Cracking the Coding Interview C++11

張世明 (simon.zhangsm@gmail.com)

https://github.com/simonzhangsm/crackcoding

2015-2-7

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 $\epsilon = 1e-9, a == b \equiv fabs(a-b) \le \epsilon$。

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

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

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。

【Algorithm】

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

相关题目

- Remove Duplicates from Sorted Array II，见 §2.1.2

2.1.2 Remove Duplicates from Sorted Array II

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。

【Sequent Scan】

```
// LeetCode, Remove Duplicates from Sorted Array II
// 时间复杂度 O(n)，空间复杂度 O(1)
#define OCCUR 2
class Solution {
    int removeDuplicates(int A[], int n) {
        if (n <= OCCUR) return n;
        int index = OCCUR;
        for (int i = OCCUR; i < n; i++){
            if (A[i] != A[index - OCCUR])
                A[index++] = A[i];
        }
        return index;
    }
};
```

【Algorithm2】

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
// 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 Remove Duplicates in Array

You're given an array of integers(eg [3,4,7,1,2,9,8]) Find the index of values that satisfy A+B = C+D, where A,B,C and D are integers values in the array. Eg: Given [3,4,7,1,2,9,8] array The following 3+7 = 1+ 9 satisfies A+B=C+D so print (0,2,3,5)

2.1.4 Bar Raiser Round

Divide the array(+ve and -ve numbers) into two parts such that the average of both the parts is equal.

Input: [1 7 15 29 11 9]

Output: [15 9 1 7 11 29]

Explanation: The average of first two elements is (15+9)/2 = 12, average of remaining elements is (1+7 +11 +29)/4 = 12

2.1.5 Search in Rotated Sorted Array

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。

【二分查找】

```
// 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 - 1;
                else
                    first = mid + 1;
            } else {
                if (A[mid] < target && target <= A[last-1])
                    first = mid + 1;
                else
                    last = mid - 1;
            }
        }
        return -1;
    }
};
```

相关题目

- Search in Rotated Sorted Array II，见 §2.1.6

2.1.6 Search in Rotated Sorted Array II

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) {
        if(n<1) return false;
        int first = 0, last = n - 1;
        while (first <= last) {
            const int mid = first  + (last - first) / 2;
            if (A[mid] == target)    return true;
            if (A[first] < A[mid]) {
                if(target == A[begin]) return true;
                if (A[first] < target && target < A[mid])
                    last = mid - 1;
                else
                    first = mid + 1;
            } else if (A[first] > A[mid]) {
                if(target == A[end]) return true;
                if (A[mid] < target && target < A[last-1])
                    first = mid + 1;
                else
                    last = mid - 1;
            } else //skip duplicate one
                first++;
        }
        return false;
    }
};
```

相关题目

- Search in Rotated Sorted Array，见 §2.1.5

2.1.7 Same Gap from Three Arrays

Given three arrays A,B,C containing unsorted numbers. Find three numbers a, b, c from each of array A, B, C such that |a-b|, |b-c| and |c-a| are minimum Please provide as efficient code as you can. Can you better than this?

2.1.8 MinIntNotSum

Given a array of positive integers, you have to find the smallest positive integer that can not be formed from the sum of numbers from array.

2.1.9 Consecutive Sub-numbers

Given a sorted array with some sequenced numbers and some non-sequenced numbers. Write an algorithm that takes this array as an input and returns a list of {start, end} of all consecutive numbers. Consecutive numbers have difference of 1 only. E.g. of array: [4, 5, 6, 7, 8, 9, 12, 15, 16, 17, 18, 20, 22, 23, 24, 27]

2.1.10 Count 0s in Array

Given a sorted array of 0’s and 1’s. Find out the no. of 0’s in it. Write recursive, iterative versions of the code.

2.1.11 Median of Two Sorted Arrays

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】

本题通用表述：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。

【解法一】$O(m+n)$ 的解法比较直观，直接 merge 两个数组，然后求第 k 大的元素。

【解法二】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。

【解法三】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。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]

【解法三】

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

2.1.12 Longest Consecutive Sequence

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。

【Hash Cache】

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

【聚类】

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。

```
    // Leet Code, Longest Consecutive Sequence
    // 时间复杂度 O(n)，空间复杂度 O(n)
    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;
        }
    };
```

【HashMap1】

```
 class Solution {
    int longestConsecutive(vector<int> &num) {
        unordered_map<int,int> s;
        int longest=1;
        for(int &i : num){
            if(s.find(i) == s.end()){
             s[i] = 1;
             int left = s.find(i-1) != s.end()? (i-s[i-1]) : i;
             int right = s.find(i+1) != s.end()? (i+s[i+1]) : i;
             s[left] = s[right] = right-left+1;
             longest = s[right]>longest? s[right] : longest;
            }
        }
        return longest;
    }
```

【HashMap2】The idea is to build a map on the way. The key is the number, the value means the length of the sequence which has the key as one of its borders. we iterate each value in the num array, if it resides in the map, just ignore. Else, we find the two adjcent sequences from left and right, and combine them into one seq. after that, we update the value of the borders of the new seq to the length of the seq.

```
    class Solution {
        int longestConsecutive(vector<int> &num) {
            int res = 0;
            unordered_map<int, int> h;
            for (auto& x: num) {
                if (!h[x]) {
```

```
                h[x] = 1 + h[x+1] + h[x-1];
                if (h[x+1]) h[h[x+1]+x] = h[x];
                if (h[x-1]) h[x-h[x-1]] = h[x];
            }
            res = max(h[x], res);
        }
        return res;
    }
};
```

2.1.13 kSum

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.

```
//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 §2.1.14
- 3Sum, 见 §2.1.16
- 3Sum Closest, 见 §2.1.17
- 4Sum, 见 §2.1.18

2.1.14 2Sum

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，注意跳过重复的数，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。

【方法二】

```
//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 §2.1.13
- 3Sum, 见 §2.1.16
- 3Sum Closest, 见 §2.1.17
- 4Sum, 见 §2.1.18

2.1.15 2Sum on BSTree

Given a binary search tree of n nodes, find two nodes whose sum is equal to a given number k in O(n) time and constant space.

【Algorithm】

```
//Two Sum in BST
// 时间复杂度 O(n)，空间复杂度 O(n)
class Solution {
    vector<Node*> twoNodeSum(Node *root, int k) {
        if(root == nullptr) return false;
        vector<Node*> res;
        if( 2*root->value > sum)
            twoNodeSum(root->left, root, sum, res);
        else if(2*root->value < sum)
            twoNodeSum(root, root->right, sum, res);
        else // this case occurs when 2*root == value
            twoNodeSum(root->left, root->right, sum, res);
        return res;
    }

    bool twoNodeSum( Node *left, Node *right, int target, vector<Node*> &res){
       assert(left && right);
       if(left.value + right.value > sum){
          if(twoNodeSum(left->left, right, sum, res))
             return true;
          if(twoNodeSum(left, right->left, sum, res))
             return true;
       }

       if(left->value + right->value < sum){
          if(twoNodeSum(left->right, right, sum, res))
             return true;
          if(twoNodeSum(left, right->right, sum, res))
             return true;
       }

       if(left->value + right->value == sum){
            res.push_back(left);
            res.push_back(right);
            return true;
        }
        return false;
    }
};
```

2.1.16 3Sum

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 $O(\max\{n \log n, n^{k-1}\})$。

```
// LeetCode, 3Sum
// 先排序，然后左右夹逼，注意跳过重复的数，时间复杂度 O(n^2)，空间复杂度 O(1)
class Solution {
    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;
    }
};
```

【DuplicateCase】

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

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

相关题目

- KSum, 见 §2.1.13
- 2Sum, 见 §2.1.14
- 3Sum Closest, 见 §2.1.17
- 4Sum, 见 §2.1.18

2.1.17 3Sum Closest

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 $O(n^2)$。

【左右夹逼】

```
// 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 §2.1.13

- 2Sum, 见 §2.1.14
- 3Sum, 见 §2.1.16
- 4Sum, 见 §2.1.18

2.1.18 4Sum

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, $a \le b \le c \le d$)
- 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 §2.1.13
- 2sum, 见 §2.1.14
- 3Sum, 见 §2.1.16
- 3Sum Closest, 见 §2.1.17

2.1.19 Remove Element

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.

【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));
    }
};
```

2.1.20 Next Permutation

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

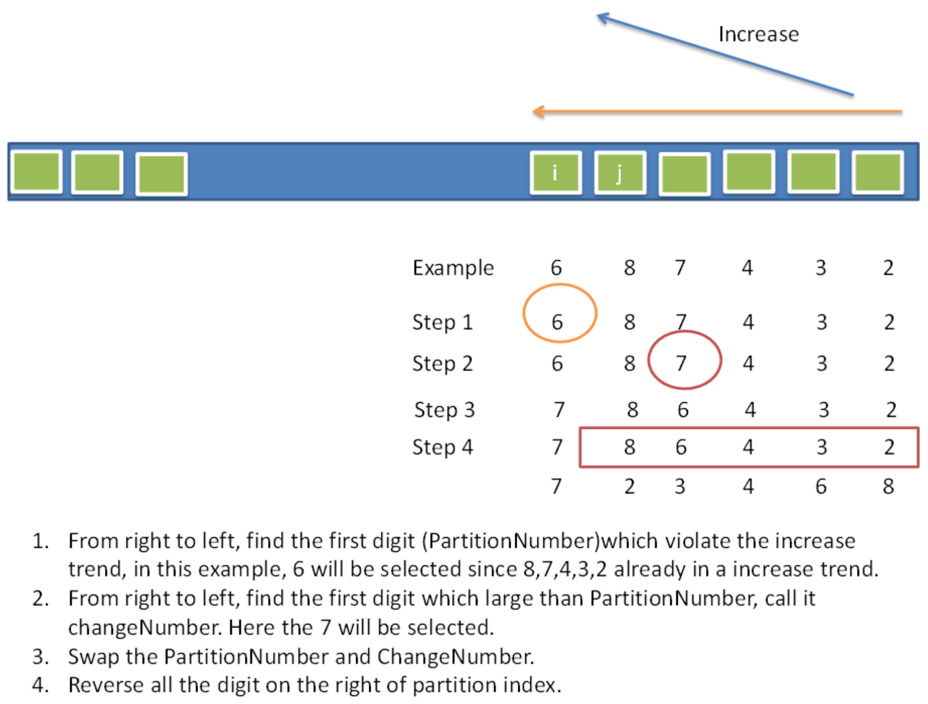
If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

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 2-1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 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;
    }
};

void nextPermutation(vector<int> &num) {
    int i=num.size()-1;
    while(i>0 && num[i]<=num[i-1])
        i--;
    if(i){
        i--;
        int j=num.size()-1;
        while(j>=i && num[j]<=num[i])
            j--;
        swap(num[i],num[j]);
        i++;
    }
    reverse(num.begin()+i,num.end());
}
```

相关题目

- Permutation Sequence, 见 §2.1.21
- Permutations, 见 §??
- Permutations II, 见 §??
- Combinations, 见 §??

2.1.21 Permutation Sequence

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;
    }
};
```

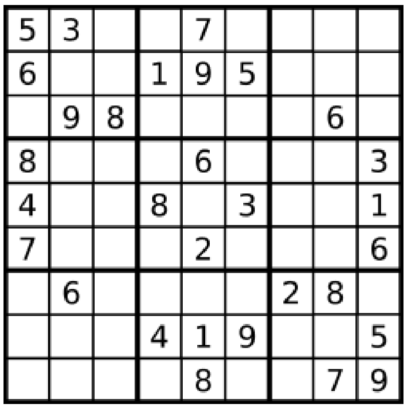
相关题目

- Next Permutation, 见 §2.1.20
- Permutations, 见 §??
- Permutations II, 见 §??
- Combinations, 见 §??

2.1.22 Valid Sudoku

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 `'.'`.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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 §??

2.1.23 Trapping Rain Water

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.

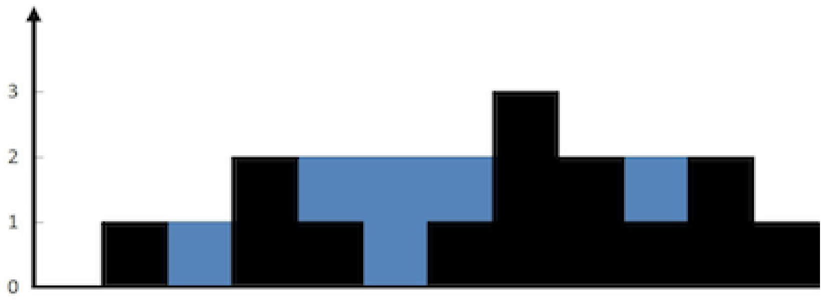


图 2-3 Trapping Rain Water

【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 §??

2.1.24 Rotate Image

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。

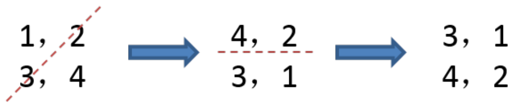


图 2-4 Rotate Image

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

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);
    }
};
```

2.1.25 Climbing Stairs

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 §??

2.1.26 Gray Code

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。

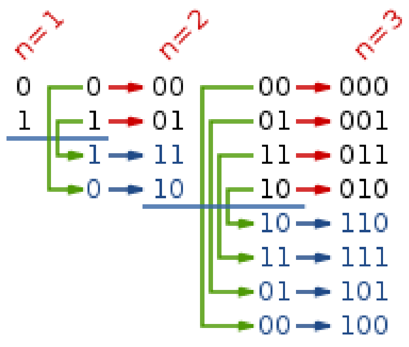


图 2-5 The first few steps of the reflect-and-prefix method.

【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;
}
```

2.1.27 Set Matrix Zeroes

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;
    }
};
```

2.1.28 Gas Station

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;
    }
};
```

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?

【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];
    }
};
```

2.1.29 Single Number

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 §2.1.30

2.1.30 Single Number II

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 §2.1.29

2.1.31 Vector Class

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

2.1.32 N Parking Slots for N-1 Cars Sorting

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

2.1.33 Word Search

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 §2.1.34

2.1.34 Robot Unique Paths

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?

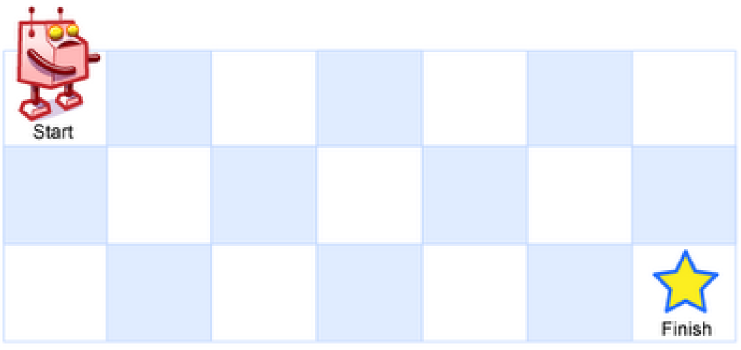


图 2-6 Robot Unique Paths

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 §2.1.33

2.1.35 FibonacciFun

Implement a Fibonacci function.

2.1.36 A+B=C+D

You're given an array of integers(eg [3,4,7,1,2,9,8]). Find the index of values that satisfy A+B = C+D, where A,B,C and D are integers values in the array.

Eg: Given [3,4,7,1,2,9,8] array The following 3+7 = 1+ 9 satisfies A+B=C+D so print (0,2,3,5)

2.1.37 Move Zeros at the rear of Array

Given a number in an array form, come up with an algorithm to push all the zeros to the end. Expectation : O(n) solution.

2.1.38 Set Zero Row, Col and Depth

Given a 3-D array, if any m[r][c][d] is <=0 mark all the cells in the entire row,col and depth as zero and return the o/p array

2.1.39 Josephus Problem

Delete every third element of an array until only one element is remaining. Tell the index of that remaining element in O(1) time complexity.

【解题思路】

This is the Josephus problem. There is a formula to calculate result directly only when $k = 2$. For other cases, there are solutions with $O(n)$ or $O(klogn)$ time complexities. In the following, n denotes the number of array in the initial circle, and k denotes the count for each step, that is, $k - 1$ people are skipped and the k-th is deleted. The people in the circle are numbered from 1 to n.

The easiest way to solve this problem in the general case is to use dynamic programming by performing the first step and then using the solution of the remaining problem. When the index starts from one, then the person at s shifts from the first person is in position $(s - 1)\%n + 1$, where n is the total number of persons. Let $f(n, k)$ denote the position of the survivor. After the k-th person is killed, we're left with a circle of $n - 1$, and we start the next count with the person whose number in the original problem was $(k\%n) + 1$. The position of the survivor in the remaining circle would be $f(n-1, k)$ if we start counting at 1; shifting this to account for the fact that we're starting at $k\%n + 1$ yields the recurrence $f(n, k) = (f(n-1, k) + k - 1)\%n + 1$, with $f(1, k) = 1$, which takes the simpler form $g(n, k) = (g(n - 1, k) + k)\%n$, with $g(1, k) = 0$. If we number the position from 0 to $n - 1$ instead.

【Algorithm】

```
int surviveWithK(int n, int k){
    return (N-pow(k-1,floor(logN/log(k-1))))+1;
}
```

2.1.40 Absent Number in Array

You are given an array of n integers which can contain integers from 1 to n only . Some elements can be repeated multiple times and some other elements can be absent from the array . Write a running code on paper which takes O(1) space apart from the input array and O(n) time to print which elements are not present in the array and the count of every element which is there in the array along with the element number . NOTE: The array isn't necessarily sorted.

2.1.41 Minimal Multiply Sum in Two Arrays

Given two arrays of same size, arrange the arrays such that a1*b1 + a2*b2 + ... + an*bn should ne minimum.

2.1.42 Divide Array into Two Half of Arrays

Given a array of size n. Divide the array in to two arrays of size n/2,n/2. such that average of two arrays is equal.

2.1.43 maximum-sum subarray

Find the maximum-sum subarray of an array.

2.1.44 Top k-th Number

Find the k-th maximal number in the given array.

2.1.45 Array Indexing

Given an array with huge number of elements. Following two operations can be performed on the array at any time

1. Find cumulative sum of first x numbers when x is input by user
2. Add/subtract value 't' from any index i of the array.

Find the most optimal way such that both the above requirements are optimized.

2.1.46 Majority Element

Find the majority element which occurs more than n/2 times in an array of n size, which contains duplicate elements in minimum time and space complexity.

2.1.47 Pair Sum

Array Pair Sum. Solve it in O(N) time complexity

```
vector< pair<int, int> > pair_sum(vector<int> arr, int s){
    vector< pair<int, int> > result;
    unordered_map< int, bool> present;
    for(int i=0; i<arr.size(); i++){
        if(present.count(s-arr[i]) != 0)
            result.push_back( make_pair(arr[i], s-arr[i]) );
        present[arr[i]] = true;
    }
    return result;
}
```

2.1.48 Gap in 2D Array

Given an nxn matrix $A(i, j)$ of integers, find maximum value $A(c, d) - A(a, b)$ over all choices of indexes such that both $c > a$ and $d > b$ in $O(n^2)$.

2.1.49 2D Array Union and Intersection

Given two array of integers write two functions that will return an Union and Intersection

2.1.50 Interleaving Sorting

Given an array sort all the elements in even positions in ascending order and odd positions in descending order

2.1.51 Pythogorean Triplets

Given an array of numbers (integers) find all pythogorean triplets ($a^2 + b^2 = c^2$). print a,b an c and the indexes.

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) { }
};
```

【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

2.2.1 Add Two Numbers

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 §??）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 §??

2.2.2 Reverse Linked List II

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;
    }
};
```

2.2.3 Partition List

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`.

【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;
    }
};
```

2.2.4 Remove Duplicates from Sorted List

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.

【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 §2.2.5

2.2.5 Remove Duplicates from Sorted List II

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.

【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 §2.2.4

2.2.6 Rotate List

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;
    }
};
```

2.2.7 Remove Nth Node From End of List

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;
    }
};
```

2.2.8 Swap Nodes in Pairs

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.

【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 §2.2.9

2.2.9 Reverse Nodes in k-Group

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`

【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 §2.2.8

2.2.10　Copy List with Random Pointer

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.

【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;
    }
};
```

2.2.11　Linked List Cycle

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 §2.2.12

2.2.12 Linked List Cycle II

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 §2.2.11

2.2.13 Reorder List

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

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;
    }
};
```

2.2.14 LRU Cache

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;
    };
```

2.2.15 BST2DLL

Convert a BST to sorted Double linked list

2.2.16 Reverse Lists behind k-th Node

1. Given number k, for Single linked list, skip k nodes and then reverse k nodes, till the end.

2. Write a program to reverse every K elements of a linked list. Example: $K = 3$; Input: $1->2->3->4->5->6->7->NULL$ Output: $3->2->1->6->5->4->7->NULL$

2.2.17 Sort Linked List

Sort a single linked list in place without using an additional node.

第 3 章
Bit Operations

3.1 ReverseBit

reverse binary bit pattern for an integer without using any string or utility methods

3.2 Even Number Present

You have an array of integers(size N), such that each integer is present an odd number of time, except 3 of them(which are present even number of times). Find the three numbers. Only XOR based solution was permitted. Time Complexity: O(N); Space Complexity: O(1).

Sample Input: 1,6,4,1,4,5,8,8,4,6,8,8,9,7,9,5,9 Sample Output: 1 6 8

3.3 Find Odd-present Numbers

Array of Integers with even number of same Integers. Find the Integer that is an odd number of times. Compare efficiency between different approaches.