 一个数组元素对应 9 个十进制位, 即数组是亿进制的
     * 因为 1000000000 * 1000000000 没有超过 2^63-1
     */
    const static int BIGINT_RADIX = 1000000000;
    const static int RADIX_LEN = 9;
    /** 万进制整数. */
    vector<int64_t> elems;
    BigInt(const vector<int64_t> num) : elems(num) {}
};


class Solution {
public:
    string multiply(string num1, string num2) {
        BigInt x(num1);
        BigInt y(num2);
```

```
        return BigInt::multiply(x, y).toString();
    }
};
```

相关题目

- 无

16.7 Substring with Concatenation of All Words

描述

You are given a string, S, and a list of words, L, that are all of the same length. Find all starting indices of substring(s) in S that is a concatenation of each word in L exactly once and without any intervening characters.

For example, given:

```
S: "barfoothefoobarman"
L: ["foo", "bar"]
```

You should return the indices: `[0,9]`.(order does not matter).

分析

无

代码

```
// LeetCode, Substring with Concatenation of All Words
// 时间复杂度 O(n*m), 空间复杂度 O(m)
class Solution {
public:
    vector<int> findSubstring(string s, vector<string>& dict) {
        size_t wordLength = dict.front().length();
        size_t catLength = wordLength * dict.size();
        vector<int> result;

        if (s.length() < catLength) return result;

        unordered_map<string, int> wordCount;

        for (auto const& word : dict) ++wordCount[word];

        for (auto i = begin(s); i <= prev(end(s), catLength); ++i) {
            unordered_map<string, int> unused(wordCount);

            for (auto j = i; j != next(i, catLength); j += wordLength) {
                auto pos = unused.find(string(j, next(j, wordLength)));

                if (pos == unused.end() || pos->second == 0) break;

                if (--pos->second == 0) unused.erase(pos);
            }

            if (unused.size() == 0) result.push_back(distance(begin(s), i));
        }

        return result;
    }
};
```

相关题目

- 无

16.8 Pascal's Triangle

描述

Given $numRows$, generate the first $numRows$ of Pascal's triangle.

For example, given $numRows = 5$,

Return

```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

分析

本题可以用队列，计算下一行时，给上一行左右各加一个 0，然后下一行的每个元素，就等于左上角和右上角之和。

另一种思路，下一行第一个元素和最后一个元素赋值为 1，中间的每个元素，等于上一行的左上角和右上角元素之和。

从左到右

```
// LeetCode, Pascal's Triangle
// 时间复杂度 O(n^2)，空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > result;
        if(numRows == 0) return result;

        result.push_back(vector<int>(1,1)); //first row

        for(int i = 2; i <= numRows; ++i) {
            vector<int> current(i,1);  // 本行
            const vector<int> &prev = result[i-2];  // 上一行

            for(int j = 1; j < i - 1; ++j) {
                current[j] = prev[j-1] + prev[j]; // 左上角和右上角之和
            }
            result.push_back(current);
        }
        return result;
    }
};
```

从右到左

```
// LeetCode, Pascal's Triangle
// 时间复杂度 O(n^2)，空间复杂度 O(n)
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > result;
        vector<int> array;
        for (int i = 1; i <= numRows; i++) {
```

```
            for (int j = i - 2; j > 0; j--) {
                array[j] = array[j - 1] + array[j];
            }
            array.push_back(1);
            result.push_back(array);
        }
        return result;
    }
};
```

相关题目

- Pascal's Triangle II，见 §??

16.9 Pascal's Triangle II

描述

Given an index k, return the k^{th} row of the Pascal's triangle.

For example, given $k = 3$,

Return `[1,3,3,1]`.

Note: Could you optimize your algorithm to use only $O(k)$ extra space?

分析

滚动数组。

代码

```
// LeetCode, Pascal's Triangle II
// 滚动数组，时间复杂度 O(n^2)，空间复杂度 O(n)
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> array;
        for (int i = 0; i <= rowIndex; i++) {
            for (int j = i - 1; j > 0; j--){
                array[j] = array[j - 1] + array[j];
            }
            array.push_back(1);
        }
        return array;
    }
};
```

相关题目

- Pascal's Triangle，见 §??

16.10 Spiral Matrix

描述

Given a matrix of $m \times n$ elements (m rows, n columns), return all elements of the matrix in spiral order.

For example, Given the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
```

You should return `[1,2,3,6,9,8,7,4,5]`.

分析

模拟。

代码

```
// LeetCode, Spiral Matrix
// @author 龚陆安 (http://weibo.com/luangong)
// 时间复杂度 O(n^2)，空间复杂度 O(1)
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int> >& matrix) {
        vector<int> result;
        if (matrix.empty()) return result;
        int beginX = 0, endX = matrix[0].size() - 1;
        int beginY = 0, endY = matrix.size() - 1;
        while (true) {
            // From left to right
            for (int j = beginX; j <= endX; ++j) result.push_back(matrix[beginY][j]);
            if (++beginY > endY) break;
            // From top to bottom
            for (int i = beginY; i <= endY; ++i) result.push_back(matrix[i][endX]);
            if (beginX > --endX) break;
            // From right to left
            for (int j = endX; j >= beginX; --j) result.push_back(matrix[endY][j]);
            if (beginY > --endY) break;
            // From bottom to top
            for (int i = endY; i >= beginY; --i) result.push_back(matrix[i][beginX]);
            if (++beginX > endX) break;
        }
        return result;
    }
};
```

相关题目

- Spiral Matrix II， 见 §??

16.11 Spiral Matrix II

描述

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

For example, Given $n = 3$,

You should return the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

分析

这题比上一题要简单。

代码 1

```
// LeetCode, Spiral Matrix II
// 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector<vector<int> > matrix(n, vector<int>(n));
        int begin = 0, end = n - 1;
        int num = 1;

        while (begin < end) {
            for (int j = begin; j < end; ++j) matrix[begin][j] = num++;
            for (int i = begin; i < end; ++i) matrix[i][end] = num++;
            for (int j = end; j > begin; --j) matrix[end][j] = num++;
            for (int i = end; i > begin; --i) matrix[i][begin] = num++;
            ++begin;
            --end;
        }

        if (begin == end) matrix[begin][begin] = num;

        return matrix;
    }
};
```

代码 2

```
// LeetCode, Spiral Matrix II
// @author 龚陆安 (http://weibo.com/luangong)
// 时间复杂度 O(n^2), 空间复杂度 O(n^2)
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector< vector<int> > matrix(n, vector<int>(n));
        if (n == 0) return matrix;
        int beginX = 0, endX = n - 1;
        int beginY = 0, endY = n - 1;
        int num = 1;
        while (true) {
            for (int j = beginX; j <= endX; ++j) matrix[beginY][j] = num++;
            if (++beginY > endY) break;

            for (int i = beginY; i <= endY; ++i) matrix[i][endX] = num++;
            if (beginX > --endX) break;

            for (int j = endX; j >= beginX; --j) matrix[endY][j] = num++;
            if (beginY > --endY) break;

            for (int i = endY; i >= beginY; --i) matrix[i][beginX] = num++;
            if (++beginX > endX) break;
        }
        return matrix;
    }
};
```

相关题目

- Spiral Matrix, 见 §??

16.12 ZigZag Conversion

描述

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

分析

要找到数学规律。真正面试中，不大可能出这种问题。

n=4:

```
P     I     N
A   L S   I G
Y A   H R
P     I
```

n=5:

```
P       H
A     S I
Y   I   R
P L     I  G
A       N
```

所以，对于每一层垂直元素的坐标 $(i, j) = (j + 1) * n + i$；对于每两层垂直元素之间的插入元素（斜对角元素），$(i, j) = (j + 1) * n - i$

代码

```
// LeetCode, ZigZag Conversion
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    string convert(string s, int nRows) {
        if (nRows <= 1 || s.size() <= 1) return s;
        string result;
        for (int i = 0; i < nRows; i++) {
            for (int j = 0, index = i; index < s.size();
                    j++, index = (2 * nRows - 2) * j + i) {
                result.append(1, s[index]);  // 垂直元素
                if (i == 0 || i == nRows - 1) continue;   // 斜对角元素
                if (index + (nRows - i - 1) * 2 < s.size())
                    result.append(1, s[index + (nRows - i - 1) * 2]);
            }
        }
        return result;
    }
};
```

相关题目

- 无

16.13 Divide Two Integers

描述

Divide two integers without using multiplication, division and mod operator.

分析

不能用乘、除和取模，那剩下的，还有加、减和位运算。

最简单的方法，是不断减去被除数。在这个基础上，可以做一点优化，每次把被除数翻倍，从而加速。

代码 1

```
// LeetCode, Divide Two Integers
// 时间复杂度 O(logn)，空间复杂度 O(1)
class Solution {
public:
    int divide(int dividend, int divisor) {
        // 当 dividend = INT_MIN 时，-dividend 会溢出，所以用 long long
        long long a = dividend >= 0 ? dividend : -(long long)dividend;
        long long b = divisor >= 0 ? divisor : -(long long)divisor;

        // 当 dividend = INT_MIN 时，divisor = -1 时，结果会溢出，所以用 long long
        long long result = 0;
        while (a >= b) {
            long long c = b;
            for (int i = 0; a >= c; ++i, c <<= 1) {
                a -= c;
                result += 1 << i;
            }
        }

        return ((dividend^divisor) >> 31) ? (-result) : (result);
    }
};
```

代码 2

```
// LeetCode, Divide Two Integers
// 时间复杂度 O(logn)，空间复杂度 O(1)
class Solution {
public:
    int divide(int dividend, int divisor) {
        int result = 0; // 当 dividend = INT_MIN 时，divisor = -1 时，结果会溢出
        const bool sign = (dividend > 0 && divisor < 0) ||
                (dividend < 0 && divisor > 0); // 异号

        // 当 dividend = INT_MIN 时，-dividend 会溢出，所以用 unsigned int
        unsigned int a = dividend >= 0 ? dividend : -dividend;
        unsigned int b = divisor >= 0 ? divisor : -divisor;

        while (a >= b) {
            int multi = 1;
            unsigned int bb = b;
            while (a >= bb) {
                a -= bb;
                result += multi;

                if (bb < INT_MAX >> 1) { // 防止溢出
                    bb += bb;
                    multi += multi;
                }
            }
```

```
        }
        if (sign) return -result;
        else return result;
    }
};
```

相关题目

- 无

16.14 Text Justification

描述

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

words: `["This", "is", "an", "example", "of", "text", "justification."]`

L: 16.

Return the formatted lines as:

```
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

Corner Cases:

- A line other than the last line might contain only one word. What should you do in this case?
- In this case, that line should be left

分析

无

代码

```
// LeetCode, Text Justification
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    vector<string> fullJustify(vector<string> &words, int L) {
        vector<string> result;
        const int n = words.size();
        int begin = 0, len = 0; // 当前行的起点，当前长度
        for (int i = 0; i < n; ++i) {
            if (len + words[i].size() + (i - begin) > L) {
                result.push_back(connect(words, begin, i - 1, len, L, false));
                begin = i;
                len = 0;
            }
            len += words[i].size();
```

```
        }
        // 最后一行不足 L
        result.push_back(connect(words, begin, n - 1, len, L, true));
        return result;
    }
    /**
     * @brief 将 words[begin, end] 连成一行
     * @param[in] words 单词列表
     * @param[in] begin 开始
     * @param[in] end 结束
     * @param[in] len words[begin, end] 所有单词加起来的长度
     * @param[in] L 题目规定的一行长度
     * @param[in] is_last 是否是最后一行
     * @return 对齐后的当前行
     */
    string connect(vector<string> &words, int begin, int end,
            int len, int L, bool is_last) {
        string s;
        int n = end - begin + 1;
        for (int i = 0; i < n; ++i) {
            s += words[begin + i];
            addSpaces(s, i, n - 1, L - len, is_last);
        }

        if (s.size() < L) s.append(L - s.size(), ' ');
        return s;
    }

    /**
     * @brief 添加空格.
     * @param[inout]s 一行
     * @param[in] i 当前空隙的序号
     * @param[in] n 空隙总数
     * @param[in] L 总共需要添加的空额数
     * @param[in] is_last 是否是最后一行
     * @return 无
     */
    void addSpaces(string &s, int i, int n, int L, bool is_last) {
        if (n < 1 || i > n - 1) return;
        int spaces = is_last ? 1 : (L / n + (i < (L % n) ? 1 : 0));
        s.append(spaces, ' ');
    }
};
```

相关题目

- 无

16.15 Max Points on a Line

描述

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

分析

暴力枚举法。两点决定一条直线，n 个点两两组合，可以得到 $\frac{1}{2}n(n+1)$ 条直线，对每一条直线，判断 n 个点是否在该直线上，从而可以得到这条直线上的点的个数，选择最大的那条直线返回。复杂度 $O(n^3)$。

上面的暴力枚举法以“边”为中心，再看另一种暴力枚举法，以每个“点”为中心，然后遍历剩余点，找到所有的斜率，如果斜率相同，那么一定共线对每个点，用一个哈希表，key 为斜率，value 为该直线上的点数，计算出哈希表后，取最大值，并更新全局最大值，最后就是结果。时间复杂度 $O(n^2)$，空间复杂度 $O(n)$。

以边为中心

```
// LeetCode, Max Points on a Line
// 暴力枚举法，以边为中心，时间复杂度 O(n^3)，空间复杂度 O(1)
class Solution {
public:
    int maxPoints(vector<Point> &points) {
        if (points.size() < 3) return points.size();
        int result = 0;

        for (int i = 0; i < points.size() - 1; i++) {
            for (int j = i + 1; j < points.size(); j++) {
                int sign = 0;
                int a, b, c;
                if (points[i].x == points[j].x) sign = 1;
                else {
                    a = points[j].x - points[i].x;
                    b = points[j].y - points[i].y;
                    c = a * points[i].y - b * points[i].x;
                }
                int count = 0;
                for (int k = 0; k < points.size(); k++) {
                    if ((0 == sign && a * points[k].y == c +  b * points[k].x) ||
                        (1 == sign&&points[k].x == points[j].x))
                        count++;
                }
                if (count > result) result = count;
            }
        }
        return result;
    }
};
```

以点为中心

```
// LeetCode, Max Points on a Line
// 暴力枚举，以点为中心，时间复杂度 O(n^2)，空间复杂度 O(n)
class Solution {
public:
    int maxPoints(vector<Point> &points) {
        if (points.size() < 3) return points.size();
        int result = 0;

        unordered_map<double, int> slope_count;
        for (int i = 0; i < points.size()-1; i++) {
            slope_count.clear();
            int samePointNum = 0; // 与 i 重合的点
            int point_max = 1;    // 和 i 共线的最大点数

            for (int j = i + 1; j < points.size(); j++) {
                double slope; // 斜率
                if (points[i].x == points[j].x) {
                    slope = std::numeric_limits<double>::infinity();
                    if (points[i].y == points[j].y) {
                        ++ samePointNum;
                        continue;
                    }
                } else {
                    slope = 1.0 * (points[i].y - points[j].y) /
                        (points[i].x - points[j].x);
                }

                int count = 0;
                if (slope_count.find(slope) != slope_count.end())
```

```
                    count = ++slope_count[slope];
                else {
                    count = 2;
                    slope_count[slope] = 2;
                }

                if (point_max < count) point_max = count;
            }
            result = max(result, point_max + samePointNum);
        }
        return result;
    }
};
```

相关题目

- 无

第 17 章
数学方法与常见模型

数学对于算法很重要。没有好的数学基础，很难在算法上达到一定高度。本章介绍算法竞赛中常用的数学方法和模型。

17.1 数论

17.1.1 欧几里德算法

求最大公约数 (greates common divisor) 有很多方法，最经典的方法是欧几里德算法 (Euclidean algorithm[①])，又称辗转相除法。

gcd.c

```
/**
 * @brief 求最大公约数，欧几里德算法，也即辗转相除法
 *
 * @param[in] a a
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
unsigned int gcd(unsigned int a, unsigned int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

/**
 * @brief 求最大公约数，欧几里德算法，迭代版本
 *
 * @param[in] a a
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
unsigned int gcd1(unsigned int a, unsigned int b) {
    while (b != 0) {
        unsigned int tmp = b;
        b = a % b;
        a = tmp;
    }
    return a;
}

/**
 * @brief 求最大公约数，欧几里德算法，迭代版本，基于减法
 *
 * @param[in] a a
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
unsigned int gcd2(unsigned int a, unsigned int b) {
    while (a != b) {
        if (a > b) {
```

[①] http://en.wikipedia.org/wiki/Euclid_algorithm

```
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

gcd.c

求出了最大公约数，可以利用它来求最小公倍数 (least common multiple), $\text{lcm}(a,b) = a \times b / \gcd(a,b)$。

例题

- wikioi 1212 最大公约数，http://www.wikioi.com/problem/1212/

17.1.2 扩展欧几里德算法

定理：对于不完全为 0 的非负整数 a, b，必然存在整数对 x, y ，使得 $\gcd(a,b) = ax + by$。

这里 x 和 y 不一定是正数。扩展欧几里德算法 (Extended Euclidean algorithm[①]) 就是用来求 x 和 y 的。

ex_gcd.c

```
/**
 * @brief 扩展欧几里德算法
 * @param[in] a a
 * @param[in] b b
 * @param[out] x
 * @param[out] y
 * @return gcd(a,b)
 */
unsigned int ex_gcd(unsigned int a, unsigned int b, int *x, int *y) {
    if(b == 0) {
        *x = 1; *y = 0; return a;
    } else {
        const unsigned int tmp = ex_gcd(b, a % b, y, x);
        *y -= (*x)*(a/b);
        return tmp;
    }
}
```

ex_gcd.c

扩展欧几里德算法的应用主要有以下三个：

- 求解不定方程；
- 求解模线性方程（线性同余方程）；
- 求解模的逆元；

17.1.3 求解不定方程

求不定方程 $ax + by = c$ 的整数解 x, y，其中 $a > b \geq c \geq 0$。

设 $g = \gcd(a,b)$，方程 $ax + by = g$ 的一组解是 (x_0, y_0)，则当 c 是 g 的倍数时 $ax + by = c$ 的一组解是 (x_0, y_0)。证明略。

```
/**
 * @brief 求解不定方程 ax+by=c
 * @param[in] a a
 * @param[in] b b
 * @param[in] c c
 * @param[out] x x
 * @param[out] y y
 * @return 是否有解，1 表示有解，0 表示无解
 */
```

[①] http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

```
int linear_equation(unsigned int a, unsigned int b, unsigned int c,
        int *x, int *y) {
    unsigned int k;
    const unsigned int g = ex_gcd(a, b, x, y);
    if (c % g) return 0;

    k = c / g;
    (*x) *= k; (*y) *= k;
    return 1;
}
```

17.1.4 求解模线性方程

解方程 $ax \equiv b \bmod n$，其中 a, b, n 是正整数。

$ax \equiv b \bmod n$ 的含义是“ax 和 b 除以 n 的余数相同”，设这个倍数为 y，则 $ax - b = ny$，这恰好就是前面介绍过的不定方程。接下来的步骤就不用说了吧。

```
/**
 * @brief 求解模线性方程 ax = b mod n
 * @param[in] a
 * @param[in] b
 * @param[in] n n>0
 */
int modular_linear_equation(unsigned int a, unsigned int b, unsigned int n) {
    int x, y, x0, i;
    unsigned int k;
    unsigned int g = ex_gcd(a, n, &x, &y);
    if(b % g) return 0;

    k = b / g;
    x0 = (k * x) % n;   //特解
    for (i = 0; i < g; i++)
        printf("%d\n", (x0 + i * (n/g)) % n);
    return 1;
}
```

17.1.5 求解模的逆元

有一个特殊情况要引起读者重视，当 $b = 1$ 时，$ax \equiv 1 \bmod n$ 的解称为 a 关于模 n 的逆 (inverse)，它类似于“倒数”的概念。什么时候 a 的逆存在呢？根据上面的讨论，方程 $ax - ny = 1$ 要有解，这样，1 必须是 $\gcd(a, n)$ 的倍数，因此 a 和 n 必须互素。

17.1.6 素数判定

素数判定，又称素数测试 (Primality test[①])，即给定一个正整数 n，判断它是否是素数。

暴力枚举法

从 2 到 n，依次作为除数，让 n 除以它们，只要有一个能整除 n，则 n 不是素数。

```
/**
 * @brief 判断正整数 n 是否是素数
 * @param[in] n 正整数
 * @return 是，返回 1，否，返回 0
 */
int is_prime(unsigned int n) {
    int i;
    if (n < 2) return 0;
    for (i = 2; i < n; i++) {
```

[①] http://en.wikipedia.org/wiki/Primality_test

```
        if (n % i == 0) return 0;
    }
    return 1;
}
```

可以稍微做一点改进，从 1 到 $\sqrt{n}$，依次作为除数。

```
/**
 * @brief 判断正整数 n 是否是素数，上界改为 sqrt(n)
 * @param[in] n 正整数
 * @return 是，返回 1，否，返回 0
 */
int is_prime(unsigned int n) {
    int i;
    if (n < 2) return 0;
    const int upper = sqrt(n);

    for (i = 2; i <= upper; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

/**
 * @brief 判断正整数 n 是否是素数，上界改为 sqrt(n)，但不使用 sqrt() 函数
 * @param[in] n 正整数
 * @return 是，返回 1，否，返回 0
 */
int is_prime1(unsigned int n) {
    int i;
    if (n < 2) return 0;
    for (i = 2; i*i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}
```

Eratosthenes 筛法

更高效的素数判定方法应该是预先计算出一张素数表，当判断一个数是否是素数时，直接查表即可。

怎样计算？用 Eratosthenes 筛法 (Sieve of Eratosthenes[①])

给出要筛数值的范围 n，找出 $\sqrt{n}$ 以内的素数 $p_1, p_2, \ldots, p_k$。先用 2 去筛，即把 2 留下，把 2 的倍数剔除掉；再用下一个质数，也就是 3 筛，把 3 留下，把 3 的倍数剔除掉；接下去用下一个质数 5 筛，把 5 留下，把 5 的倍数剔除掉；不断重复下去……。

```
#define MAXN 30000
/** prime_table[i]==1 表示 i 是素数，等于 0 则不是素数 */
int prime_table[MAXN+1];

void compute_prime_table() {
    int i, j;
    const int upper = sqrt(MAXN);

    for (i = 2; i <= MAXN; i++) prime_table[i] = 1;
    prime_table[0] = 0;
    prime_table[1] = 0;

    for (i = 2; i < upper; i++) if(prime_table[i]) {
        for (j = 2; j * i <= MAXN; j++) prime_table[j*i] = 0;
    }
}
```

[①] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

```
int is_prime(unsigned int n) {
    return prime_table[n];
}
```

暴力枚举法优化版

下面这段代码是从 GNU GMP(http://gmplib.org/) 的源代码中抠出来的。

```
int is_prime(unsigned int n) {
    unsigned int q, r, d;    /* 商，余数，除数 */

    /* 把这个常数展开成二进制就容易理解了 */
    if (n < 32) return (0xa08a28acUL >> n) & 1;
    if ((n & 1) == 0) return 0;
    if (n % 3 == 0) return 0;
    if (n % 5 == 0) return 0;
    if (n % 7 == 0) return 0;

    for (d = 11;;) {
        q = n / d;
        r = n - q * d;
        if (q < d) return 1;    /* 保证 d 不超过 sqrt(n) */
        if (r == 0) break;

        d += 2;
        q = n / d;
        r = n - q * d;
        if (q < d) return 1;
        if (r == 0) break;

        d += 4;
    }
    return 0;
}
```

例题

- wikioi 1430 素数判定，http://www.wikioi.com/problem/1430/

17.1.7 大整数取模

描述

求 $a \bmod b, 1 \leq a \leq 10^{1000}, 1 \leq b \leq 100000$。

输入

每行 2 个整数 a 和 b

输出

对每一行输出结果 $a \bmod b$

样例输入

```
2 3
12 7
152455856554521 3250
```

样例输出

```
2
5
1521
```

代码

bigint_mod.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (1000/MOD_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * MOD_LEN];
int     x[MAX_LEN], y;

/**
 * @brief 将输入的字符串转化为大整数，用数组表示，低位在低地址.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    /* for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0'; */
    for (i = len; i > 0; i -= MOD_LEN) {  /* [i-MOD_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-MOD_LEN > 0 ? i-MOD_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 计算 x mod y
 * @param[in] x 大整数
 * @param[in] y 模
 * @return x mod y
 */
int bigint_mod(const int x[MAX_LEN], int y) {
    int ret = 0;
    int i;
    for (i = MAX_LEN-1; i >= 0; i--) {
        ret = (ret * BIGINT_MOD + x[i]) % y;
    }
    return ret;
}

int main() {
    while(scanf("%s%d", a, &y) > 1) {
        bigint_input(a, x);
        printf("%d\n", bigint_mod(x, y));
    }
```

```
    return 0;
}
```
bigint_mod.c

相关的题目

与本题相同的题目：

- HDU 1212 Big Number, http://acm.hdu.edu.cn/showproblem.php?pid=1212

与本题相似的题目：

- None

17.1.8 单变元模线性方程

描述

已知 a,b,n, 求 x，使得 $ax \equiv b(\mod n)$.

分析

令 $d = gcd(a, n)$, 先使用扩展欧几里得求 ax+ny=d 的解。如果 b 不能整除 d 则无解，否则 mod n 意义下得解有 d 个，可以通过对某个解不断的加 n/d 得到。

代码

line_mod.c
```
//O(logn)
vector<long long> line_mod_equations(long long a,long long b,long long n){
    long long x,y;
    long long d=gcd(a,n,x,y);
    vector<long long> ans;
    if(d\%d==0){
        x\%=n; x+=n;x\%=n;
        ans.push_back(x*(b/d)\%(n/d));
        for(long long i=1;i<d;++i)
            ans.push_back((ans[0]+i\%n/d)\%n);
    }
    return ans;
}
```
line_mod.c

17.1.9 中国剩余定理

求出方程 $x \equiv a_i(modm_i), 0 \leq i < n$ 的解 x. 其中 $m_1, m_2, \cdot, m_{n-1}$ 两两互质。

分析

令 $M_i = \Pi_{j \neq i} m_j$, 因为 $(M_i, m_i) = 1$, 故存在 p_i, q_i, 使 $M_i p_i + m_i q_i = 1$。令 $e_i = M_i p_i$, 故 $e_0 a_0 + e_1 a_1 + e_2 a_2 + \cdots + e_{n-1} a_{n-1}$ 是方程的一个解。在 $[0, \Pi_{i=0}^{n-1})$，只有一个解。

代码

ctr.c
```
//O(nlogm)
ing ctr(int a[], int m[], int n){
    int M=1;
    for(int i=0;i<n; ++i){
        int x, y;
        int tm=M/m[i];
        extend_gcd(tm,m[i],x,y);
```

```
            ret=(ret+tm*x*a[i])\%M;
        }
        return (ret+M)%M;
    }
```

ctr.c

17.1.10 欧拉函数

欧拉函数表示小于或等于 n 的数中与 n 互质的数的数目。欧拉函数求值的方法是:
(1) $\phi(1) = 1$
(2) 若 n 是素数 p 的 k 次幂，$\phi(n) = p^k - p^{k-1} = (p-1)p^{k-1}$
(3) 若 m,n 互质，$\phi(nm) = \phi(n)\phi(m)$
因此，当 p 为 n 的最小质因数，若 $p^2|n, \phi(n) = \phi(\frac{n}{p}) \times p$; 否则 $\phi(n) = \phi(\frac{n}{p}) \times (p-1)$

17.2 组合数学

17.2.1 Sin Average

描述

What is the average of $sin^2(x)$?

思路

The easiest way to see it is to note that $sin^2(x) + cos^2(x) = 1$. Since $cos^2(x)$ is just $sin^2(x)$ shifted over by 90 degrees, the average of $cos^2(x)$ and the average of $sin^2(x)$ over a full period are the same. Since the two functions sum to 1, the average of each of them must therefore be $\frac{1}{2}$.

17.2.2 全排列散列

对于一个 n 的全排列，返回一个整数代表的它在所有排列中的排名。同样对于一个排名我们返回原排列，排名从 0 到 n！-1。

分析

把排列开成一个多进制数。第 i 位的进制 $(n-i+1)!$。设 $a[i] = x$, 前 $i-1$ 个有 k 个比 x 小. 那么这一位应该用 $(i-1-k) \times (n-i+1)!$ 来做为权值，最后加上所有位的权值即可。

用数求排列的时候，从高位到低位一位一位确定。用这一位的权值除以这一位对应的阶乘，若为 k，那么从当前还没有用过的数中找出第 k+1 小得即可。

第 18 章
大整数运算

在 32 位 CPU 下，C/C++ 中的 int 能表示的范围是 $[-2^{32}, 2^{32}-1]$，unsigned int 能表示的范围是 $[0, 2^{32}]$。所以，int 和 unsigned int 都不能保存超过 10 位的整数（解方程 $10^x \leq 2^{32}$，可得 $x \leq 9.63$）。有时我们需要参与运算的整数，可能会远远不止 10 位，我们称这种基本数据类型无法表示的整数为大整数。如何表示和存放大整数呢？基本的思想是：用数组模拟大整数。一个数组元素，存放大整数中的一位。

例如，一个 200 位的十进制整数，可以用 `int x[200]` 来表示，一个数组元素对应一个位。这样做有点浪费空间，因为一个 int 可以表示的范围远远大于 10。因此，我们可以用一个数组元素，表示 4 个数位（一个 int 可以表示的范围也远远大于 10000，为什么一个数组元素只表示 4 个数位，可不可以表示 9 个数位？留给读者思考），这时，数组不再是 10 进制，而是 10000 进制。使用万进制，数组长度可以缩减到原来的 1/4。

18.1 大整数加法

描述

求两个非负的大整数相加的和。

输入

有两行，每行是一个不超过 200 位的非负整数，可能有多余的前导 0。

输出

一行，即相加后的结果。结果里不能有多余的前导 0，即如果结果是 342，那么就不能输出为 0342。

样例输入

```
22222222222222222222
33333333333333333333
```

样例输出

```
55555555555555555555
```

代码

bigint_add.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (200/RADIX_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN + 1];

/**
 * @brief 打印大整数.
```

```
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数，用数组表示，低位在低地址
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= RADIX_LEN) {  /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 大整数加法
 * @param[in] x x
 * @param[in] y y
 * @param[out] z z=x+y
 * @return 无
 */
void bigint_add(const int x[], const int y[], int z[]) {
    int i;
    for (i = 0; i < MAX_LEN + 1; i++) z[i] = 0;

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相加 */
        z[i] += x[i] + y[i];
        if (z[i] >= BIGINT_RADIX) {  /* 看是否要进位 */
            z[i] -= BIGINT_RADIX;
            z[i+1] ++;  /* 进位 */
        }
    }
}
```

```
int main() {
    scanf("%s%s", a, b);

    bigint_input(a, x);
    bigint_input(b, y);

    bigint_add(x, y, z);
    bigint_print(z, MAX_LEN + 1);
    printf("\n");
    return 0;
}
```

bigint_add.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[①]第 144 页 7.1 节
- 百练 2981 大整数加法, http://poj.grids.cn/practice/2981/

与本题相似的题目：

- None

18.2 大整数减法

描述

求两个非负的大整数相减的差。

输入

第 1 行是测试数据的组数 T，每组测试数据占 2 行，第 1 行是被减数 a，第 2 行是减数 b(a > b)，每行数据不超过 100 个字符，没有多余的前导 0。每组测试数据之间有一个空行。

输出

每组测试数据输出一行，即相应的差

样例输入

```
2
99999999999999999999999999999999999999
9999999999999

540965677509785089568705679806897093454654657567676867843543534 5
1
```

分析

模拟小学生列竖式做加法，从个位开始逐位相加，超过或达到 10（这里用万进制，则是 10000）则进位。

两个 200 位的大整数相加，结果可能会有 201 位。

样例输出

```
99999999999999999999999990000000000000
540965677509785089568705679806897093454654657567676867843543534 4
```

[①]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

代码

bigint_sub.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (100/RADIX_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数，用数组表示，低位在低地址
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= RADIX_LEN) {  /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 大整数减法.
 *
 * @param[in] x x
 * @param[in] y y, x>y
```

```
 * @param[out] z z=x-y
 * @return 无
 */
void bigint_sub(const int x[], const int y[], int z[]) {
    int i;
    for (i = 0; i < MAX_LEN; i++) z[i] = 0;

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相减 */
        z[i] += x[i] - y[i];
        if (z[i] < 0) {  /* 看是否要借位 */
            z[i] += BIGINT_RADIX;
            z[i+1] --;  /* 借位 */
        }
    }
}

int main() {
    int T;
    scanf("%d", &T);

    while (T-- > 0) {
        scanf("%s%s",a,b);

        bigint_input(a, x);
        bigint_input(b, y);

        bigint_sub(x, y, z);
        bigint_print(z, MAX_LEN);
        printf("\n");
    }
    return 0;
}
```

bigint_sub.c

相关的题目

与本题相同的题目：

- 百练 2736 大整数减法, http://poj.grids.cn/practice/2736/

与本题相似的题目：

- None

18.3 大整数乘法

描述

求两个非负的大整数相乘的积。

输入

有两行，每行是一个不超过 200 位的非负整数，没有多余的前导 0。

输出

一行，即相乘后的结果。结果里不能有多余的前导 0。

样例输入

```
12345678900
98765432100
```

样例输出

```
1219326311126352690000
```

分析

两个 200 位的数相乘，积最多会有 400 位。

计算的过程基本上和小学生列竖式做乘法相同。为编程方便，并不急于处理进位，而将进位问题留待最后统一处理。

现以 835 × 49 为例来说明程序的计算过程。

先算 835 × 9。5 × 9 得到 45 个 1 ，3 × 9 得到 27 个 10，8 × 9 得到 72 个 100。由于不急于处理进位，所以 835 × 9 算完后，aResult 如下：

下标		5	4	3	2	1	0
aResult	····	0	0	0	72	27	45

接下来算 4 × 5 。此处 4 × 5 的结果代表 20 个 10 ，因此要 aResult[1]+=20，变为：

下标		5	4	3	2	1	0
aResult	····		0	0	72	**47**	45

再下来算 4 × 3。此处 4 × 3 的结果代表 12 个 100，因此要 aResult[2]+= 12，变为：

下标		5	4	3	2	1	0
aResult	····	0	0	0	**84**	47	45

最后算 4 × 8。此处 4 × 8 的结果代表 32 个 1000 ，因此要 aResult[3]+= 32，变为：

下标		5	4	3	2	1	0
aResult	····	0	0	**32**	84	47	45

乘法过程完毕。接下来从 aResult[0] 开始向高位逐位处理进位问题。aResult[0] 留下 5，把 4 加到 aResult[1] 上，aResult[1] 变为 51 后，应留下 1 ，把 5 加到 aResult[2] 上……最终使得 aResult 里的每个元素都是 1 位数，结果就算出来了：

下标		5	4	3	2	1	0
aResult	····	0	4	0	9	1	5

总结一个规律，即一个数的第 i 位和另一个数的第 j 位相乘所得的数，一定是要累加到结果的第 i+j 位上。这里 i, j 都是从右往左，从 0 开始数。

代码

bigint_mul.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (200/RADIX_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN * 2];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
```

```
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数，用数组表示，低位在低地址
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= RADIX_LEN) {  /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 大整数乘法.
 * @param[in] x x
 * @param[in] y y
 * @param[out] z z=x*y
 * @return 无
 */
void bigint_mul(const int x[], const int y[], int z[]) {
    int i, j;
    for (i = 0; i < MAX_LEN * 2; i++) z[i] = 0;

    for (i = 0; i < MAX_LEN; i++) {
        for (j = 0; j < MAX_LEN; j++) { /* 用 y[i]，去乘以 x 的各位 */
            z[i + j] += y[i] * x[j]; /* 两数第 i, j 位相乘，累加到结果的第 i+j 位 */

            if (z[i + j] >= BIGINT_RADIX) { /* 看是否要进位 */
                z[i + j + 1] += z[i + j] / BIGINT_RADIX; /* 进位 */
                z[i + j] %= BIGINT_RADIX;
            }
        }
    }
}
```

```
int main() {
    scanf("%s%s", a, b);

    bigint_input(a, x);
    bigint_input(b, y);

    bigint_mul(x, y, z);
    bigint_print(z, MAX_LEN * 2);
    printf("\n");
    return 0;
}
```

bigint_mul.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[①]第 146 页 7.2 节
- 百练 2980 大整数乘法, http://poj.grids.cn/practice/2980/

与本题相似的题目：

- None

18.4 大整数除法

描述

求两个非负的大整数相除的商。

输入

第 1 行是测试数据的组数 T，每组测试数据占 2 行，第 1 行是被除数，第 2 行是除数，每行数据不超过 100 个字符。每组测试数据之间有一个空行。

输出

每组测试数据输出一行，即相应的整数商

样例输入

```
3
240533731296337335900926045774205743923049649393035559579766079108273964 6
298719258531870175258442993116087037290707924897109501250979055083793197894

1000000000000000000000000000000000000000
10000000000

5409656775097850895687056798068970934546546575676768678435435345
1
```

样例输出

```
0
100000000000000000000000000000
5409656775097850895687056798068970934546546575676768678435435345
```

[①]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

分析

基本的思想是反复做减法，看看从被除数里最多能减去多少个除数，商就是多少。一个一个减显然太慢，如何减得更快一些呢？以 7546 除以 23 为例来看一下：开始商为 0。先减去 23 的 100 倍，就是 2300，发现够减 3 次，余下 646。于是商的值就增加 300。然后用 646 减去 230，发现够减 2 次，余下 186，于是商的值增加 20。最后用 186 减去 23，够减 8 次，因此最终商就是 328。

所以本题的核心是要写一个大整数的减法函数，然后反复调用该函数进行减法操作。

计算除数的 10 倍、100 倍的时候，不用做乘法，直接在除数后面补 0 即可。

代码

bigint_div.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (100/RADIX_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数，用数组表示，低位在低地址
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= RADIX_LEN) {  /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_LEN : 0;
        for (k = low; k < i; k++) {
```

```
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 计算大整数的位数.
 *
 * @param[inout] x 大整数
 * @return 位数
 */
static int length(const int x[]) {
    int i;
    int result = 0;
    for (i = MAX_LEN - 1; i >= 0; i--) if (x[i] > 0) {
        result = i + 1;
        break;
    }
    return result;
}

/**
 * @brief 大整数减法.
 *
 * @param[inout] x x
 * @param[in] y y
 * @return 如果 x < y, 返回-1, 如果 x=y, 返回 0, 如果 x>y, 返回 1
 */
static int bigint_sub(int x[], const int y[]) {
    int i;
    const int lenx = length(x);
    const int leny = length(y);

    /* 判断 x 是否比 y 大 */
    if (lenx < leny) return -1;
    else if (lenx == leny) {
        int larger = 0;
        for (i = lenx - 1; i >= 0; i--) {
            if (x[i] > y[i]) {
                larger = 1;
            } else if (x[i] < y[i]) {
                if (!larger) return -1;
            }
        }
    }

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相减 */
        x[i] -= y[i];
        if (x[i] < 0) {  /* 看是否要借位 */
            x[i] += BIGINT_RADIX;
            x[i+1] --;  /* 借位 */
        }
    }

    return 1;
}

/**
 * @brief 大整数除法.
 *
 * @param[inout] x x
 * @param[in] y y
```

```
 * @param[out] z z=x/y
 * @return 无
 */
void bigint_div(int x[], const int y[], int z[]) {
    int i;
    int *yy; /* y 的副本 */
    const int xlen = length(x);
    int ylen = length(y);
    const int times = xlen - ylen;

    for (i = 0; i < MAX_LEN; i++) z[i] = 0;
    if (times < 0) return;

    yy = (int*)malloc(sizeof(int) * MAX_LEN);
    memcpy(yy, y, sizeof(int) * MAX_LEN);


    /* 将 yy 右移 times 位，使其长度和 x 相同，即 yy 乘以 10000 的 times 次幂 */
    for (i = xlen - 1; i >= 0; i--) {
        if (i >= times) yy[i] = yy[i - times];
        else yy[i] = 0;
    }

    /* 先减去若干个 y×(10000 的 times 次方)，
      不够减了，再减去若干个 y×(10000 的 times-1 次方)
      一直减到不够减为止 */
    ylen = xlen;
    for (i = 0; i <= times; i++) {
        int j;
        while (bigint_sub(x, yy) >= 0) {
            z[times - i]++;
        }

        /* yy 除以 BIGINT_RADIX，即左移一位 */
        for (j = 1; j < ylen; j++) {
            yy[j - 1] = yy[j];
        }
        yy[--ylen] = 0;
    }

    /* 下面的循环统一处理进位 */
    for (i = 0; i < MAX_LEN - 1; i++) {
        if (z[i] >= BIGINT_RADIX) {  /* 看是否要进位 */
            z[i+1] += z[i] / BIGINT_RADIX;  /* 进位 */
            z[i] %= BIGINT_RADIX;
        }
    }
    free(yy);
}

int main() {
    int T;
    scanf("%d", &T);

    while (T-- > 0) {
        scanf("%s%s",a,b);

        bigint_input(a, x);
        bigint_input(b, y);

        bigint_div(x, y, z);
        bigint_print(z, MAX_LEN);
        printf("\n");
    }
```

```
    return 0;
}
```
bigint_div.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[①]第 149 页 7.3 节
- 百练 2737 大整数除法, http://poj.grids.cn/practice/2737/
- wikioi 3118 高精度练习之除法, http://poj.grids.cn/practice/3118/

与本题相似的题目：

- None

18.5 大数阶乘

18.5.1 大数阶乘的位数

描述

求 $n!$ 的位数，$0 \le n \le 10^7$。

输入

第一行是一个正整数 T，表示测试用例的个数。接下来的 T 行，每行一个正整数 n。

输出

对每个 n，每行输出 $n!$ 的位数

样例输入

```
2
10
20
```

样例输出

```
7
19
```

分析

最简单的办法，是老老实实计算出 $n!$，然后就知道它的位数了。但这个方法很慢，会超时 (TLE)。

组合数学里有个 Stirling 公式 (Stirling's formula[②])：

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

可以用这个公式来计算 $n!$ 的位数，它等于

$$n \log \frac{n}{e} + \frac{1}{2} \log 2\pi n + 1$$

[①]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

[②]http://en.wikipedia.org/wiki/Stirling's_approximation

代码

bigint_factorial_digits.c

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

/**
 * @brief 计算 n 的阶乘的位数，用 Stirling 公式
 * @param[in] n n>=0
 * @return n 的阶乘的位数
 */
int factorial_digits(unsigned int n) {
    const double PI = 3.14159265358979323846;
    const double E = 2.7182818284590452354;
    if (n == 0) return 1;
    return (int)(n * log10(n/E) + 0.5 * log10(2*PI*n)) + 1;
}

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

    scanf("%d", &T);
    for (i = 0; i < T; i++) {
        scanf("%d", &n);
        printf("%d\n", factorial_digits(n));
    }
    return 0;
}
```

bigint_factorial_digits.c

相关的题目

与本题相同的题目：

- POJ 1423 Big Number, http://poj.org/problem?id=1423

与本题相似的题目：

- None

18.5.2 大数阶乘

描述

计算 $n!, 0 \leq n \leq 10000$)。

输入

每行一个整数 n

输出

对每个 n，每行输出 $n!$

样例输入

```
1
2
3
```

样例输出

```
1
2
6
```

代码

bigint_factorial.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_RADIX 10000
#define RADIX_LEN 4
/* 10000! 有 35660 位 */
#define MAX_LEN (35660/RADIX_LEN+1)  /* 整数的最大位数 */

int     x[MAX_LEN + 1];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 大整数乘法，x = x*y.
 * @param[inout] x x
 * @param[in] y y
 * @return 无
 */
void bigint_mul(int x[], const int y) {
    int i;
    int c = 0; /* 保存进位 */

    for (i = 0; i < MAX_LEN; i++) { /* 用 y，去乘以 x 的各位 */
        const int tmp = x[i] * y + c;
        x[i] = tmp % BIGINT_RADIX;
        c = tmp / BIGINT_RADIX;
    }
}

/**
 * @brief 计算 n 的阶乘
 * @param[in] n
 * @param[out] x 存放结果
 * @return 无
 */
```

```
void bigint_factorial(int n, int x[]) {
    int i;
    memset(x, 0, sizeof(int) * (MAX_LEN + 1));
    x[0] = 1;

    for (i = 2; i <= n; i++) {
        bigint_mul(x, i);
    }
}


int main() {
    int n;
    while (scanf("%d", &n) != EOF) {
        bigint_factorial(n, x);
        bigint_print(x, MAX_LEN + 1);
        printf("\n");
    }
    return 0;
}
```

bigint_factorial.c

相关的题目

与本题相同的题目：

- HDU 1042 N!, http://acm.hdu.edu.cn/showproblem.php?pid=1042

与本题相似的题目：

- None

第 19 章
基础功能

在面试和笔试中，经常会出现这类题目，把 C++, Java 标准库中的一些函数单独拿出来，让你重新实现。这类题目短小精悍，很能考验一个人的基本功是否扎实。

19.1 下一个排列

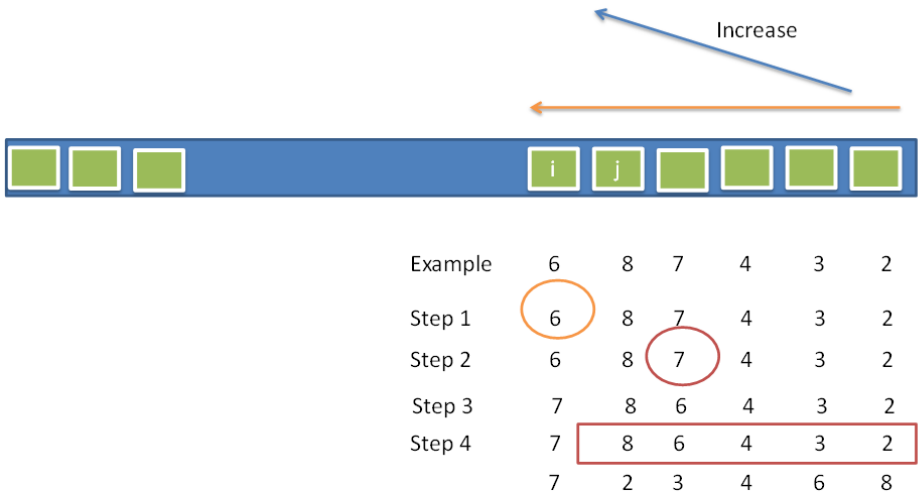
描述

实现 C++ STL 中的 `next_permutation()`，函数原型如下：

```
/**
 * @brief 返回下一个排列，例如当前排列是 12345，下一个是 12354
 * @param[inout] num 当前排列，例如 12345
 * @param[in] len num 的长度
 * @return 无
 */
void next_permutation(int num[], int len);
```

分析

算法过程如图 18-1所示（来自 http://fisherlei.blogspot.com/2012/12/leetcode-next-permutation.html）。



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

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

图 19-1　下一个排列算法流程

代码

next_permutation.c

```
static void swap(int array[], int i, int j) {
    const int tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
}

static void reverse(int array[], int first, int last) { //左闭右开区间
    last--;
    while (first < last)
        swap(array, first++, last--);
}

/**
 * @brief 返回下一个排列，例如当前排列是 12345，下一个是 12354
 * @param[inout] num 当前排列，例如 12345
 * @param[in] first 开始位置
 * @param[in] last 结束位置，左闭右开区间
 * @return 成功返回 0，失败返回 -1
 */
int next_permutation(int num[], int first, int last) {
    int i, j;

    i = last - 2;  // partition number's index
    while (i >= 0 && num[i] >= num[i + 1])
        i--;

    if (i == -1) {
        reverse(num, first, last);
        return -1;
    }

    j = last - 1;  // change number's index
    while (num[j] <= num[i])
        --j;
    swap(num, i, j);
    reverse(num, i + 1, last);

    return 0;
}
```

next_permutation.c

相关的题目

与本题相同的题目：

- LeetCode - Next Permutation, http://leetcode.com/onlinejudge#question_31

19.2 数组循环右移

描述

将一个长度为 n 的数组 A 的元素循环右移（ROR, Rotate Right）k 位，比如数组 1, 2, 3, 4, 5 循环右移 3 位之后就变成 3, 4, 5, 1, 2。

方法一

最直接的做法是另开一个大小一样的数组 B，遍历一下，令 $B[(i+k)\%n] = A[i]$，再将 B 的内容写回到 A 即可。这个方法的时间复杂度为 O(n)，空间复杂度也为 O(n)。代码如下：

ror.c
```
void ror1(int array[], int n, int k) {
    int i;
    int *B = (int*) malloc(n * sizeof(int));

    k %= n;
    if (k == 0)
        return;

    for (i = 0; i < n; i++) {
        B[(i + k) % n] = array[i];
    }
    for (i = 0; i < n; i++) {
        array[i] = B[i];
    }
}
```
ror.c

方法二

另一种简单的做法，每次将数组中的所有元素右移一位，循环 k 次。这个方法的时间复杂度为 O(n*k)，空间复杂度为 O(1)。代码如下：

ror.c
```
void ror2(int array[], int n, int k) {
    int i, tmp;
    k %= n;
    if (k == 0)
        return;

    while (k--) {
        tmp = array[n - 1];
        for (i = n - 1; i > 0; i--) {
            array[i] = array[i - 1];
        }
        array[0] = tmp;
    }
}
```
ror.c

方法三

先将 A 的元素倒置，即 1, 2, 3, 4, 5 变成 5, 4, 3, 2, 1，然后将前 k 位倒置，即 3, 4, 5, 2, 1，再将后 n-k 位倒置，即 3, 4, 5, 1, 2，完成。

证明：记 A 的前 n-k 位为 X，后 k 位为 Y，则 A=XY，将 A 循环右移 k 位后，应该得到 YX。根据该算法，先将 A 整体倒置，得到 $(XY)^T = Y^T X^T$，然后将前 k 位倒置，得到 YX^T，最后将后 n-k 位倒置，得到 YX，正好是所求的结果，证毕。

这个方法的时间复杂度为 O(2n)，空间复杂度为 O(1)。代码如下：

ror.c
```
static void swap(int array[], int i, int j) {
    const int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

static void reverse(int array[], int begin, int end) { //左闭右开区间
    end--;
    while (begin < end)
        swap(array, begin++, end--);
}

void ror3(int *array, int n, int k) {
```

```
    k %= n;
    if (k == 0)
        return;

    reverse(array, 0, n);
    reverse(array, 0, k);
    reverse(array, k, n - k);
}
```

ror.c

这种方法需要对每个位置写入 2 次，看上去也不怎么好，那有没有更好的呢？

方法四

我们要做的只是把每个元素放到它应该在的位置，比如开头的例子，1 应该放在 4 的位置，把 1 放好之后，4 就没地方了，那 4 应该在哪呢，在 2 的位置，依此类推，就可以把所有的元素都放好，而且只放了一次。看上去这样做很完美，但仔细想想就能想出反例子，比如 1, 2, 3, 4, 5, 6, 7, 8, 9 右移 3 位，就是 1 放在 4 个位置，4 放在 7 的位置，然后 7 放回 1，这时候一圈兜完了，但只排好了 3 个元素，剩下的 6 个元素没有动过，怎么办呢？继续下一个，就是 2，然后 2、5、8 也排好了，继续 3、6、9，这时候下一个元素是 1 了，应该停止了（因为 1、4、7 已经排好了），那程序怎么会知道停在这里了，于是就想到了最大公约数，9 和 3 的最大公约数是 3，于是做前 3 个数的循环就可以了，为什么上一个例子只需做一次，因为元素个数（5）和移动位数（3）互质。具体的数学证明略。

代码如下：

ror.c

```
static int gcd(int a, int b) {
    assert(a >= b);
    if (b == 0) {
        return a;
    }

    while (b > 0) {
        int tmp = a % b;
        a = b;
        b = tmp;
    }

    return a;
}

void ror4(int *array, int n, int k) {
    int i;
    const int g = gcd(n, k);

    k %= n;
    if (k == 0)
        return;

    for (i = 0; i < g; ++i) {
        int j = i;
        int cur = array[j], tmp;

        do {
            tmp = array[(j + k) % n];
            array[(j + k) % n] = cur;
            cur = tmp;
            j = (j + k) % n;
        } while (j != i);
    }
}

// test
int main(void) {
```

```
    int i;
    int a[] = { 1, 2, 3, 4, 5 };
    ror4(a, 5, 3);
    for (i = 0; i < 5; i++) {
        printf("%d ", a[i]);
    }

    return EXIT_SUCCESS;
}
```

ror.c

第 20 章
Parallel Programming

【Note】Asyn(), thread, ref, mutable, mutex

在进行多线程编程时，难免还要碰到两个问题，那就线程间的互斥与同步：线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步（下文统称为同步）。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

下面我们来分别看一下这些方法：

临界区（Critical Section）

保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

临界区包含两个操作原语：EnterCriticalSection () 进入临界区 LeaveCriticalSection () 离开临界区

EnterCriticalSection () 语句执行后代码将进入临界区以后无论发生什么，必须确保与之匹配的 LeaveCriticalSection () 都能够被执行到。否则临界区保护的共享资源将永远不会被释放。虽然临界区同步速度很快，但却只能用来同步本进程内的线程，而不可用来同步多个进程中的线程。

事件（Event）

事件对象也可以通过通知操作的方式来保持线程的同步。并且可以实现不同进程中的线程同步操作。

信号量包含的几个操作原语：CreateEvent () 创建一个信号量 OpenEvent () 打开一个事件 SetEvent () 回置事件 WaitForSingleObject () 等待一个事件 WaitForMultipleObjects () 等待多个事件

WaitForMultipleObjects 函数原型：WaitForMultipleObjects（IN DWORD nCount, // 等待句柄数 IN CONST HANDLE *lpHandles, //指向句柄数组 IN BOOL bWaitAll, //是否完全等待标志 IN DWORD dwMilliseconds //等待时间）

参数 nCount 指定了要等待的内核对象的数目，存放这些内核对象的数组由 lpHandles 来指向。fWaitAll 对指定的这 nCount 个内核对象的两种等待方式进行了指定，为 TRUE 时当所有对象都被通知时函数才会返回，为 FALSE 则只要其中任何一个得到通知就可以返回。dwMilliseconds 在这里的作用与在 WaitForSingleObject () 中的作用是完全一致的。如果等待超时，函数将返回 WAIT_TIMEOUT。

事件可以实现不同进程中的线程同步操作，并且可以方便的实现多个线程的优先比较等待操作，例如写多个 WaitForSingleObject 来代替 WaitForMultipleObjects 从而使编程更加灵活。

互斥量（Mutex）

互斥量跟临界区很相似，只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其他线程在获得后得以访问资源。互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

互斥量包含的几个操作原语：CreateMutex () 创建一个互斥量 OpenMutex () 打开一个互斥量 ReleaseMutex () 释放互斥量 WaitForMultipleObjects () 等待互斥量对象

信号量（Semaphores）

信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的 PV 操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同

一时刻访问此资源的最大线程数目。在用 CreateSemaphore () 创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减 1，只要当前可用资源计数是大于 0 的，就可以发出信号量信号。但是当前可用计数减小到 0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过 ReleaseSemaphore () 函数将当前可用资源计数加 1。在任何时候当前可用资源计数决不可能大于最大资源计数。

PV 操作及信号量的概念都是由荷兰科学家 E.W.Dijkstra 提出的。信号量 S 是一个整数，S 大于等于零时代表可供并发进程使用的资源实体数，但 S 小于零时则表示正在等待使用共享资源的进程数。

P 操作申请资源：（1）S 减 1；（2）若 S 减 1 后仍大于等于零，则进程继续执行；（3）若 S 减 1 后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。

V 操作释放资源：（1）S 加 1；（2）若相加结果大于零，则进程继续执行；（3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。

信号量包含的几个操作原语：CreateSemaphore()创建一个信号量 OpenSemaphore()打开一个信号量 ReleaseSemaphore () 释放信号量 WaitForSingleObject () 等待信号量

信号量的使用特点使其更适用于对 Socket（套接字）程序中线程的同步。例如，网络上的 HTTP 服务器要对同一时间内访问同一页面的用户数加以限制，这时可以为每一个用户对服务器的页面请求设置一个线程，而页面则是待保护的共享资源，通过使用信号量对线程的同步作用可以确保在任一时刻无论有多少用户对某一页面进行访问，只有不大于设定的最大用户数目的线程能够进行访问，而其他的访问企图则被挂起，只有在有用户退出对此页面的访问后才有可能进入。

综上所述：当在同一进程中的多线程同步时，临界区是效率最最高，基本不需要什么开销。而内核对象由于要进行用户态和内核态的切换，开销较大，但是内核对象由于可以命名，因此它们同时可以用于进程间的同步。另外，值得一提的是，信号量可以设置允许访问资源的线程或进程个数，而不仅仅是只允许单个线程或进程访问资源。

20.1 Questions

第一题：线程的基本概念、线程的基本状态及状态之间的关系?

第二题：线程与进程的区别?

1、线程是进程的一部分，所以线程有的时候被称为是轻权进程或者轻量级进程。

2、一个没有线程的进程是可以被看作单线程的，如果一个进程内拥有多个进程，进程的执行过程不是一条线（线程）的，而是多条线（线程）共同完成的。

3、系统在运行的时候会为每个进程分配不同的内存区域，但是不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源。那就是说，出了 CPU 之外（线程在运行的时候要占用 CPU 资源），计算机内部的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源。

4、与进程的控制表 PCB 相似，线程也有自己的控制表 TCB，但是 TCB 中所保存的线程状态比 PCB 表中少多了。

5、进程是系统所有资源分配时候的一个基本单位，拥有一个完整的虚拟空间地址，并不依赖线程而独立存在。

第三题：多线程有几种实现方法，都是什么?

1. 继承 Thread 类

2. 实现 Runnable 接口再 new Thread(YourRunnableOjbect)

第四题：多线程同步和互斥有几种实现方法，都是什么?

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

第五题：多线程同步和互斥有何异同，在什么情况下分别使用他们？举例说明。

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步（下文统称为同步）。

第三题（某培训机构的练习题）：

子线程循环 10 次，接着主线程循环 100 次，接着又回到子线程循环 10 次，接着再回到主线程又循环 100 次，如此循环 50 次，试写出代码。

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<pthread.h>
#include<unistd.h>
int num=1; //全局变量
static pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
void *func();
int main() {
    pthread_t tid;
    int ret=0,i,j;
    if((ret=pthread_create(&tid,NULL,func,NULL))!=0)
        printf("create thread error!\n");
        for(i=0;i<3;i++) {
            pthread_mutex_lock(&mutex);
            if(num!=0) pthread_cond_wait(&cond,&mutex);
            num=1;
            for(j=0;j<10;j++)
                printf("0");
            printf("\n");
            pthread_mutex_unlock(&mutex);
            pthread_cond_signal(&cond);
        }
        return 0;
}
void *func() {
    int i,j;
    for(i=0;i<3;i++) {
        pthread_mutex_lock(&mutex);
        if(num!=1) pthread_cond_wait(&cond,&mutex);
        num=0;
        for(j=0;j<5;j++)
            printf("1");
        printf("\n");
        pthread_mutex_unlock(&mutex);
        pthread_cond_signal(&cond);
    }
    pthread_exit(0);
}
```

第四题（迅雷笔试题）：

编写一个程序，开启 3 个线程，这 3 个线程的 ID 分别为 A、B、C，每个线程将自己的 ID 在屏幕上打印 10 遍，要求输出结果必须按 ABC 的顺序显示；如：ABCABC…. 依次递推。

```
#include<stdio.h>
#include<stdlib.h>
#include<error.h>
#include<unistd.h>
#include<pthread.h>
int num=0;
static pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
void *func(void *);
int main() {
    pthread_t tid[3];
    int ret=0,i;
    for(i=0;i<3;i++)
    if((ret=pthread_create(&tid[i],NULL,func,(void*)i))!=0)
        printf("create thread_\%c error\n",i+'A');
    for(i=0;i<3;i++)
        pthread_join(tid[i],NULL);
```

```
    printf("\n");
    return 0;
}
void *func(void *argc) {
    int i;
    for(i=0;i<10;i++) {
        pthread_mutex_lock(&mutex);
        while(num!=(int)argc)
            pthread_cond_wait(&cond,&mutex);
        printf("\%c",num+'A');
        num=(num+1)\%3;
        pthread_mutex_unlock(&mutex);
        pthread_cond_broadcast(&cond);
    } pthread_exit(0);
}
```

第五题（Google 面试题）

有四个线程 1、2、3、4。线程 1 的功能就是输出 1，线程 2 的功能就是输出 2，以此类推..... 现在有四个文件 ABCD。初始都为空。现要让四个文件呈如下格式：

A：1 2 3 4 1 2....

B：2 3 4 1 2 3....

C：3 4 1 2 3 4....

D：4 1 2 3 4 1....

请设计程序。

```
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
using namespace std;

pthread_mutex_t myloack=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mycond=PTHREAD_COND_INITIALIZER;
int n=0;
void *ThreadFunc(void *arg){
    int num=(int )arg;
    for (int i = 0; i < 10; ++i){
        pthread_mutex_lock(&myloack);
        while (n!=num)
            pthread_cond_wait(&mycond,&myloack);

        if (num==0)    cout<<"1";
        else if(num==1)    cout<<"2";
        else if(num==2)    cout<<"3";
        else cout<<"4"<<endl;
        n=(n+1)\%4;
        pthread_mutex_unlock(&myloack);
        pthread_cond_broadcast(&mycond);
    }
    return (void *)0;
}

int  main(int argc, char const *argv[]){
    pthread_t id[4];
    for (int i = 0; i < 4; ++i){
        int err=pthread_create(&id[i],NULL,ThreadFunc,(void *)i);
        if (err!=0){
            cout<<"create err:"<<endl;
            exit(-1);
        }
    }
    for (int i = 0; i < 4; ++i){
        int ret=pthread_join(id[i],NULL);
        if (ret!=0){
```

```
            cout<<"join err:"<<endl;
            exit(-1);
        }
    }
    return 0;
}
```

第六题生产者消费者问题

有一个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个有多个缓冲区的缓冲池，生产者将它生产的产品放入一个缓冲区中，消费者可以从缓冲区中取走产品进行消费，所有生产者和消费者都是异步方式运行的，但它们必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经装满产品且尚未被取走的缓冲区中投放产品。

```
#include<iostream>
#include<thread>
#include<mutex>
#include<condition_variable>
#include<array>
#include<boost/circular_buffer.hpp>
using namespace std;
mutex m;
condition_variable full,empty;
boost::circular_buffer<int> Q(10);// 缓冲区大小为 10, 缓冲区数据为 int, 这里充当 blocking queue
bool flag=true;//一个简陋的设计, 当不再生产时采用 flag 终止消费者线程
void put(int x){
    for(int i=0;i<x;i++){
        unique_lock<mutex> lk(m);
        while(Q.full())
        empty.wait(lk);
        assert(!Q.full());
        Q.push_back(i);
        cout<<"@ "<<i<<endl;//生产
        full.notify_all();
    }
    flag=false;
}
void take(){
    while(flag){
        unique_lock<mutex> lk(m);
        while(Q.empty())
        full.wait(lk);
        if(flag){
            assert(!Q.empty());
            cout<<"# "<<Q.front()<<endl;//消费
            Q.pop_front();
            empty.notify_all();
        }
    }
}
int main(){
    thread one(take);
    thread two(take);
    thread three(take);
    put(100);
    one.join();
    two.join();
    three.join();
    return 0;
}
```

第七题读者写者问题

有一个写者很多读者，多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，同样有读者读时写者也不能写。

```
#include<iostream>
#include<mutex>
#include<thread>
#include<memory>
#include<vector>
#include<assert.h>
using namespace std;
mutex m;
shared_ptr<vector<int>> ptr;
int loop=100;
void read(){//读者
    while(1){
        {//放在块内可以使临时对象得到及时析构
            shared_ptr<vector<int>> temp_ptr;
            {
                unique_lock<mutex> lk(m);//这里读者和读者之间，读者和写者之间都互斥，但是临界区很小所以不用担心
                temp_ptr=ptr;//这里会使对象的引用计数加 1
                assert(!temp_ptr.unique());
            }
            for(auto it=temp_ptr->begin();it!=temp_ptr->end();it++)//如果存在写者，那么读者访问的是旧的 vecto
            cout<<*it<<" ";
        }
    }
}
void write(){//写者
    for(int i=0;;i++){
        {//在一个块内使临时对象及时得到析构
            unique_lock<mutex> lk(m);//写者和写者之间，写者和读者之间都要互斥
            if(!ptr.unique())//如果存在其它写者或读者，则需要拷贝当前的 vector
            ptr.reset(new vector<int>(*ptr));
            assert(ptr.unique());
            ptr->push_back(i);
        }
    }
}
int main(){
    ptr.reset(new vector<int>);
    thread one(read);
    thread two(read);
    write();
    one.join();
    two.join();
    return 0;
}
```

第 21 章
Brain Teaser

21.0.1 Exponent Grows

An bacteria grows at the speed that it will double its volume per minute. If you put it in a jar, it will fill the jar in one hour. how long will it take to fill a half of the jar?

解题思路

$2^{60} = jar ==> \frac{2^{60}}{2} = \frac{jar}{2}$

21.0.2 Pizzas

If one and a half teenagers, eat one and a half pizzas in one and a half days, how many pizzas can 9 teenagers eat in 3 days?

解题思路

->1.5 teenagers eat 1.5 pizzas in 1.5 days
=>1.5 teenagers eat 3 pizzas in 3 days
=>3 teenagers eat 3 pizzas in 6 days
=>9 teenagers eat 3 pizzas in 18 days

21.0.3 Lights

There is a two storey home. We have three switch on the first floor and three rooms on the second floor. You can go upstairs only once and can touch switches only one. How will you find out which switch is for what room?

解题思路

1. turn on one switch for 5 minutes and turn off
2. turn on another switch
3. go upstair and the light bulb is for the turning switch, the hot bulb for first turned switch(turn on for 5 minutes) and the last one is for the third switch

21.0.4 Volum

Two container of 5L and 3L are given. Then are is 9.5L water given you need to make 4L water with minimum attempts and water wastage.

解题思路

21.0.5 Eggs

Given one egg and a building with infinite number of floors. Find out minimum number of throws at which (least) floor egg will break, if thrown?

I said we have to start at floor 1 and keep incrementing and testing by moving 1 floor up. Then he said optimize it by minimizing no of throws. I could not find more optimal way. I told him that I know with problem with 2 eggs and finite floor building.

Then, he told me that now lets there are 2 eggs and infinite floor building, find minimum no if throws required to find least floor at which egg breaks.

I still could not do that for infinite floors.

解题思路

lets calculate value of optimal x: x + (x-1) + (x-2) + (x-3) + ... 2 + 1 = no of floors x(x+1)/2 = no. of floors

21.0.6 Cross Bridge

4 men- each can cross a bridge in 1,3, 7, and 10 min. Only 2 people can walk the bridge at a time. How many min. minutes would they take to cross the bridge.

解题思路

Two pass schedule in parallel. 11 min for optimal solution.

21.0.7 Ball Identify

You are given a set of 8 balls, all of them identical in appearance and weight, except for one which is slightly heavier than the rest. You are also given a scale with no units, which can only tell you if one load is heavier than the other (think a scale-of-justice type scale). How can you find the heavy ball with only two comparisons? You may place as many balls as you wish on either side of the scale for each comparison.

解题思路

Balls: 1 2 3 4 5 6 7 8 One Weight: 1 2 3 vs 4 5 6

Case 1: If it matches then take remaining to balls say 7 and 8 and put on either side.

Now which ever is heavier is the one

Case 2: If first 6 does not match, then take the 3 balls from heavier side and pick any 2 of them and weigh it.

Case A: If not same then the heavier is one.

Case B: If same then the remaining 3rd ball is the heavier one

21.0.8 Alarm Clock Design

Design an alarm clock for a deaf person

解题思路

Vibrator, light & pillow vibrator. To cover minimal to severely deaf.

Vibrator + Light = Normal deaf.

Pillow vibrator + Light = severely deaf.

```
Class Clock
    Set the time.
    Advance the time.
    Display the time.

Class Hour indicator
    Set its value
    Advance its value
    Display its value
```

```
Class Minute indicator
    Set its value
    Advance its value
    Display its value

Class Seconds indicator
    Advance its value
    Display its value

Vibrator
    Trigger
    Silent

Light
    On
    Off
```

21.0.9 Thread Safe Queue

Implement a circular queue of integers of user-specified size using a simple array. Provide routines to initialize(), enqueue() and dequeue() the queue. Make it thread safe. Please implement this with your own code and do not use any class libraries or library calls.

解题思路

21.0.10 Clock Hand Angle

Find the angle between minute and hour hand when time is 6:50 am

解题思路

Angle = (h*30 + (m/60) * 30) - (m * 6)

21.0.11 Triangle Division

Divide a triangle into 5 triangles of equal area

解题思路

21.0.12 Water Level

There is a boat on the water. There is a rock in it. At this time, assume the level of the water is L1. Now say, you threw the rock into the water and it reached the bottom. Now what will be the level of the water?

解题思路

The level of the water should be lower compared to L1, the reasons are explained as following:

case (I), boat + rock are in an equlibrium state which means: floating force generated by water == gravitational force of the boat+rock, in physics language rho * Volume_water * g = (m_rock + m_boat) *g

case (II), boat is still in an equilibrium state (floating force== gravitational force) however the rock must go to the bottom of the water as it has much larger density (floating force « gravitational force) so rho * Volume_water * g « (m_rock+m_boat) *g

to conclude, Volume_water_1 > Volume_water_2 so, water_level_2 < water_level_1

21.0.13 Ropes Burning

There are unlimited ropes of same length and if we burn them it will take 1 hour. How to measure 45 mins?

解题思路

Take 2 ropes R1, R2. Light R1 at both ends and R2 at one end. When R1 is completely burned (30 minutes have passed), light R2 at the other end. When R2 is completely burned 45 minutes have passed.

21.0.14 Cross Bridge

There are four people at the end of bridge. They take following time to cross the bridge:

1st: 1/1 min

2nd: 2/2 min

3rd: 7/5 min

4th: 9/10 min

Condition:

1. They have only one torch and they can't cross bridge without torch.
2. and only 2 person can cross bridge at a time.

解题思路

-> 1. + 2. - 2 min

<- 1. - 1 min

-> 3. + 4. - 9/10 min

<- 2. - 2 min

-> 1. + 2. - 2 min

21.0.15 Comperision Number

Which is greated 3^{20} or 2^{30}

解题思路

$3^{20} = (3 * 3)^{10} = 9^{10}$

$2^{30} = (2 * 2 * 2)^{10} = 8^{10}$

第 22 章
Operating System

22.1 Questions

22.1.1 Operating Sequence

Sequence of steps that happen in CPU, cache, TLB, VM, HDD leading to execution of "x = 7"which isn't present in cache or sysmem nor translation in TLB. Also specify if any intrs, exceptions or faults are generated.

1) CPU first fetches the instruction x = 7 from the Instruction cache (when it reads this address in the PC/IR)
2) After decoding and executing the instruction, it sees that, it needs to access the memory location of variable x (which will be a virtual address)
3) Hence it issues a request to the TLB to return the physical address/tag.
Assuming the cache is Virtually indexed, it will parallely calculate the index for this virtual address.
4) Since it's a TLB miss, it accesses the Page table which resides mostly in Memory.
//Not sure what the interrupt here is?
5) But since, the translation is not found, meaning the page for this address is not in RAM, it issues a DMA request to transfer the page from Secondary storage to the RAM. It knows the address of the page on Secondary storage through the vm_area struct for this process, which maintains the location of all the pages.
// This is done in page fault handler. Page fault is raised when this even occurs.
6) Once DMA is complete, the processor is interrupted with this event. It then updates the page table with this entry and also the TLB.
//This would be an I/O interrupt to the processor 7) Once it gets the tag, it checks if that tag matches in the cache.
8) It won't, since cache does not have this entry.
9) Hence it fetches this block (cache block) from memory and places into the cache and restarts the execution.
10) In the MEM phase of the execution pipeline, it writes the value 7 to this location in the cache.

[Alternation]There are 2 cases viz. Swapping/Non-Swapping:

Case 1: No Swapping. Here we assume, we have physical pages available, hence no swapping needed. 1. CPU fetches the instruction x=7 (We assume, code page is in memory) and decodes it. 2. CPU tries to find corresponding physical address of "x" by looking at TLB. Since there is no TLB entry present, (first stop the instruction) MMU (Memory management unit) raises TLB exception, otherwise know as TLB Fault. 3. Now, TLB fault handling code, visits PTE (Page table entries) to find corresponding physical page. Since, there is no physical page assigned, this causes Page Fault to be raised, which allocates a page in RAM and updates PTE. 4. Restart the instruction. Note - We havent updated TLB entry yet. This would be done when the restarted instruction runs again, which would again raise TLB fault, but this time TLB fault handling code would find the corresponding physical page and would load the TLB entry and the instruction is restarted. 5. Thus finally, this time, there would be no TLB fault and Page fault and instruction would be successful.

Case 2 - Swapping. In this case, RAM is out of physical pages and only way to find a physical page is to swap out a used physical page to the HDD and then use that page (the one that we swapped out) to assign during Page fault.

Rest remains the same. Only, Page Fault handling changes i.e. it involves swapping (HDD is involved). Note - We also need to update PTE of the page we swapped out to the disk, so that the process (whose page we swapped out) knows about its page location.

22.1.2 Conseptions

What is a process.
What is a stack.
How many stacks can a process have.
What is a thread.
How many threads can a process have.
Do the threads have their own stacks, or share the process stack(s), or both?
What is deadlock or livelock?
How to prevent deadlock?

22.1.3 Compiler Optimization

Compiler optimization for memory based code. How do you make sure your code works fine with such a code?

Memory based code should have the ”volatile” keyword for compiler to not optimize that particular variable

22.1.4 Singleton Class

What is a singleton class? Explain the code?
How do you handle singleton object in multithreaded programs?

22.1.5 Mmap vs DMA

Explain the difference between Memory mapped I/O and DMA?

Memory mapped I/O allows the CPU to control hardware by reading and writing specific memory addresses. Usually this would be used for low-bandwidth operations such as changing control bits.

DMA allows hardware to directly read and write memory without involving the CPU. Usually this would be used for high-bandwidth operations such as disk I/O or camera video input.

22.1.6 Stack Grown up or down

Write a C function to return whether the stack grows up or grows down.

22.1.7 Context Switch

Describe what happens when a function is called from another function.

1. All local variables are reserved in stack
2. ES register such like return address is saved
3. Jump to the function address to fetch the code
4. When the function is created, the local variables in the function is created in stack area, too.
5. When it’s done, it goes back to the return address saved in ES.
6. Keep going on the caller function.

22.1.8 title

If an application is running, but it does not produce output; memory utilization is constant, cpu utilization goes down to 0; what will be the problem.

When you say CPU utilization goes down to zero, are you talk about CPU utilization for this process alone. What is the memory utilization for this process, is it constant, If yes, Then, It may be due to high priority process switching the context and
- May be in a deadlock.
- Waiting for I/O.
- Thrashing

22.1.9 Core Dump

what are core dumps? What kind of editor do you use to look at core dumps?

A core dump is the recorded state of the working memory of a computer program at a specific time, generally when the program has terminated abnormally (crashed).[1] In practice, other key pieces of program state are usually dumped at the same time, including the processor registers, which may include the program counter and stack pointer, memory management information, and other processor and operating system flags and information. The name comes from the once-standard memory technology core memory. Core dumps are often used to diagnose or debug errors in computer programs.

22.1.10 Memory Allocation

malloc vs calloc

new vs malloc

第 23 章
Networking & Distributed Computing

23.1 Distributed Computing

23.1.1 System Design

Facebook Friends

Design a system like friend's functionality in facebook. should have all features of facebook's friends functionality. like for each person , he can have any number of friends , he will get suggestions for new firends , showing common friends if we visits any other profile . algo should be scalable , robust.

Basically a friends network is an undirected graph. Each person is a vertex of the graph and friendship is an edge between the two vertices. We can represent the graph by by maintaining an adjacency list. - Like: Each person maintains a list of Posts. Lets say person A puts a post, persons who are connected to person A will be notified about the post (observer pattern). Friend B can like a post which increase the 'Like' count. - Friends Suggestion: For person A find all nodes X that is at distance 2 from node A. (Now this is scalable as the requirements can tell what is the depth we want to look for making suggestions.) - Common Friends: When A visits person B. Find all nodes X that are at distance 1 from node A. - Tagging Friends:

REST API

Design an architecture for REST APIs where you have to upload big data like images/videos etc. Request should be async. Follow up: How will you tune the performance if you have millions of requests coming at same time? Clues: Queueing the request, Storing data in filesystems rather than traditional DB etc.

SWMR

Design a distributed keystore with a single write end-point and multiple read end-points.

Phrase Search

How does a search engine perform exact phrase search? i.e. search for the term "the bees knees" exactly.

When a phrase search is done, there could be multiple ways based on how the search engine is implemented. Will explain one way here. Each of the words of the phrases are searched on the inverted index and the posting lists of the corresponding words will all be retrieved. We can do a repeated intersection of the sets of documents in the posting lists.. In the document set where all the words are present, a more detailed search for the specific location of the word in the documents where each of these words occur. The result will have those documents that have them in the consecutive positions in each of the documents in the specified order. Sometimes the phrases may be stored as they are too.. For instance to search phrases like "To be or not to be", which consists of all stopwords.. If the n gram frequencies in words or phrases are stored as in vector space models, phrase searches are simpler.

Caching Design

Design a collaborative text editor where each participant has infinite undo/redo. Consider the scenario where a user goes offline and then comes online and tries to undo/redo.

At a high level very naive approach: each participant generates a string of actions associated with a timestamp. It's a product design decision whether to use client side timestamps or server side timestamps for each action. From the server side, it's a matter of merging these action sorted by timestamp. Every participants series of actions are stored, so every time participant sends an undo or redo update that action with the latest timestamp (so that it's the latest action now). Action is defined as a character (including backspace and cursor move)

When user goes offline, the client side should gray out the document and make it read-only otherwise there's no logical way to merge his changes made in old document to the new document when he get online.

Parallel BFS and DFS

Design a parallel Breadth First Search algorithm for a directed weighted graph.

Basically you need to find the minimum cost to reach to a node from the starting node <given>. (Just save the optimum cost and not the optimum path). Calculate and output the optimum reachability cost for all the nodes from a given starting point.

Implement in C with openMP.

1. How about using DFS or Shortest path first instead. Would these algorithms perform better than BFS with parallel implementation. Yes/No Why?

While implementing BFS, we enque the neighbours of the present node into the queue. Then we deque each of them and again run a BFS on them sequentially. Because we have a parallel processing available, we can execute them in parallel instead of sequential. But we need to keep the common information like visited nodes etc with each of the process.

Scalability

How to add a counter to www.google.com to track the billionth user.

Assuming all web requests were always from unique users - worst case for computing! Operationally, the web server fleet needs these processes in it:

1. Accumulatator: Add number of web requests and send to a timebound computing server. Add a random delay to report this stat so that the computing server has time to process requests over a period of time. 2. Stat listener: Fetches the current stat data from a timebound computing server

The timebound computing server (lets say 600 servers exist, each counts the request data for 6 seconds) and are roundrobined in 1 hr - it's job is to use the time it has to sort the requests timestamp wise. It also fetches data from the previous server in roundrobin when it finishes counting, and adds the current fetched data to this list. If it sees a crossover of a billion, it can very well know which transaction caused it.

This can now be sent to the web server thru' the stat listener, and it could choose to update the unique user during his/her next request.

Alternately, it can be sent to the user's data in the distributed hash table.

Distributed Time Zone

While developing a globally distributed web application that needs to manage time records for system internals and for end-user usage (e.g. what date/time do i have the appointment?)
what best practices would you implement?
Hints - Server time versus user time. Distributed servers. Several timezones. Daylight savings time etc.

we can use NTP algorithm to deal with this situation.it uses a hierarchical, semi-layered system of levels of clock sources. This algorithm is a class of mutual network synchronization algorithm which allows for use-selectable policy control in the design of the time synchronization and evidence model. NTP supports single inline and meshed operating models in which a clearly defined master source of time is used ones in which no penultimate master or reference clocks are needed.

In NTP service topologies based on peering all clocks equally participate in the synchronization of the network by exchanging their timestamps using regular beacon packets. In addition NTP supports a UNICAST type time transfer which provides a higher level of security. NTP performance is tunable based on its application and environmental loading as well.

Heap vs Stack

What is the difference between a computers heap and it's stack?

Physically stack and heap both are allocated on RAM and their implementation varies from language, compiler and run time

Stack is used for local variables of functions and to track function calling sequences. Heap is used for allocating dynamically created variables using malloc, calloc or new. Stack memory is freed whenever the function completes execution but the heap memory needs to be freed explicitly using delete, free or by garbage collector of the language.

Stack memory of a process is fixed size and heap is variable memory.

Stack is faster than heap as allocating memory on stack is simpler just moving stack pointer up.

In case of multi threading, each thread of process will have a different stack but all threads share single heap

第 24 章
Database

Consider the following two tables, (a) CUSTOMERS and (b) ORDERS:

(a) CUSTOMERS

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

(b) ORDERS

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

24.1 SQL Operators

24.1.1 SQL Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

24.1.2 SQL Comparison Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

24.1.3 SQL Logical Operators

Here is a list of all the logical operators available in SQL.

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

24.2 Join

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them.

The most common type of join is: SQL INNER JOIN (simple join). An SQL INNER JOIN return all rows from multiple tables where the join condition is met.

let us join these two tables in our SELECT statement as follows:

```
SELECT ID, NAME, AGE, AMOUNT
FROM CUSTOMERS, ORDERS
WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

ID	NAME	AGE	DATE	AMOUNT
3	kaushik	23	2009-10-08 00:00:00	3000
3	kaushik	23	2009-10-08 00:00:00	1500
2	Khilan	25	2009-11-20 00:00:00	1560
4	Chaitali	25	2008-05-20 00:00:00	2060

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

SQL Join Types: There are different types of joins available in SQL:

INNER JOIN: returns rows when there is a match in both tables.

LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.

RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.

FULL JOIN: returns rows when there is a match in one of the tables.

SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

24.2.1 INNER JOIN

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax: The basic syntax of INNER JOIN is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

24.2.2 LEFT JOIN

The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax: The basic syntax of LEFT JOIN is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

24.2.3 RIGHT JOIN

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax: The basic syntax of RIGHT JOIN is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

24.2.4 FULL JOIN

The SQL FULL JOIN combines the results of both left and right outer joins.

The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Syntax: The basic syntax of FULL JOIN is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
FULL JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
```

```
|  7 | Muffy    |   NULL | NULL                |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

24.2.5 SELF JOIN

The SQL SELF JOIN is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

Syntax: The basic syntax of SELF JOIN is as follows:

```
SELECT  a.ID, b.NAME, a.SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;
```

This would produce the following result:

```
+----+----------+---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2 | Ramesh   | 1500.00 |
|  2 | kaushik  | 1500.00 |
|  1 | Chaitali | 2000.00 |
|  2 | Chaitali | 1500.00 |
|  3 | Chaitali | 2000.00 |
|  6 | Chaitali | 4500.00 |
|  1 | Hardik   | 2000.00 |
|  2 | Hardik   | 1500.00 |
|  3 | Hardik   | 2000.00 |
|  4 | Hardik   | 6500.00 |
|  6 | Hardik   | 4500.00 |
|  1 | Komal    | 2000.00 |
|  2 | Komal    | 1500.00 |
|  3 | Komal    | 2000.00 |
|  1 | Muffy    | 2000.00 |
|  2 | Muffy    | 1500.00 |
|  3 | Muffy    | 2000.00 |
|  4 | Muffy    | 6500.00 |
|  5 | Muffy    | 8500.00 |
|  6 | Muffy    | 4500.00 |
+----+----------+---------+
```

24.2.6 CARTESIAN JOIN

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from the two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to True or where the join-condition is absent from the statement.

Syntax: The basic syntax of INNER JOIN is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS, ORDERS;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
```

```
|  2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6 | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1560 | 2009-11-20 00:00:00 |
|  6 | Komal    |   2060 | 2008-05-20 00:00:00 |
|  7 | Muffy    |   3000 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1500 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1560 | 2009-11-20 00:00:00 |
|  7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

24.3 SELECT

24.3.1 ORDER BY

The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some database sorts query results in ascending order by default.

Syntax:

The basic syntax of ORDER BY clause is as follows:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

24.3.2 GROUP BY

The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

24.3.3 HAVING

The HAVING clause enables you to specify conditions that filter which group results appear in the final results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

24.3.4 DISTINCT

The SQL DISTINCT keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax: The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

24.3.5 TOP

The SQL TOP clause is used to fetch a TOP N number or X percent records from a table.

Note: All the databases do not support TOP clause. For example MySQL supports LIMIT clause to fetch limited number of records and Oracle uses ROWNUM to fetch limited number of records.

Syntax: The basic syntax of TOP clause with SELECT statement would be as follows:

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE [condition]
```

24.4 UNION

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax: The basic syntax of UNION is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
```

```
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

24.4.1 UNION ALL

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to UNION apply to the UNION ALL operator.

Syntax:

The basic syntax of UNION ALL is as follows:

```
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
SELECT  ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

There are two other clauses (i.e., operators), which are very similar to UNION clause:

SQL INTERSECT Clause: is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.

SQL EXCEPT Clause : combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

24.5 Interviewing Questions

24.5.1 Exclusive-Read with LOCK

Suppose that there is a database table, and four processes read the table at the same time. But, only one process is allowed to read the same row of the table at the same time. How do you enforce the exclusive-read on a row?

LOCKs:

24.5.2 TOP-K DELETE

How to delete two rows from the table in database ?

```
DELETE FROM table
WHERE id in (SELECT TOP 2 * FROM table);
```

24.5.3 ACID

Describe ACID properties in terms of a database transaction?

The ACID properties are atomic, consistent, isolated, and durable.

- **Atomic** means that a transaction cannot be subdivided. The transaction is all or nothing.

- **Consistent** means that a transaction satisfies integrity constraints after completion.

- **Isolated** means that transactions do not interfere with each other except in allowable ways. A transaction should never overwrite changes made by another transaction. In addition, a transaction may be restricted from interfering in other ways such as not viewing the uncommitted changes made by other transactions.

- **Durable** means that any changes resulting from a transaction are permanent. No failure will erase any changes after a transaction terminates.

Concurrency control and recovery management ensure that transactions meet the ACID properties. Recovery management involves actions to deal with failures such as communication errors and software crashes. Concurrency control involves actions to control interference among multiple, simultaneous users of the database.

24.5.4 Distribution of Database Servers

Database server is in US and web server is in India, how will you handle time zone conversion?

Store the time in US timezone on database side. Use views to display the time on the web server side. These views must calculate the indian time before displaying

24.5.5 Query Performance

How to improve query performance?

To improve the query performance we need to follow the below check list.

1. run the query and check the execution plan
2. in the execution plan if we are seeing table scan it means there is not index created. we need to create index on that table (the column which selected in select list)
3. if there is index is created but still ur getting table scan then need to check on which column the index has been created.. we need to make sure index has been created on the column which has int value .. coz the performance will hit if u have created index on Name column
4. from the execution plan we need to find how many hints we are getting based on that we need to modify the query.. hints are nothing but the suggestion for better performance.
5. there should not more index created on same table. else that will become problem.
6. what is the network speed .. that also matter for the performance issue.

Search Efficiently :

Depends on what job position you are looking for, you might want to answer it from a different aspect. But I think this question is asking you methods or different factors can effect the search.

1. Definitely ”Hash ” is a good answer.
2. (building) Indexes , in data warehouse, bitmap index may apply on some columns like gender or something (since you dont really do transaction in data warehouse, so bitmap works here, not in transaction tables)
3. Table Partitions
4. parallel operations (well, depends on if the DBA allows you to sacrifice some server performance to increase your efficiency)

24.5.6 Isolation

What are different isolation level?

Different isolation level are :

1. Read uncommitted (the lowest level where transactions are isolated only enough to ensure that physically corrupt data is not read)

2. Read committed (Database Engine default level)
3. Repeatable read
4. Serializable (the highest level, where transactions are completely isolated from one another)

24.5.7 Database Backup

Select the option that correctly describes the database replication concept where two or more replicas synchronize each other through a transaction identifier.

-Quorum
-Multimasterslave
-Master-Slave
-Multimaster

Master/Slave Replication: All write requests are performed on the master and then replicated to the slaves

Quorum: The result of Read and Write requests is calculated by querying a ”majority” of replicas.

Multimaster: Two or more replicas sync each other via a transaction identifier.

24.5.8 MapReduce vs Joins

In mapreduce, which type of join is more efficient, map side join or reduce side join and why?

'Reduce-Side joins are more simple than Map-Side joins since the input datasets need not to be structured. But it is less efficient as both datasets have to go through the MapReduce shuffle phase.'

24.5.9 JOIN vs UNION

Can you do JOIN using UNION keyword? Can you do UNION using JOIN keyword?

FULL OUTER JOIN is a UNION of LEFT JOIN and RIGHT JOIN

24.5.10 Distributed Queries

What is the best way to handle distributed databases and to merge their results ?

24.5.11 Specific Conceptions

Explain Indexing, Mining algorithms, Joins, SQL, normalisation, Materialized view

Indexing - database index can speed up a query by hundreds or thosands of times. This data structure(indexes) are created by using one or more columns in the table, providing a basis for both rapid random lookups and efficient access of ordered records. in a rational db an index is a copy of one part of a table. Code to create an index:

CREATE INDEX table_column1 ON table(column1) - this allows the database to very quickly match records from column1.

SQL - structured query language - a database language designed for managing data in a relational database management system RDBMS

Joins - a SQL clause that combines records from two or more tables in a database. An SQL join creates a set that can be saved as a table or used as is. different types of joins: INNER- create a set that has a similar set of column values ex. SELECT * FROM table a, table b INNER JOIN WHERE a.column1 = b.column2

OUTER - basically the opposite of an INNER JOIN, it will return all/some the records that do not match of the joined tables. outer hoins subdivide further into different types of joins LEFT, RIGHT, FULL

LEFT OUTER- aka left join - returns all the records on the left table and only the records on the right table that match. as for the sql below - all the records in table 1 and only the records that match for table 2

SELECT * FROM table1 a, table2 b LEFT JOIN ON column1 WHERE a.column1 = b.column2

RIGHT OUTER JOIN - aka RIGHT JOIN - is opposite of left join - return all the records from right table and only the records that match on left table

FULL OUTER JOIN - combines the records of both left and right joins and fill in NULLs for the missing records matches on either side. you need to use caution with FULL OUTER JOINs becuase not all database systems support this but they do support UNIONS so you will have to write a left and right join with a union.

Mining Algorithms: Apriori - an algorithm for association rules - aka association rule mining, given a set of itemsets (for instance, sets of retail transactions, each listing individual items purchased), the algorithm attempts to find subsets which are common to at least a minimum number C of the itemsets. Apriori uses a ”bottom up” approach, where frequent subsets are extended one item at a time (a step known as candidate generation), and groups of candidates are tested against the data. The algorithm terminates when no further successful extensions are found.

normalisation is the process of reducing the redundancy and replication in the database design.

Materialized view:1) MV is stored on disk 2) The reason MV is stored on disk is to reduce the computation time i.e. in data warehousing environments we use aggregated or summary tables, these summary/aggregate tables are created before hand in the form of MV and stored on disk. 3) Materialized view is a name in oracle environment and in sql server it is known by indexed views.

24.5.12 Stored Proc

Stored Proc vs Normal Query. Which is faster and why?

Stored Procedures are used to execute N number of times, update multiple tables at the same time and perform faster than a normal query since the communication from PL SQL Engine to SQL engine can be made faster by using collections. On the other hand, a normal query is just run once and update only one table at a time.

24.5.13 Indexing

1. Difference between clustered and non clustered index
2. Difference between having and where
3. What is query optimization

1. Clustred vs Non-Clustred A clustered index forces the data to be stored in the index column sequence. Clustered index is helpful to search range of data values. Eg: Assume there is an Employee(EmpId, EmpName, Sal, Dept) with clustered index on EmpId column SELECT Empname FROM Employee WHERE EmpId between 8 AND 16. As the employee records are stored in the sequence of EmpId, we can easily locate EmpIds 9 to 16 once we find the location of record for EmpID 8.

A good example of clustered index is: Dictionary where each page header contains the starting letters of words that are present in that page. If we go to a page, we can easily find all the words adjacent to a particular word. There can be only one clustered index per table but there can be approximately 249 non-clustred indexes.

Non-Clustred index: A non-clustered index stores location of the data page and an offset to the data record in that page. A good example is: index on the back of a book

Diff in Having and Where: Having is used with Group by clause and without group by clause it works similar to where. Where : It is conditional join between table and it filter the data from tables.

Query Optimization : The best way is to figure out the execution plan and then analysis how the data is fetched. If there are full table scans in large data tables then we need to consider the index. See if table joins are necessary otherwise we need to remove few tables if it is possible.

24.5.14 Big Database

You have to design a database that can store terabytes of data. It should support efficient point queries. How would you do it ?

If the data has some set of numbers that repeat very often, then we can have a hash table for it. for the remaining digits, we can have a BST. This will reduce the height of the BST and hence the value of log(n)

Patitioned Hashing Primary Indexing

[note]: hash trie array is the best in this case

24.5.15 Unique Key vs Primary Key

What's the difference between a primary key and a unique constraint?

Unique key columns may allow null values but the columns of a primary key do not allow null values.

Only one primary key column is allowed in a table But any number of Unique Constraint column is allowed in a table

第 25 章
OO & C++11

【Compiler】Compiling with c++11

g++ -std=c++11 your_file.cpp -o your_program

【OO】Three main features:

- Encapsulation (exposed by interface)
- Inheritance (implemented by Inheritance and Composition)
- Polymorphism (implemented by override and overload)

25.1 New Feature

The members and base classes of a struct are public by default, while in class, they default to private. Note: you should make your base classes explicitly public, private, or protected, rather than relying on the defaults.

【C++ FAQ】http://www.parashift.com/c++-faq

【__cplusplus 宏】

指针与引用的区别

指针与引用看上去完全不同 (指针用操作符“*”和“->”, 引用使用操作符“. ”), 但是它们似乎有相同的功能。指针与引用都是让你间接引用其他对象。

在任何情况下都不能使用指向空值的引用。引用肯定会指向一个对象, 在 C++ 里, 引用应被初始化。

指针与引用的另一个重要的不同是指针可以被重新赋值以指向另一个不同的对象。但是引用则总是指向在初始化时被指定的对象, 以后不能改变。

总的来说, 在以下情况下你应该使用指针, 一是你考虑到存在不指向任何对象的可能 (在这种情况下, 你能够设置指针为空), 二是你需要能够在不同的时刻指向不同的对象 (在这种情况下, 你能改变指针的指向)。如果总是指向一个对象并且一旦指向一个对象后就不会改变指向, 那么你应该使用引用。

还有一种情况, 就是当你重载某个操作符时, 你应该使用引用。最普通的例子是操作符 []。这个操作符典型的用法是返回一个目标对象, 其能被赋值。

当你知道你必须指向一个对象并且不想改变其指向时, 或者在重载操作符并为防止不必要的语义误解时, 你不应该使用指针。而在除此之外的其他情况下, 则应使用指针。

尽量使用 C++ 风格的类型转换

C++ 通过引进四个新的类型转换操作符克服了 C 风格类型转换的缺点, 这四个操作符是, static_cast, const_cast, dynamic_cast, 和 reinterpret_cast。在大多数情况下, 对于这些操作符你只需要知道原来你习惯于这样写,

(type) expression

而现在你总应该这样写:

static_cast<type>(expression)

static_cast 在功能上基本上与 C 风格的类型转换一样强大, 含义也一样。另外,static_cast 不能从表达式中去除 const 属性, 因为另一个新的类型转换操作符 const_cast 有这样的功能。

const_cast 用于类型转换掉表达式的 const 或 volatileness 属性。通过使用 const_cast, 你向人们和编译器强调你通过类型转换想做的只是改变一些东西的 constness 或者 volatileness 属性。这个含义被编译器所约束。如果你试图使用 const_cast 来完成修改 constness 或者 volatileness 属性之外的事情, 你的类型转换将被拒绝。

dynamic_cast, 它被用于安全地沿着类的继承关系向下进行类型转换。这就是说, 你能用 dynamic_cast 把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用, 而且你能知道转换是否成功。失败的转换将返回空指针 (当对指针进行类型转换时) 或者抛出异常 (当对引用进行类型转换时).dynamic_casts 在帮助你浏览继承层次上是有限制的。它不能被用于缺乏虚函数的类型, 也不能用它来转换掉 constness.

reinterpret_cast。使用这个操作符的类型转换, 其的转换结果几乎都是执行期定义 (implementation-defined)。因此, 使用 reinterpret_casts 的代码很难移植。reinterpret_casts 的最普通的用途就是在函数指针类型之间进行转换。

如果你使用的编译器缺乏对新的类型转换方式的支持, 你可以用传统的类型转换方法代替 static_cast, const_cast, 以及 reinterpret_cast。也可以用下面的宏替换来模拟新的类型转换语法: #define static_cast(TYPE,EXPR) ((TYPE)(EXPR)) #define const_cast(TYPE,EXPR) ((TYPE)(EXPR)) #define reinterpret_cast(TYPE,EXPR) ((TYPE)(EXPR))

你可以象这样使用使用:

double result = static_cast(double, firstNumber)/secondNumber;

不要对数组使用多态

类继承的最重要的特性是你可以通过基类指针或引用来操作派生类。这样的指针或引用具有行为的多态性, 就好像它们同时具有多种形态。C++ 允许你通过基类指针和引用来操作派生类数组。多态和指针算法不能混合在一起来用, 所以数组与多态也不能用在一起。

避免无用的缺省构造函数

Never call virtual fuctions in construction or destruction

运算符重载

谨慎定义类型转换函数

C++ 编译器能够在两种数据类型之间进行隐式转换 (implicit conversions), 它继承了 C 语言的转换方法, 例如允许把 char 隐式转换为 int 和从 short 隐式转换为 double。

有两种函数允许编译器进行这些的转换: 单参数构造函数 (single-argument constructors) 和隐式类型转换运算符。单参数构造函数是指只用一个参数即可以调用的构造函数。该函数可以是只定义了一个参数, 也可以是虽定义了多个参数但第一个参数以后的所有参数都有缺省值。

隐式类型转换运算符只是一个样子奇怪的成员函数:operator 关键字, 其后跟一个类型符号。你不用定义函数的返回类型, 因为返回类型就是这个函数的名字。

容易的方法是利用一个最新编译器的特性,explicit 关键字。为了解决隐式类型转换而特别引入的这个特性, 它的使用方法很好理解。构造函数用 explicit 声明, 如果这样做, 编译器会拒绝为了隐式类型转换而调用构造函数。

使用析构函数防止资源泄漏

c++ 中 inline, static, constructor 三种函数都不能带有 virtual 关键字。构造函数（默认，拷贝）都不能是虚函数。

inline 是编译时展开，必须有实体；static 属于 class 自己的，也必须有实体；virtual 函数基于 vtable（内存空间），constructor 函数如果是 virtual 的，调用时也需要根据 vtable 寻找，但是 constructor 是 virtual 的情况下是找不到的，因为 constructor 自己本身都不存在了，创建不到 class 的实例，没有实例，class 的成员（除了 public static/protected static for friend class/functions，其余无论是否 virtual）都不能被访问了。

然而，机构函数需要是虚拟函数。

使用析构函数防止资源泄漏

禁止异常信息 (exceptions) 传递到析构函数外

通过重载避免隐式类型转换

理解虚拟函数、多继承、虚基类和 RTTI 所需的代价

要求或禁止在堆中产生对象

【要求在堆中建立对象】

禁止以调用“new”以外的其它手段建立对象, 最直接的方法是把构造函数和析构函数声明为 private。这样做副作用太大。没有理由让这两个函数都是 private。最好让析构函数成为 private, 让构造函数成为 public。

另一种方法是把全部的构造函数都声明为 private。这种方法的缺点是一个类经常有许多构造函数, 类的作者必须记住把它们都声明为 private。否则如果这些函数就会由编译器生成, 构造函数包括拷贝构造函数, 也包括缺省构造函数; 编译器生成的函数总是 public。因此仅仅声明析构函数为 private 是很简单的, 因为每个类只有一个析构函数。

通过限制访问一个类的析构函数或它的构造函数来阻止建立非堆对象, 这种方法也禁止了继承和包容 (containment). 通过把析构函数声明为 protected(同时它的构造函数还保持 public) 就可以解决继承的问题, 需要包含对象的类可以修改为包含指向的指针.

【禁止堆对象】

通常对象的建立这样三种情况: 对象被直接实例化; 对象做为派生类的基类被实例化; 对象被嵌入到其它对象内。

禁止用户直接实例化对象很简单, 因为总是调用 new 来建立这种对象, 你能够禁止用户调用 new。自己声明 operator new/delete 函数, 而且你可以把它声明为 private（同理数组也是禁止 operator new[] 和 operator delete[]）。但是如此对象做为一个位于堆中的派生类对象的基类被实例化将变为不可能。

智能指针

共享资源的智能指针——shared_ptr: shared_ptr 是一种计数指针。当引用计数变为 0 时，shared_ptr 所指向的对象就会被删除。

unique_ptr 的用途。unique_ptr, 将一个指针从一个拥有者传个另一个，会以更为小的开销来更好的实现这个功能。

另外，不要不加思考地把指针替换为 shared_ptr 来防止内存泄露。shared_ptr 并不是万能的，而且使用它们的话也是需要一定的开销的：

- 环状的链式结构 shared_ptr 将会导致内存泄露（你需要一些逻辑上的复杂化来打破这个环。比如使用 weak_ptr）。
- 共享拥有权的对象一般比限定作用域的对象生存更久。从而将导致更高的平均资源使用时间。
- 在多线程环境中使用共享指针的代价非常大。这是因为你需要避免关于引用计数的数据竞争。
- 共享对象的析构器不会在预期的时间执行。
- 与非共享对象相比，在更新任何共享对象时，更容易犯算法或者逻辑上的错误。

unique_ptr 的使用能够包括：

- 为动态申请的内存提供异常安全
- 将动态申请内存的所有权传递给某个函数
- 从某个函数返回动态申请内存的所有权
- 在容器中保存指针

弱指针（weak pointer）经常被解释为用来打破使用 shared_ptr 管理的数据结构中循环 (?)。

如何在 C + + 代码中包含非系统的 C 头文件

如果你是其中一个 C 头文件不是由系统提供的，你可能需要把 #include 行放到 extern“C”(/ * ... * /）构造中。这告诉 C + + 编译器的功能在头文件中声明的 C 函数。

属性 Attributes

“属性”是 C11 标准中的新语法，用于让程序员在代码中提供额外信息。

```
void f [[ noreturn ]] () // f() 永不返回
{
    throw "error"; // 虽然不得返回，但可以抛出异常
}

struct foo* f [[carries_dependency]] (int i); // 编译优化指示
int* g(int* x, int* y [[carries_dependency]]);

// 使用 [[omp::parallel()]] 属性告诉编译器，这个 for 循环可以并行执行
for [[omp::parallel()]] (int i=0; i&lt;v.size(); ++i) { ... }
```

属性被放置在两个双重中括号“[[…]]”之间。目前，noreturn 和 carries_dependency 是 C++11 标准中仅有的两个通用属性。

auto – 从初始化中推断数据类型

常量表达式（constexpr）—一般化的受保证的常量表达式

常量表达式机制是为了：
提供一种更加通用的常量表达式
允许用户自定义的类型成为常量表达式
提供了一种保证在编译期完成初始化的方法（可以在编译时期执行某些函数调用）

decltype – 表达式的数据类型

decltype(E) 是一个标识符或者表达式的推断数据类型 (declared type)，可以用在变量声明中作为变量的数据类型。

控制默认函数——移动 (move) 或者拷贝 (copy)

在默认情况下，一个类有 5 个默认函数或操作符（译注：分为拷贝、移动和析构三大类）：

拷贝赋值操作符（copy assignment ）拷贝构造函数（copy constructor ）移动赋值操作符（move assignment ）移动构造函数（move constructor ）析构函数（destructor ）如果你显式声明了上述 5 个函数或操作符中的任何一个，你必须考虑其余的 4 个，并且显式地定义你需要的操作，或者使用这个操作的默认行为。拷贝、移动和析构是三个密切相关的操作，三者的行为必须完全协调一致。对于这五个函数，如果你只重定义了其中一小部分，而忽视了其余的函数，那等待你的只有灾难。

一旦我们显式地指明（声明, 定义, =default , 或者 =delete ）了上述五个函数之中的任意一个，编译器将不会默认自动生成 move 操作。一旦我们显式地指明（声明, 定义, =default , 或者 =delete ）了上述五个函数之中的任意一个，编译器将默认自动生成所有的拷贝操作。但是，我们应该尽量避免这种情况的发生，不要依赖于编译器的默认动作。

Lambda 表达式

Lambda 表达式是一种描述函数对象的机制，它的主要应用是描述某些具有简单行为的函数. 一个 Lambda 表达式可以存取在它被调用的作用域内的局部变量。

[&] “捕捉列表 (capture list)”，用于描述将要被 lambda 函数以引用传参方式使用的局部变量。
[&v] 仅“捕捉”参数 v
[=v] 以传值方式使用参数 v
[] 什么都不捕捉
[&] 将所有的变量以引用传递方式使用
[=] 以传值方式使用所有变量

noexcept 防止抛出异常

如果一个函数不能抛出异常或者一个程序没有对函数抛出的异常进行处理，那么这个函数可以用关键字 noexcept 进行修饰，例如：

extern ”C” double sqrt(double) noexcept;

nullptr——空指针标识

对重写（Override）的控制：final

对重写（Override）的控制：final

禁止继承类对基类的某些虚函数进行重写。在 C++11 中，可以通过使用“final”关键字来实现。

静态（编译期）断言—static_assert

静态（编译期）断言由一个常量表达式及一个字符串文本构成：

```
static_assert(expression, string);
```

expression 在编译期进行求值，当结果为 false（即：断言失败）时，将 string 作为错误消息输出。例如：

```
static_assert(sizeof(long) >= 8, “64-bit code generation required for this library.”);
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y),” unexpected padding in S”);
```

static_assert 在判断代码的编译环境方面（译注：比如判断当前编译环境是否 64 位）十分有用。但需要注意的是，由于 static_assert 在编译期进行求值，它不能对那些依赖于运行期计算的值的进行检验。例如：

```
int f(int* p, int n){
    //错误：表达式“p == 0”不是一个常量表达式
    static_assert(p == 0, “p is not null”);
}
```

统一初始化的语法和语义

C++11 的解决方法是对于所有的初始化，均可使用“{}-初始化变量列表”

async()

async 的工作是根据需要来启动新线程，而 future 的工作则是等待新线程运行结束。

不要使用 async() 来启动类似 I/O 操作，操作互斥体（mutex），多任务交互操作等复杂任务。

async() 可以启动一个新线程或者复用一个它认为合适的已有线程（非调用线程即可）

std::future 可用于异步任务中获取任务结果，但是它只是获取结果而已，真正的异步调用需要配合 std::async, std::promise, std::packaged_task。这里 async 是个模板函数，promise 和 packaged_task 是模板类，通常模板实例化参数是任务函数 (callable object)。下面是它们的部分组合用法：假设计算任务 int task(string x); 1 async+future 简单用法: future<int> myFuture=async(task,10)

async 开启后台线程执行任务

```
//自动选择线程执行任务 fn，args 是 fn 的参数，若 fn 是某个对象的非静态成员函数那么第一个 args 必须是对象的名字，
async (Fn&& fn, Args&&...args);

async (launch policy, Fn&& fn, Args&&... args);//有三种方式 policy 执行任务 fn
policy=launch::async 表示开启一个新的线程执行 fn
policy=launch::deferred 表示 fn 推迟到 future::wait/get 时才执行
policy=launch::async|launch::deferred 表示由库自动选择哪种机制执行 fn，和第一种构造方式 async(fn,args) 策略相同
```

std::future 和 std::promise

标准库中提供了 3 种 future：普通 future 和为复杂场合使用的 shared_future 和 atomic_future。在本主题中，只展示了普通 future，它已经完全够用了。如果我们有一个 future f，通过 get() 可以获得它的值:

X v = f.get(); // if necessary wait for the value to get computed

如果它的返回值还没有到达，调用线程会进行阻塞等待。

如果我们不需要等待返回值（非阻塞方式），可以简单询问一下 future，看返回值是否已经到达：if (f.wait_for(0)) ...

但是，future 最主要的目的还是提供一个简单的获取返回值的方法：get()。

promise 的主要目的是提供一个"put"（或"get"，随你）操作，以和 future 的 get() 对应。

promise 为 future 传递的结果类型有 2 种：传一个普通值或者抛出一个异常

lock 锁

锁是这样一个对象，它能够保持对一个 mutex 对象的引用并且可能会在自身销毁时（比如离开一个 block 域时）来对 mutex 对象进行解锁 unlock 操作。作为一种良好的异常处理习惯，可以在线程中使用锁来帮助管理 mutex 的拥有权。

可以使用 unique_lock 来很直接明了地给出关于锁的描述。unique_lock 能够更加安全的执行任何 mutex 所能达到的功能。

与使用 mutex 相比，使用锁可以提供异常处理并且能够在忘记 unlock() 时提供相应的保护。在并发编程中，我们可以通过锁来得到所有需要的功能。

现在考虑一个新的问题：如果我们需要两个分别用一个 mutex 表示的资源，应该如何实现？一种最简单的办法就是如下例所示的那样依次获取这两个 mutex：

```
std::mutex m1;
std::mutex m2;
int sh1; // 共享数据
int sh2
// ...
void f(){
    // ...
    std::unique_lock lck1(m1);
    std::unique_lock lck2(m2);
    //操作共享数据
    sh1+=sh2;
}
```

然而，上例中的方法有着一个致命缺陷：如果其它线程试图以相反的次序来获取 m1 和 m2，便会出现死锁现象。对于一个含有很多锁的系统来说，这一做法相当有风险。为了解决这类问题，锁提供了两个方法来安全地尝试获取两个或者两个以上的锁。下面是一个相应的例子：

```
void f(){
    // ...
    std::unique_lock lck1(m1,std::defer_lock); //使用锁但并不试图获取 mutex
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    lock(lck1,lck2,lck3);
    // 操作共享数据
}
```

很明显，必须非常并且精巧地设计上例中的 lock() 才能避免死锁。从本质上来讲，它和小心使用 try_lock()s 所达到的效果是一样的。当 lock() 没有成功获取所有所锁时，它会抛出一个异常。实际上，由于 lock() 和 try_lock(), unlock() 接受任何参数，所以我们无法分清 lock() 到底抛出了哪个异常。这依赖于它的参数。

如果你倾向于自己使用 try_lock() 来实现上例所示的相应功能，下面的例子可能对你有一定的帮助：

```
void f(){
    // ...
    std::unique_lock lck1(m1,std::defer_lock); // 使用锁但是并不试图去获取 mutex
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    int x;
    if ((x = try_lock(lck1,lck2,lck3))==-1) { // 欢迎来到 C 的地界
```

```
        // 操作共享数据
    }
    else {
        // x 拥有正在拥有一个 mutex，因此我们无法获取
        // 比如，如果 lck2.try_lock() 失效了，x 的值就等于 1
    }
}
```

mutex 互斥

互斥是多线程系统中用于控制访问的一个原对象（primitive object）。下面的例子给出了它最基本的用法：

```
std::mutex  m;
int sh; //共享数据
// …
m.lock();
// 对共享数据进行操作：
sh += 1;
m.unlock();
```

在任何时刻，最多只能有一个线程执行到 lock() 和 unlock() 之间的区域（通常称为临界区). 除了 lock()，mutex 还提供了 try_lock() 操作。线程可以借助该操作来尝试进入临界区，这样一来该线程不会在失败的情况下被阻塞。

recursive_mutex 是一种能够被同一线程连续锁定多次的 mutex。

Volatile 易变的

不可优化性:volatile 告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

易变性：在汇编层面反映出来，每次取值不是直接取 volatile 变量的寄存器内容，而是重新从内存中读取。

顺序性：多线程编程，并发访问/修改的全局变量，通常都会建议加上 Volatile 关键词修饰，来防止 C/C++ 编译器进行不必要的优化。

C/C++ Volatile 变量，与非 Volatile 变量之间的操作，是可能被编译器交换顺序的。

C/C++ Volatile 变量间的操作，是不会被编译器交换顺序的。

第 26 章
Appendex

26.1 註記

26.1.1 dungeon-game

動態規劃，對於每個格子記錄從它出發順利到達終點需要的最小血量。從 (m-1,n-1) 遞推至 (0,0)。注意對於每個格子其實有進和出兩個狀態：進入該格子時 (未考慮當前格子影響)；和離開該格子後 (已考慮當前格子影響)。這兩種方法都行，考慮到程序的結構可能有三部分：初始化邊界條件、狀態轉移、根據狀態表示答案，需要仔細斟酌以追求代碼簡短。這裏兩種方式並無明顯代碼長度差異。

26.1.2 majority-element

Boyer-Moore majority vote algorithm http://www.cs.utexas.edu/ moore/best-ideas/mjrty/ 這個可以拓展爲找出所有出現次數大於 N/K 的元素。方法是每次找出 K 個互不相同的元素丟掉，最後剩下的元素是候選解。時間複雜度 O(N*K)。

26.1.3 fraction-to-recurring-decimal

注意 INT_MIN/(-1) 會溢出。maximum-gap 平均数原理，求出极差 d 后，根据桶大小 ceil(d/(n-1)) 分成若干个桶，答案必为不同桶的两个元素之差。find-peak-element 二分查找，把區間縮小爲仍包含候選值的子區間。

```
int l = 0, h = a.size();
while (l < h-1) {
    int m = l+h >> 1;
    if (a[m-1] > a[m]) h = m;
    else if (m+1 == h || a[m] > a[m+1]) l = h = m;
    else l = m+1;
}
return l;
```

26.1.4 min-stack

使用兩個棧，原棧 S 存放各個元素。每當新元素小於等於當前最小值時，就把新元素複製一份壓入另一個棧 S'。彈出時，若元素在 S' 中出現，則也從 S' 中彈出。另一種方法是記錄新元素與當前最小值的差值，每個元素需要多記錄 1 個 bit。可惜 C++ 會 Memory Limit Exceeded，感覺不合理。

26.1.5 find-minimum-in-rotated-sorted-array

方法是二分，如果要寫短可以採用下面的思路：子數組中如果有相鄰兩個元素 a[i]>a[i+1]，則 a[i+1] 是整個數組的最小值。若子數組 a[i..j] 滿足 a[i]>a[j]，則存在 i<=k<j 使得 a[k]>a[k+1]。對於子數組 a[l..h] 判斷 a[l]>a[m] 或 a[m]>a[h] 是否有一個成立，成立則可以去除一半候選元素，不然 a[h]>a[l] 爲一個逆序對。

26.1.6 find-minimum-in-rotated-sorted-array-ii

對於子數組 a[l..h] 判斷 a[l]>a[m] 或 a[m]>a[h] 是否有一個成立，成立則可以去除一半候選元素。兩個條件均不滿足則 a[l]<=a[m]<=a[h]，若 a[h]>a[l] 則找到逆序對，否則 a[l]=a[m]=a[h]，可以把範圍縮小爲 a[l+1..h-1]

26.1.7 first-missing-positive

空間複雜度 O(1)。

26.1.8 jump-game

空間複雜度 O(1)。

26.1.9 jump-game-ii

使用 output-restricted queue 優化的動態規劃，注意到動態規劃值的單增性可以用類似 BFS 的方式，空間複雜度 O(1)。

26.1.10 linked-list-cycle-ii

Brent's cycle detection algorithm

26.1.11 longest-palindromic-substring

Manacher's algorithm

26.1.12 maximum-subarray

Kadane's algorithm

26.1.13 merge-k-sorted-lists

Tournament sort

26.1.14 minimal-window-substring

尺取法

26.1.15 n-queen

bitmask 存儲列，正反斜線控制的格子。

26.1.16 palindrome-partitioning

f[i][j] = calc(f[ii][jj] : i <= ii <= jj <= j) 形式的動態規劃可以採用如下計算方式。

```
for (int i = n; --i >= 0; )
for (int j = i; j < n; j++) {
    // calc [i,j]
}
```

26.1.17 recover-binary-search-tree

使用 Morris in-order traversal 找到鄰接失序對。如果交換的元素相鄰，則有一個鄰接失序對 (如 0 1 2 3 4 ->0 1 3 2 4)，否則有兩個鄰接失序對 (如 0 1 2 3 4 5 -> 0 4 2 3 1 5)。

26.1.18 remove-nth-node-from-end-of-list

http://meta.slashdot.org/story/12/10/11/0030249/linus-torvalds-answers-your-questions 使用 pointers-to-pointers 很多時候能簡化實現。

26.1.19 regular-expression-matching

P 爲模式串，T 爲文本，f[i][j] 表示 P 的前 i 個字符能否匹配 T 的前 j 個字符。根據 f[i-1][*] 計算 f[i][*]。這個方法也可以看作構建了精簡表示的 Thompson's automaton，時間複雜度 O(|P|*|T|)。

26.1.20 sort-colors

Dutch national flag problem 如果不要求 000111222，允許 111000222111，那麼有交換次數更少的 Bentley-McIlroy 算法 http://www.iis.sinica.edu.tw/ scm/ncs/2010/10/dutch-national-flag-problem-3/

26.1.21 sqrtx

Hacker's Delight (2nd) 11.1.1 46340 = floor(sqrt(INT_MAX))

26.1.22 scramble-string

我用了 $O(n^3)$ 的空間和 $O(n^4)$ 的時間，應該有更好的算法。

26.1.23 sort-list

Natural merge sort

26.1.24 sudoku-solver

轉化爲 exact cover problem，使用 dancing links + Algorithm X 求解。

26.1.25 trapping-rain-water

兩個指針夾逼，空間 O(1)。

26.1.26 unique-binary-search-trees-ii

子樹可以復用。

26.1.27 maximal-rectangle

秋葉拓哉（iwi）、巖田陽一（wata）和北川宜稔（kita_masa）所著，巫澤俊（watashi）、莊俊元（navi）和李津羽（itsuhane）翻譯的《挑戰程序設計競賽》逐行掃描棋盤，掃描第 i 行時，h[j] 表示第 j 列上方連續爲 1 的行數。若 a[i][j]==0 則 h[j]==0，當 a[i][j]==1 時，l[j] 表示 min(k : h[k+1..j]>=h[j])，r[j] 表示 max(k : h[j..k-1]>=h[j])。

```
#define ROF(i, a, b) for (int i = (b); --i >= (a); )
#define FOR(i, a, b) for (int i = (a); i < (b); i++)
#define REP(i, n) for (int i = 0; i < (n); i++)
class Solution {
    public:
    int maximalRectangle(vector<vector<char> > &a) {
        if (a.empty()) return 0;
        int m = a.size(), n = a[0].size(), ans = 0;
        vector<int> h(n), l(n), r(n, n-1);
        REP(i, m) {
            int ll = -1;
            REP(j, n) {
                h[j] = a[i][j] == '1' ? h[j]+1 : 0;
                if (a[i][j] == '0') ll = j;
                l[j] = h[j] ? max(h[j] == 1 ? 0 : l[j], ll+1) : j;
            }
            int rr = n;
            ROF(j, 0, n) {
```

```
                if (a[i][j] == '0') rr = j;
                r[j] = h[j] ? min(h[j] == 1 ? n-1 : r[j], rr-1) : j;
                ans = max(ans, (r[j]-l[j]+1)*h[j]);
            }
        }
        return ans;
    }
};
```

26.1.28 4-sum

使用了 multimap，時間複雜度 $O(n^2 * log(n))$，得到所有答案後不需要去重操作。合法的四元組有三類：a<=b<c<=d，枚舉 c 和 d，判斷是否存在和爲 target-c-d 的二元組 a<=b=c<=d，枚舉 b 和 d，判斷是否存在 target-b-b-d a=b=c<=d，枚舉 a，判斷是否存在 target-a-a-a 分別統計，小心實現可以保證不會產生相同的候選解，從而無需去重。

```
#define ROF(i, a, b) for (int i = (b); --i >= (a); )
#define FOR(i, a, b) for (int i = (a); i < (b); i++)
#define REP(i, n) for (int i = 0; i < (n); i++)
class Solution {
    public:
    int maximalRectangle(vector<vector<char> > &a) {
        if (a.empty()) return 0;
        int m = a.size(), n = a[0].size(), ans = 0;
        vector<int> h(n), l(n), r(n, n-1);
        REP(i, m) {
            REP(j, n) {
                h[j] = a[i][j] == '1' ? h[j]+1 : 0;
                l[j] = j;
                while (l[j] && h[l[j]-1] >= h[j])
                l[j] = l[l[j]-1];
            }
            ROF(j, 0, n) {
                r[j] = j;
                while (r[j]+1 < n && h[j] <= h[r[j]+1])
                r[j] = r[r[j]+1];
                ans = max(ans, (r[j]-l[j]+1)*h[j]);
            }
        }
        return ans;
    }
};
```

26.1.29 ACRush 某 TopCoder SRM

計數排序 h[*]，從小到大插入各個指標大意是這一行有存在若干個區間，對於一個區間 [L,R]，需要保證 peer[L] == R && peer[R] == L 插入指標時獲取所在區間的左右端點並更新 peer[*]

```
#define ROF(i, a, b) for (int i = (b); --i >= (a); )
#define FOR(i, a, b) for (int i = (a); i < (b); i++)
#define REP(i, n) for (int i = 0; i < (n); i++)
#define REP1(i, n) for (int i = 1; i <= (n); i++)
class Solution {
    public:
    int maximalRectangle(vector<vector<char> > &a) {
        if (a.empty()) return 0;
        int m = a.size(), n = a[0].size(), ans = 0;
        vector<int> h(n), p(n), b(m+1), s(n);
        REP(i, m) {
            REP(j, n)
            h[j] = a[i][j] == '1' ? h[j]+1 : 0;
            fill(b.begin(), b.end(), 0);
            REP(j, n)
            b[h[j]]++;
```

```
            REP1(j, m)
            b[j] += b[j-1];
            REP(j, n)
            s[--b[h[j]]] = j;
            fill(p.begin(), p.end(), -1);
            ROF(j, 0, n) {
                int x = s[j], l = x, r = x;
                p[x] = x;
                if (x && p[x-1] != -1) {
                    l = p[x-1];
                    p[l] = x;
                    p[x] = l;
                }
                if (x+1 < n && p[x+1] != -1) {
                    l = p[x];
                    r = p[x+1];
                    p[l] = r;
                    p[r] = l;
                }
                ans = max(ans, (r-l+1)*h[x]);
            }
        }
        return ans;
    }
};
```

26.1.30 棧維護 histogram

常見解法。

第 27 章
ACM 基础知识

27.1 方程组

$$AX = B, A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}, X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

高斯消元法：

克莱姆法则：$AX = B$,$x_i = \frac{|A_i|}{|A|}$

27.2 多项式

$f(x) = \sum_{i=0}^{n} a_i x^i$

【性质】1. $(1+x)^n = \sum_{i=0}^{n} C_n^i x^i$

2. n 个点可以唯一确定一个 n 次多项式；

3. 一个 n 次多项式有 n 个复数根

【算法】

用牛顿迭代法求解方程的根：

选择一个靠近多项式 $f(x)$ 零点的点 x_0 作为迭代初值，计算 $f(x_0)$ 和 $f^{'}(x_0)$, 解方程：

$$x \cdot f^{'}(x_0) + f(x_0) - x_0 \cdot f^{'}(x_0) = 0$$

解得 x_1, 如此反复迭代，可求出多项式的一个根。

27.3 快速傅里叶变换 FFT

«««< HEAD 快速傅里叶变换主要用于求两个多项式的乘积，即给定两个阶小于 n 的多项式 $f_A(x) = \sum_{i=0}^{n-1} a_i x^i, f_B(x) = \sum_{i=0}^{n-1} b_i x^i$，求解 $f_c(x) = \sum_{i=0}^{2n-2} c_i x^i = f_a(x) f_b(x)$. FFT 求解可以 $O(n \log(n))$. ======= 快速傅里叶变换主要用于求两个多项式的乘积，即给定两个阶小于 n 的多项式 $f_A(x) = \sum_{i=0}^{n-1} a_i x^i, f_B(x) = \sum_{i=0}^{n-1} b_i x^i$，求解 $f_c(x) = \sum_{i=0}^{2n-2} c_i x^i = f_a(x) f_b(x)$. FFT 求解可以 $O(n \log(n))$. »»»> 75843333e311d9a510db36a4c738da415fec2541

```
VOID WINAPI FFT(complex<double> * TD, complex<double> * FD, int r)  {
    LONG count;  // 付立叶变换点数
    int i,j,k;  // 循环变量
    int bfsize,p;  // 中间变量
    double angle;// 角度
    complex<double> *W,*X1,*X2,*X;
    count = 1 << r;  // 计算付立叶变换点数
    // 分配运算所需存储器
    W = new complex<double>[count / 2];
    X1 = new complex<double>[count];
    X2 = new complex<double>[count];
    // 计算加权系数
    for(i = 0; i < count / 2; i++)  {
        angle = -i * PI * 2 / count;
        W[i] = complex<double> (cos(angle), sin(angle));
```

```
        }
        // 将时域点写入 X1
        memcpy(X1, TD, sizeof(complex<double>) * count);
        for(k = 0; k < r; k++) {  // 采用蝶形算法进行快速付立叶变换
            for(j = 0; j < 1 << k; j++)  {
                bfsize = 1 << (r-k);
                for(i = 0; i < bfsize / 2; i++)  {
                    p = j * bfsize;
                    X2[i + p] = X1[i + p] + X1[i + p + bfsize / 2];
                    X2[i + p + bfsize / 2] = (X1[i + p] - X1[i + p + bfsize / 2]) * W[i * (1<<k)];
                }
            }
            X = X1;
            X1 = X2;
            X2 = X;
        }

        // 重新排序
        for(j = 0; j < count; j++)  {
            p = 0;
            for(i = 0; i < r; i++)  {
                if (j&(1<<i))
                    p+=1<<(r-i-1);
            }
            FD[j]=X1[p];
        }
        // 释放内存
        delete W;
        delete X1;
        delete X2;
    }
```

27.4 随机化

随机化是一类不确定但能以极大概率正确的算法。如拉斯维加斯算法和蒙特卡洛算法。

蒙特卡洛算法：

蒙特卡罗法（Monte Carlo method）是以概率和统计的理论、方法为基础的一种计算方法，将所求解的问题同一定的概率模型相联系，用电子计算机实现统计模拟或抽样，以获得问题的近似解，故又称统计模拟法或统计试验法。

蒙特卡罗算法在一般情况下可以保证对问题的所有实例都以高概率给出正确解，但是通常无法判定一个具体解是否正确。

«««< HEAD 设 p 是一个实数，且 1/2 <p <1。如果一个蒙特卡罗算法对于问题的任一实例得到正确解的概率不小于 p，则称该蒙特卡罗算法是 p 正确的，且称 p - 1/2 是该算法的优势。如果对于同一实例，蒙特卡罗算法不会给出 2 个不同的正确解答，则称该蒙特卡罗算法是一致的。======= 设 p 是一个实数，且 1/2 <p <1。如果一个蒙特卡罗算法对于问题的任一实例得到正确解的概率不小于 p，则称该蒙特卡罗算法是 p 正确的，且称 p - 1/2 是该算法的优势。如果对于同一实例，蒙特卡罗算法不会给出 2 个不同的正确解答，则称该蒙特卡罗算法是一致的。»»»> 75843333e311d9a510db36a4c738da415fec2541

拉斯维加斯算法：

拉斯维加斯算法的一个显著特征是它所作的随机性决策有可能导致算法找不到所需的解。因此通常用一个 bool 型函数表示拉斯维加斯算法。

【验证矩阵乘法的正确性】【验证两个函数是否等价】