# Shortlisted problems

255 261 277 284 295 296 300

# Index

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 251 | 16 | 20 | Sol: iterator impl. | easy |  |
| 252 | 16 | 16 | Sol1: enumerate all pairs  Sol2: sort + comparator  Sol3: sort | easy |  |
| 253 | 16 | 20 | Sol1: sort + priority queue  Sol2: sort | easy |  |
| 254 | 0 | 0 | Sol1: DFS (recursion)  Sol2: BFS (iteration) | medium |  |
| 255 | 56 | 64 | Sol1: idea of pre-order and in-order traversal + stack  Sol2: abuse input as stack |  | classic requires deep understanding of non-recursive traversal |
| 256 | 8 | 12 | Sol: DP | easy |  |
| 257 | 4 | 4 | Sol1: DFS(preorder traversal)  Sol2: BFS | easy |  |
| 258 | 8 | 8 | Sol: observe + prove | easy |  |
| 259 | 12 | 12 | Sol: prunning with two-pointers | medium |  |
| 260 | 16 | 20 | Sol: radix sort + bit trick | medium |  |
| 261 | 12 | 16 | Sol: graph theory+union find | difficult | classic |
| 262 |  |  | - | - | - |
| 263 | 4 | 8 | Sol: simple math | trivial |  |
| 264 | 4 | 8 | Sol1: merge sort  Sol2: BSF with priority | medium |  |
| 265 | 16 | 16 | Sol: DP | medium |  |
| 266 | 0 | 0 | Sol: Histogram | trivial |  |
| 267 | 0 | 0 | Sol: string permutation | easy |  |
| 268 | 28 | 32 | Sol: simple math | trivial |  |
| 269 | 4 | 8 | Sol: topological sort | easy |  |
| 270 | 12 | 12 | Sol: binary search | easy |  |
| 271 | 52 | 64 | Sol1: separation by length  Sol2: separation by diliminator + escaping char | interesting | classic A good example for istringstream  may have space for improvment |
| 272 | 16 | 20 | Sol1: in-order traversal  Sol2: BST+in-order traversal iterator | medium | classic  Sol2 requires deep understanding of in-order traversal |
| 273 | 4 | 4 | Sol: modularization | easy |  |
| 274 | 0 | 4 | Sol: radix sort | medium |  |
| 275 | 8 | 8 | Sol: binary search | easy |  |
| 276 | 0 | 0 | Sol1: DP  Sol2: math | easy |  |
| 277 | 160 | 200 | Sol: divide & conquer | medium-difficult | classic |
| 278 | 0 | 0 | Sol: binary search | easy |  |
| 279 | 4 | 4 | Sol1: DP  Sol2: memorized recursion  Sol3: static DP  Sol4: numer theory | easy |  |
| 280 | 44 | 52 | Sol: observe and prove | medium | may have space for improvment |
| 281 | 24 | 28 | Sol: queue+iterator | easy |  |
| 282 | 20 | 112 | Sol: recursion | difficult | the recursive function design not trivial |
| 283 | 16 | 20 | Sol: read-write pointer | trivial |  |
| 284 | 0 | 4 | Sol1: prefetch in iterator;  Sol2: copy constructor | interesting | classic Sol2 requires good understanding of C++ |
| 285 | 36 | 40 | Sol: idea of in-order traversal | easy |  |
| 286 | 120 | 144 | Sol: BFS | trivial | may have space for improvment |
| 287 | 14 | 24 | Sol: binary search | difficult |  |
| 288 | 144 | 172 | Sol: hashing (unordered\_map) | easy |  |
| 289 | 0 | 0 | Sol: bit mask | easy |  |
| 290 | 0 | 0 | Sol: hashing (unordered\_set) | trivial |  |
| 291 | 0 | 4 | Sol: recursion + flattening + prunning + hashing | rmedium | classic a good example for code optimization |
| 292 | 0 | 0 | Sol: mathmatical reduction | easy |  |
| 293 | 0 | 0 | Sol: iteration | trival |  |
| 294 | 0 | 16 | dp |  | There are definitely space to improve with (1) game theory.  (2) simple backtracking with 0ms performance! |
| 295 | 124 | 152 | Sol: heap | medium | classic may have space for improvment |
| 296 | 8 | 8 | Sol: 2D-to-1D separation + linear programming | difficult | classic |
| 297 | 28 | 44 | Sol: pre-order traversal | easy |  |
| 298 | 36 | 44 | Sol: post-order traversal | easy |  |
| 299 | 4 | 4 | Sol: histogram | trivial |  |
| 300 | 0 | 4 | Sol1: dp  Sol2: Iteration + binary search | difficult | classic |

# 

# 

# 251. Flatten 2D vector [Locked]

|  |
| --- |
| Implement an iterator to flatten a 2d vector.  For example, given 2d vector =[ [1,2], [3], [4,5,6] ]. By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1, 2, 3, 4, 5, 6]. |

**Solution: iterator pattern design + corner case checking**

定义一个函数move用于在next时指向下一个元素位置。维护当前待输出位置row, col

边界情况：（1）col到达本行尾部时需换行 (2) row在当前行长度为0时需继续指向下一行。 (3) 如果row已到最后一行需停止。

move()函数被调用的位置：在next()中，这样可以防止调用hasNext()时多次call next()。此外初始时可能要skip最前面的一些空行，因此在构造函数中也发声调用。

|  |
| --- |
| **class Vector2D** {  public:  **Vector2D(vector<vector<int>>& vec2d):** data(vec2d) {  move();  }    **int next()** {  int retv = data[row][col];  move();  return retv;  }    **bool hasNext()** {  return row < data.size();  }  private:  void move() {  if (row == data.size()) return;  if (++col == data[row].size()) {  while (++row < data.size() && data[row].size()==0);  if (row < data.size()) col = 0;  }  }  vector<vector<int>> data;  int row=0, col=-1;  }; |

# 252. Meeting Rooms [Locked]

|  |  |
| --- | --- |
| Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), determine if a person could attend all meetings.  For example,  Given [[0, 30],[5, 10],[15, 20]],  return false.   |  | | --- | | **struct Interval** {  int start;  int end;  Interval(int start\_, int end\_) : start(start\_), end(end\_) {};  }; | |

**Solution1: (brute-force) enumerate all pairs**

只要存在一对会议A, B相交(假设A.start <= B.start)，就必须返回false。如果不存在这样的pair，则可以返回true。所以brute-force的做法是比较所有的interval pair。

|  |
| --- |
| **bool canAttendMeetings(vector<Interval>& intervals)** {  for (int j = 0; j < intervals.size(); ++j) {  for (int k = j+1; k < intervals.size(); ++k) {  bool flag = intervals[k].start <= intervals[j].start;  Interval& a = flag? intervals[k]:intervals[j];  Interval& b = flag? intervals[j]:intervals[k];  if (b.start < a.end) return false;  }  }  return true;  } |

**Solution2: sort + comparator**

对每个interval A，只需要寻找A.start后面的第一个start位置，检查它是否落在A里就可以了。这是因为如果它落在A里，则已经找到一对overlapping interval pair可以返回false。如果它不落在A里，则任意start位置晚于A的interval C通过传递性可以证明也和A不相交。

A

A

B

B

C

因此我们只要把所有interval按start排序，这样每个interval之后最早的start就是排序数组中下一个interval的start.

|  |
| --- |
| **bool canAttendMeetings(vector<Interval>& intervals)** {  sort(intervals.begin(), intervals.end(),  [](Interval i1, Interval i2){ return i1.start < i2.start; }  );  for (int k = 1; k < intervals.size(); ++k)  if (intervals[k].start < intervals[k-1].end) return false;  return true;  } |

**Solution3: sort**

把intervals的端点排序。然后检查是否拍好序后端点类型为start, end交替出现。注意重合的timepoints需要全部读过之后才能检查。

|  |
| --- |
| **bool canAttendMeetings(vector<Interval>& intervals)** {  vector<pair<int, bool>> timepoints;  for (auto& i : intervals) {  timepoints.emplace\_back(i.start, true);  timepoints.emplace\_back(i.end, false);  }    sort( timepoints.begin(), timepoints.end(),  [](pair<int, bool>p1, pair<int, bool>p2) { return p1.first < p2.first; }  );    int nstart = 0, nend = 0;  for (int k = 0; k < timepoints.size(); ) {  int t = timepoints[k].first;  while (k < timepoints.size() && timepoints[k].first == t) {  if (timepoints[k].second) ++nstart; else ++nend;  ++k;  }  if (nstart!=nend && nstart!=nend+1) return false;  }  return true;  } |

# 253. Meeting Rooms II [Locked]

|  |
| --- |
| Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.  For example,  Given [[0, 30],[5, 10],[15, 20]],  return 2. |

**Solution1: sort + priority queue**

假设时间t以前开始的meeting已经有一个最优安排。在时间t开始的meeting，需要检查有哪些房间的上一个会议已经结束(即end time <= t)，如果所有会议都没有结束，即end的最小值>t，则需要开一个新房间。否则我们只要把会议放在任意一间在时间t已经空出的房间就好。

因此只要按照interval开始时间顺序加入interval，同时用一个priority queue维护所有已开房间的最后结束时间，最后queue的大小即为房间数。

|  |
| --- |
| int minMeetingRooms(vector<Interval>& intervals) {  sort(intervals.begin(), intervals.end(),  [](Interval i1, Interval i2){return i1.start < i2.start;});  priority\_queue<int, vector<int>, greater<int>> heap;    for (Interval i : intervals) {  if (heap.size() != 0 && heap.top() <= i.start) heap.pop();  heap.push(i.end);  }  return heap.size();  } |

**Solution2: sort**

我们事实上数的是任意时间最大并发的会议个数。这个数量的改变只发生在interval开始和结束的时候。所以一个比较简单的办法是把所有的time point排序并纪录当时发生的事件，在end发生时interval数－1,start发生时interval数+1。注意让end先被计数。

|  |
| --- |
| int minMeetingRooms(vector<Interval>& intervals) {  vector<pair<int, bool>> timepoints;  for (auto& i : intervals) {  timepoints.emplace\_back(i.start, false);  timepoints.emplace\_back(i.end, true );  }  sort(timepoints.begin(), timepoints.end(),  [](pair<int, bool>& p1, pair<int, bool>& p2) {  return p1.first < p2.first || (p1.first==p2.first && p1.second && !p2.second);  }  );    int num\_rooms = 0, num\_concurrent = 0;  for (auto t : timepoints) {  if (t.second) --num\_concurrent; else ++num\_concurrent;  num\_rooms = max(num\_rooms, num\_concurrent);  }  return num\_rooms;  } |

# 254. Factor Combination [Locked]

|  |
| --- |
| Numbers can be regarded as product of its factors. For example,  8 = 2 x 2 x 2;  = 2 x 4.  Write a function that takes an integer n and return all possible combinations of its factors.  Note:  Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is [2, 6], not [6, 2].  You may assume that n is always positive.  Factors should be greater than 1 and less than n. |

12

2x

6

3x

4

2x2x3

2x6

3

2x

2

2x

3x4

**Solution1: recursion**

递归结构：factorization(x, l)

x为被分解的target，l为分解出的item的lower bound

递推关系：

假设x的factor k中满足k <= n/k, k > l的包括：k1, k2, ...km,

36

18

12

9

6

9

3

6

4

2x

2x

3x

4x

3x

3x

6x

3x

对每个k，在factorization(n, l)中加入：

1. k \* (n/k)
2. k \* factorization(n/k, k)

例如：对12的分解如右图 解为: [2, 6], [2, 2, 3], [3, 4]

初始状态：x = n, l = 0

结束状态： x没有满足上述条件的factor

此题不适用memorized recursion 因为即使递归到分解同一个数（如下面例子中分解9），因为path上最大数的不同，仍然可能会导致不同的剪枝。因此其实搜索过程中overlapping的结构并不多。

|  |
| --- |
| vector<vector<int>> getFactors(int n) {  vector<vector<int>> result;  vector<int> prefix;  getFactors\_recur(n, 2, prefix, result);  return result;  }  void getFactors\_recur(int x, int lb, vector<int>& prefix, vector<vector<int>>& result) {  for (int f = lb; f <= x/f; ++f) {  if (x%f) continue;  prefix.push\_back(f);  result.push\_back(prefix);  result.back().push\_back(x/f);  getFactors\_recur(x/f, f, prefix, result);  prefix.pop\_back();  }  } |

**Solution2: BFS**

因为我们需要返回上述搜索树的所有非根结点对应的expressions，所以也可以采取BFS的方式，从而避免递归。

|  |
| --- |
| vector<vector<int>> getFactors(int n) {  vector<vector<int>> result;  vector<vector<int>> pre;    for (int k = 2; k <= n/k; ++k) {  if (n%k==0) pre.push\_back({k, n/k});  }    while (!pre.empty()) {  vector<vector<int>> cur;  for (auto expression : pre) {  result.push\_back(expression);  int last = expression.back();  int lb = \*(expression.end()-2);  for (int k = lb; k <= last/k; ++k) {  if (last % k) continue;  cur.push\_back(expression);  cur.back().back() = k;  cur.back().push\_back(last/k);  }  }  swap(pre, cur);  }    return result;  } |

# 

# 255. Verify Preorder Sequence in Binary Search Tree [Locked]

|  |
| --- |
| Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.  You may assume each number in the sequence is unique.  Follow up:  Could you do it using only constant space complexity? |

**Solution 1：In-order traversal + stack**

当我们以前序遍历顺序访问时，出现在preorder[k]之前的元素包括：

1. 中序遍历中在它之前的元素（右图中蓝色）
2. 它到root路径上的祖先中向左branch的那些结点（右图中红色）。这是中序遍历时的栈内元素。

前序遍历为我们提供了每个node到root的路径，以及这条路径中每个node的左子树中的元素（如果当前node在祖先node的右子树中）。

验证BST时，我们关心的是蓝色结点的最大值和红色结点的最小值。它们分别对应最后一个出栈（即最后一个被中序遍历到的）元素left，和最后一个入栈的元素。

算法：从左到右扫描preorder。对新元素v：

(1) 更新栈：把当前栈中<=v的元素全部退栈，并用它们update left。

(2) 如果当前left>=v，则违反BST定义，返回false。

(3) 把v进栈

|  |
| --- |
| bool verifyPreorder(vector<int>& preorder) {  int left = INT\_MIN;  stack<int> stk;  for (int v : preorder) {  while (!stk.empty() && stk.top()<=v) {  left = stk.top();  stk.pop();  }  if (left>=v) return false;  stk.push(v);  }  return true;  } |

**Solution 2：Recursion + abuse input for stack**

因为栈内元素是preorder[0...i]的一个子序列，因此可以把stack写在输入数组里。

|  |
| --- |
| bool verifyPreorder(vector<int>& preorder) {  int left = INT\_MIN;  int pos = -1;  for (int v : preorder) {  while (pos>=0 && preorder[pos]<=v) left = preorder[pos--];  if (left>=v) return false;  preorder[++pos] = v;  }  return true;  } |

# 256. Paint House [Locked]

|  |
| --- |
| There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.  The cost of painting each house with a certain color is represented by a n x 3 cost matrix. For example, costs[0][0] is the cost of painting house 0 with color red;costs[1][2] is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.  Note:  All costs are positive integers. |

**Solution: DP**

递归结构：mincost(k, c) 表示粉刷前k幢房子，且最后一幢房子为c的最优方案。

最后返回结果为k=#house时，mincost(k, \*)中的最小值

递推关系：

初始状态：mincost(0, c) = cost(0, c)

由上式可以得到一个2D DP。最后注意到我们在动态规划时只需要维护最后两个k对应的结果，因此把问题压缩为1D空间，对这个问题来说因为c有3中可能，因此空间复杂度是O(1)。

|  |
| --- |
| **int minCost(vector<vector<int>>& costs)** {  int pre[3] = {0, 0, 0};  for (vector<int> v : costs) {  int cur[3] = {  v[0] + min(pre[1], pre[2]),  v[1] + min(pre[0], pre[2]),  v[2] + min(pre[0], pre[1])  };  swap(pre, cur);  }  auto it = min\_element(pre, pre+3);  return \*it;  } |

# 257. Binary Tree Paths

|  |
| --- |
| Given a binary tree, return all root-to-leaf paths.  For example, given the following binary tree:  1  / \  2 3  \  5  All root-to-leaf paths are: ["1->2->5", "1->3"] |

**Solution: preorder traversal**

在前序遍历中，访问每个子结点时把它输入prefix，见到叶结点时把prefix输入最终结果。

|  |
| --- |
| **vector<string> binaryTreePaths(TreeNode\* root)** {  if (root == NULL) return vector<string>{};  vector<string> result;  string prefix = "";  binaryTreePaths\_recur(root, result, prefix);  return result;  }  void binaryTreePaths\_recur(TreeNode \*root, vector<string> &result, string prefix) {  if (prefix.size() > 0) prefix += "->";  prefix += to\_string(root->val);    if (root->left == NULL && root->right == NULL) { //leaf node  result.push\_back(prefix);  }  else {  if (root->left) binaryTreePaths\_recur(root->left, result, prefix);  if (root->right) binaryTreePaths\_recur(root->right, result, prefix);  }  } |

**Solution: BFS**

|  |
| --- |
| vector<string> binaryTreePaths(TreeNode\* root) {  vector<string> result;  queue<pair<TreeNode\*, string>> q;  if (root != NULL) q.emplace(root, "");    while (!q.empty()) {  TreeNode\* t = q.front().first;  string prefix = q.front().second;  q.pop();    if (!prefix.empty()) prefix += "->";  prefix += to\_string(t->val);  if (t->left == NULL && t->right == NULL) result.push\_back(prefix);  if (t->left) q.emplace(t->left, prefix);  if (t->right) q.emplace(t->right, prefix);  }    return result;  } |

# 258. Add Digits

|  |
| --- |
| Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.  Could you do it without any loop/recursion in O(1) runtime?  For example:  Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it. |

**Solution: observe & prove**

我们可以把add digit操作视为一个状态转移过程（如下图）。我们关心的是(1) 状态转移是否收敛 (2) 状态转移中是否有不变量可以利用。

10

28

37

100

46

55

20

299

389

200

398

488

结束状态：0-9 (一位数），其中除了0本身没有其它数会变为0

状态转移：对两位数开始的数字[nk...n1n0]会转移成(nk+...n0)，其中数字[nk … n1n0] ＝

状态转移使它减少了，这个差是正数，且是9的倍数。因此状态收敛，且转移过程中mod 9的余数恒定。

|  |
| --- |
| **int addDigits(int num)** {  return num == 0? 0 : ((num-1) % 9)+1;  } |

# 259. 3 Sum Smaller [Locked]

|  |
| --- |
| Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.  Follow up:  Could you solve it in O(n^2) runtime? |
| For example, given nums = [-2, 0, 1, 3], and target = 2.  Return 2. Because there are two triplets which sums are less than 2:  [-2, 0, 1]  [-2, 0, 3] |

-2

0

1

1

2

target =1

l=1, r=4: 3 pairs

l=2, r=4: 0 pairs (because 1+2>2)

l=2, r=3: 1 pair

l=2, r=4: 0 pairs (because 1+2>1)

l=2, r=3: 0 pairs (becuase 1+1>1)

l=3, r=4: 0 pairs (because 1+2>0)

**Solution: two pointer**

和3-sum问题的类比：类似3-sum，我们通过遍历第一个元素nums[k]（图中红色位置）把问题转化为sorted 1D矩阵里的2-sum问题。

核心优化：

1. 我们可以固定第二个元素（图中绿色）nums[l]的位置l，同时找到最右元素（途中蓝色）的位置r，使nums[k]+nums[l]+nums[r]<target。这个tuple 同时表明r位置出现在l+1, … r的位置都能得到更小的tuple。
2. 搜索l和r的过程可以利用two-pointer的思路从2D优化到1D。这时因为已知nums[l-1]+nums[r+1]>=target，而当前nums[l]>nums[l-1]，因此nums[l]+nums[r+1]>=target。因此对一个新的l，r的搜索从上一行开始。

进一步优化：

当nums[k]+nums[k+1]+nums[k+2]>=target时，可以不必继续新的循环 (即上图中完成第一个外循环时就可以结束）

|  |
| --- |
| **int threeSumSmaller(vector<int>& nums, int target)** {  sort(nums.begin(), nums.end());  int count = 0;  for (int k = 0; k+2 < nums.size() && nums[k]+nums[k+1]\*2<target; ++k) {  int target1 = target-nums[k];  for (int l = k+1, r = nums.size()-1; l < r; ) {  if (nums[l] + nums[r] < target1) count += r-l++;  else r--;  }  }  return count;  } |

# 260. Single Number III

|  |
| --- |
| Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.  Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity? |
| For example:  Given nums = [1, 2, 1, 3, 2, 5], return [3, 5].  Note:  The order of the result is not important. So in the above example, [5, 3] is also correct. |

**Solution: radix sort + bit trick**

对数字的每一位，如果两个single number相等，则有偶数个1,否则有奇数个1 .通过异或所有数字，我们得到有奇数个1的位。

我们取其中任意一位（位运算比较容易的是最后一位）。然后我们把数字按照该位的0,1分组，则问题转化为single number。

|  |
| --- |
| **vector<int> singleNumber(vector<int>& nums)** {  int diff = 0;  for (int v : nums) diff ^= v;  diff &= (-diff); //isolate rightmost bit 1  int v1 = 0, v2 = 0;  for (int v: nums) if ((v & diff) == 0) v1 ^= v; else v2 ^= v;  return vector<int>{v1, v2};  } |

# 261. Graph Valid Tree [Locked]

|  |
| --- |
| Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.  Note: you can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges. |
| For example:  Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.  Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false. |

**Solution: graph theory + union find**

图是一棵树的充要条件是满足以下三个条件中的两个（另一个自动满足）：

1. 顶点数是边数+1
2. 图只有一个联通分量
3. 图中没有环

对给定问题的表示，检查条件(1)(2)比较简单。

条件(1) 直接由输入给出

条件(2) 对图作union find，并纪录访问过的顶点总数。(BFS/DFS等方法也行，但union-find更直接。）

|  |
| --- |
| **bool validTree(int n, vector<pair<int, int>>& edges)** {  if (n==0) return true;  return (edges.size() == n-1) && is\_connected(n, edges);  }  inline bool is\_connected(int n, vector<pair<int, int>>& edges) {  vector<int> sz(n, 1);  vector<int> prev(n, 0);  iota(prev.begin(), prev.end(), 0);    int max\_sz = 1;  for (auto& e : edges) {  int r1 = e.first;  int r2 = e.second;  while (prev[r1] !=r1) r1 = prev[r1];  while (prev[r2] !=r2) r2 = prev[r2];  if (sz[r1]>sz[r2]) swap(r1, r2); //make sure sz[r1]<=sz[r2]  prev[r1] = r2;  sz[r2] += sz[r1];  max\_sz = sz[r2];  }  return max\_sz == n;  } |

262. [Database problem]

# 

# 263. Ugly Number

|  |
| --- |
| Write a program to check whether a given number is an ugly number.  Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.  Note that 1 is typically treated as an ugly number. |

**Solution: simple math**

把数字尽量除去2,3,5，最后检查是否余下的因子是1.

|  |
| --- |
| **bool isUgly(int num)** {  if (num <= 0) return false;  while (num % 2 == 0) num/=2;  while (num % 3 == 0) num/=3;  while (num % 5 == 0) num/=5;  return num==1;  } |

# 264. Ugly Number II

|  |
| --- |
| Write a program to find the n-th ugly number.  Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.  Note that 1 is typically treated as an ugly number. |

**Solution: merge sort**

维护三个队列l0, l1, l2，分别存储最大质因子是2, 3, 5的数。因此从lk出队列的数只需要乘以lk和它之后的质因子，并进相应队列。出队列选择最小数。

|  |
| --- |
| int nthUglyNumber(int n) {  list<long> lists[3]; //maximal prime factor is 2, 3, 5  const long coef[3] = {2, 3, 5};  long cur = 1;  for (int k = 1, grp = 0; k < n; k++) {  for (int j = grp; j < 3; ++j) lists[j].push\_back (cur \*coef[j]);    long fronts[3] = {lists[0].front(), lists[1].front(), lists[2].front()};  long\* it = min\_element(fronts, fronts+3);    cur = \*it;  grp = it-fronts;  lists[grp].pop\_front();  }  return cur;  } |

**Solution: BFS with priority**

上面解法的问题在于队列里可能放太多数字。解决办法是把已经出队列的数放在一个队列里。新加入队列的数必定是出队列的数里某个数\*2或\*5或\*3。

此外因为factor是固定的，所以我们可以把所有数据cache成static的。

|  |
| --- |
| int nthUglyNumber(int n) {  static vector<int> history;  static int idx[3];  static const int factors[3] = {2, 3, 5};  static priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>  q;  if (history.size()==0) {  history.push\_back(1);  idx[0] = idx[1] = idx[2] = 0;  q.emplace(2, 0); q.emplace(3, 1); q.emplace(5, 2);  }    for (int k = history.size(); k < n; ++k) {  int minv = q.top().first， minj = q.top().second;  q.pop();  if (minv == history.back()) --k;  else history.push\_back(minv);  idx[minj]++;  q.emplace(history[idx[minj]]\*factors[minj], minj);  }  return history[n-1];  } |

# 265. Paint House II [Locked]

|  |
| --- |
| There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.  The cost of painting each house with a certain color is represented by a n x k cost matrix. For example, costs[0][0] is the cost of painting house 0 with color 0; costs[1][2] is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.  Note: All costs are positive integers.  Follow up: Could you solve it in O(nk) runtime? |

**Solution: DP**

维护paint到第j个房子时，假设这个房子是颜色k对应的总代价。我们只需要maintain代价最小的两个颜色极其代价。这样当paint第j+1个房子时，如果前一个房子最小代价对应颜色与它相同，就paint第2种颜色，否则paint第一种颜色。

|  |
| --- |
| **int minCostII(vector<vector<int>>& costs)** {  if (costs.size() == 0 || costs[0].size()==0) return 0;  int cost1 = 0, cost2 = 0;  int color1 = -1, color2 = -1;    for (int j = 0; j < costs.size(); ++j) {  int ncost1 = INT\_MAX, ncost2 = INT\_MAX;  int ncolor1 = -1, ncolor2 = -1;  for (int k = 0; k < costs[0].size(); ++k) {  int cost\_k = (color1 == k? cost2 : cost1) + costs[j][k];  if (cost\_k <= ncost2) {  ncolor2 = k;  ncost2 = cost\_k;  }  if (ncost2 < ncost1) { //maintain ordering of top two candidates  swap(ncolor1, ncolor2);  swap(ncost1, ncost2);  }  }  cost1 = ncost1, cost2 = ncost2, color1 = ncolor1, color2 = ncolor2;  }  return cost1;  } |

# 266. Palindrome Permutation [Locked]

|  |
| --- |
| Given a string, determine if a permutation of the string could form a palindrome.  Hint:  Consider the palindromes of odd vs even length. What difference do you notice?  Count the frequency of each character.  If each character occurs even number of times, then it must be a palindrome. How about character which occurs odd number of times? |
| For example,  "code" -> False, "aab" -> True, "carerac" -> True. |

**Solution: histogram**

就是数一下是不是最多只有一个字符出现奇数次。

|  |
| --- |
| **bool canPermutePalindrome(string s)** {  bool cnt[256] = {false};  int count = 0;  for (char c : s) {  cnt[c] = !cnt[c];  if (cnt[c]) ++count; else --count;  }  return count <= 1;  } |

# 267. Palindrome Permutation II [Locked]

|  |
| --- |
| Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.  For example:  Given s = "aabb", return ["abba", "baab"].  Given s = "abc", return [].. |

**Solution: string permutation**

穷举palindrome的左边一半所有可能的permutation，然后补全右边一半即可。

|  |
| --- |
| **vector<string> generatePalindromes(string s)** {  string odd\_chars;  string even\_chars(s.size()/2, 0);  //for each char with n occurrence, put n/2 into even\_char  //put char with odd occurrence into odd\_char, this happens at most 1 time  array<int, 256> count;  count.fill(0);  for (char c : s) count[c]++;  int idx = 0;  for (int c = 0; c < 256; ++c) {  fill(even\_chars.begin()+idx, even\_chars.begin()+idx+count[c]/2, c);  idx += count[c]/2;  if (count[c]%2 == 1) odd\_chars += c;  }  if (odd\_chars.size()>1) return vector<string>{};  // generate the permutations for the left part  vector<string> result;  do{  result.push\_back(even\_chars);  }while (next\_permutation(even\_chars.begin(), even\_chars.end()));  // complete the strings  for (string &str : result) {  string str1(str);  reverse(str1.begin(), str1.end());  str += odd\_chars + str1;  }  return result;  } |

# 268. Missing Number

|  |
| --- |
| Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.  Note:  Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity? |
| For example, given nums = [0, 1, 3] return 2. |

**Solution: simple math**

比较0,1,2...n的和以及数组元素之和，它们的差是missing number。为了防止overflow我们在每个循环里做一次加法一次减法。

|  |
| --- |
| **int missingNumber(vector<int>& nums)** {  int result = 0;  for (int k = 0; k < nums.size(); ++k) result += k - nums[k] + 1;  return result;  } |

# 269. Alien Directory [Locked]

|  |
| --- |
| There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, wherewords are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.  Note:  You may assume all letters are in lowercase.  If the order is invalid, return an empty string.  There may be multiple valid order of letters, return any one of them is fine. |
| For example,  Given the following words in dictionary,  [ "wrt", "wrf","er","ett","rftt" ]  The correct order is: "wertf". |

**Solution: topological sort**

首先通过比较两个相邻的单词，它们的第一个不同的字母（如果有的话）决定了先后顺序，这样构造了一个graph。然后我们拓扑排序即可。

|  |
| --- |
| string alienOrder(vector<string>& words) {  array<int, 26>count; count.fill(-1);  for (string w : words) //set count for chars of interest as 0, others as -1  for (char c : w) count[c-'a'] = 0;    unordered\_map<char, unordered\_set<char>> edges; //c2 is in edges[c1] means c2 > c1  for (int k = 0; k < words.size()-1; ++k) {  // find first char in word[k] and word[k+1] that is different  int j = 0;  while (j < words[k].size() && j < words[k+1].size() && words[k][j]==words[k+1][j])  ++j;  // add an edge due to comparison between the two word  if (j < words[k].size() && j < words[k+1].size())  edges[words[k][j]].insert(words[k+1][j]);  }  //topological sort  for (pair<char, unordered\_set<char>> e : edges)  for (char c: e.second) count[c-'a']++;  queue<char> q;  int alphabet\_size = 0; //count number of nodes  for (int k = 0; k < 26; ++k) {  if (count[k] == 0) q.push(k+'a');  if (count[k] >= 0) ++alphabet\_size;  }    string result;  while (!q.empty()) {  for (char c : edges[q.front()]) {  count[c-'a']--;  if (count[c-'a']==0) q.push(c);  }  result += q.front();  q.pop();  }  return result.size()==alphabet\_size?result:"";  } |

# 270. Closest Binary Search Tree Value [Locked]

|  |
| --- |
| Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.  Note: Given target value is a floating point. You are guaranteed to have only one unique value in the BST that is closest to the target. |

**Solution: Binary search**

BST的每个叶子结点对应连续不相交的子区间。通过二分搜索narrow down子区间，当到达根结点或NULL时，即检索到包含target的最小子区间，返回两端中离target较近的即可。

|  |
| --- |
| **int closestValue(TreeNode\* root, double target)**{  TreeNode \*p = root, \*q = root, \*t = root;  while (p->left != NULL) p = p->left;  while ( q->right != NULL) q = q->right;  if (target <= p->val) return p->val;  if (target >= q->val) return q->val;  int minv = p->val, maxv = q->val;  while (t) {  if (t->val == target) return t->val;  if (t->val < target) {  minv = t->val;  t = t->right;  }  else {  maxv = t->val;  t = t->left;  }  }  return target \* 2 < minv + maxv? minv : maxv;  } |

# 271. Encode and Decode Strings [Locked]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.  Machine 1 (sender) has the function:   |  | | --- | | **string encode(vector<string> strs)** {  // ... your code  return encoded\_string;  } |   Machine 2 (receiver) has the function:   |  | | --- | | **vector<string> decode(string s)** {  //... your code  return strs;  } |   So Machine 1 does:   |  | | --- | | string encoded\_string = encode(strs); |   and Machine 2 does:   |  | | --- | | vector<string> strs2 = decode(encoded\_string); |   strs2 in Machine 2 should be the same as strs in Machine 1.  Implement the encode and decode methods.  Note:   * The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters. * Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless. * Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm. |

**Solution: running length encoding + istringstream**

受pgm格式的启发，我们在每个string前面append它的长度加一个空格，这样我们就能正确地分割串了。

例如：输入为[”str1“, ”str2“, ”str3“]

encoded串为“4 str14 str24 str3"

|  |
| --- |
| class Codec {  public:  // Encodes a list of strings to a single string.  string encode(vector<string>& strs) {  string result("");  for (string s : strs) result += to\_string(s.size()) + ' ' + s;  return result;  }  // Decodes a single string to a list of strings.  vector<string> decode(string s) {  vector<string> result;  istringstream iss(s);  while (true) {  int len = 0;  iss >> len;  if (iss.eof()) break;  iss.ignore(1); //skip space  string str(len, 0);  iss.read(&str[0], len);  result.push\_back(str);  }  return result;  }  }; |

**Solution: deliminator + escaping character + istringstream**

把单词中间用','分割，对原单词里的逗号改为"\,"，原单词里的’\‘改为"\\"。

这样，当decoder看到2n个\时，把它decode成n个\，

看到2n+1个\时，后面一定跟一个','，把它decode为n个'\'和一个','

看到孤立的','时，把它识别为分隔符。

|  |
| --- |
| // Encodes a list of strings to a single string.  string encode(vector<string>& strs) {  string result("");  for (string s : strs) {  result.push\_back(',');  for (char c : s) {  if (c == ',' || c == '#') result += '#';  result += c;  }  }  return result;  }  // Decodes a single string to a list of strings.  vector<string> decode(string s) {  vector<string> result;  for (int k = 0; k < s.size(); ++k) {  switch (s[k]) {  case ',': result.push\_back(""); break;  case '#': result.back() += s[++k]; break;  default: result.back() += s[k]; break;  }  }  return result;  } |

# 

# 272. Closest Binary Search Tree Value II [Locked]

|  |
| --- |
| Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.  Note:  Given target value is a floating point.  You may assume k is always valid, that is: k ≤ total nodes.  You are guaranteed to have only one unique set of k values in the BST that are closest to the target.    Follow up:  Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)? |
| Hint:  Consider implement these two helper functions:  getPredecessor(N), which returns the next smaller node to N.  getSuccessor(N), which returns the next larger node to N.  Try to assume that each node has a parent pointer, it makes the problem much easier.  Without parent pointer we just need to keep track of the path from the root to the current node using a stack.  You would need two stacks to track the path in finding predecessor and successor node separately. |

**Solution1：In-order Traversal**

用一个sliding window纪录当前k个最接近target的数。

|  |
| --- |
| vector<int> closestKValues(TreeNode\* root, double target, int k) {  stack<TreeNode\*> stk;  queue<int> q;  TreeNode \*t = root;  while (!stk.empty() || t != NULL) {  if (t != NULL) {  stk.push(t);  t = t->left;  }  else {  int v = stk.top()->val;  if (q.size()==k && abs(target - q.front()) > abs(target - v)) q.pop();  if (q.size()<k) q.push(v);  t = stk.top()->right;  stk.pop();  }  }  vector<int> result;  while (!q.empty()) {  result.push\_back(q.front());  q.pop();  }  return result;  } |

**Solution2：binary search + in-order traversal**

搜索target，从而直接构造非递归中序遍历时第一个>=target(或<target时)iterator的状态。

用stack保存path到node的路径，然后对这两个path，分别向前和向后iterate

|  |
| --- |
| void next(TreeNode\* &t, stack<TreeNode\*> &stk) {  if (t->right != NULL) {  t = t->right;  while (t) {  stk.push(t);  t = t->left;  }  }  if (stk.empty()) t = NULL;  else {  t = stk.top();  stk.pop();  }  }    void prev(TreeNode\* &t, stack<TreeNode\*> &stk) {  if (t->left != NULL) {  t = t->left;  while (t) {  stk.push(t);  t = t->right;  }  }  if (stk.empty()) t = NULL;  else {  t = stk.top();  stk.pop();  }  }  **vector<int> closestKValues(TreeNode\* root, double target, int k)** {  stack<TreeNode\*> stk1, stk2;  while (root) {  if (target < root->val) {  stk1.push(root);  root = root->left;  }  else {  stk2.push(root);  root = root->right;  }  }  TreeNode\* t1 = NULL, \* t2 = NULL; //t1: next, t2: prev  if (!stk1.empty()) {t1 = stk1.top(); stk1.pop();}  if (!stk2.empty()) {t2 = stk2.top(); stk2.pop();}    vector<int> result;  for (int j = 0; j < k; ++j) {  if (t1 == NULL || (t2 != NULL && t1->val + t2->val > 2\*target)) {  result.push\_back(t2->val);  prev(t2, stk2);  }  else {  result.push\_back(t1->val);  next(t1, stk1);  }  }  return result;  } |

# 273. Integer to English Words

|  |
| --- |
| Convert a non-negative integer to its english words representation.  Given input is guaranteed to be less than 231 - 1. |

**Solution: modularization in programming + corner case**

首先0是一个特例，对它单独处理。  
对其余情况，把数字分为三位一截，转为英语后分别加上Billion, Millon, Thousand的后缀，最后一段没有后缀。

对小于1000的数：百位和个位数Map到[空串], One … Nine，十位Map到[空串]，Ten, … Ninty。

但如果最后两位是11-19，Map到Eleven...Ninteen.

因为整个处理中空格比较复杂，所以我们单独实现一个helper函数append，实现把单词加入输出串并适当地加入空格的操作。

|  |
| --- |
| **string numberToWords(int num)** {  if (num == 0) return "Zero";  string s;  static const string suffix[] = {"Billion", "Million", "Thousand", ""};  for (int k = 0, base = 1000000000; k < 4; ++k, base/=1000)  numberToWords2(num/base%1000, s, suffix[k]);  return s;  }  // turn a number between 1-999 into english words and append suffix  // and insert it to the tail of s  inline void numberToWords2(int num, string &s, const string &suffix) {  static const string words1[] = {"", "One", "Two", "Three", "Four", "Five", "Six",  "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen",  "Fifteen", "Sixteen","Seventeen", "Eighteen", "Nineteen"};  static const string words10[] = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty",  "Sixty", "Seventy", "Eighty", "Ninety"};    if (num == 0) return;    if (num/100) append(s, words1[num/100] + " Hundred");// hundred digit  int n = num%100; // last two digits  if (n >= 20) {  append(s, words10[n/10]); // ten digit  if (n%10) append(s, words1[n%10]); // one digit  }  else if (n>0) append(s, words1[n]); //directly lookup for 0-19    append(s, suffix);  }  // append word to phrase. if word is empty, do nothing.  // add space before word if phrase is non empty  inline void append(string &phrase, const string &word) {  if (word.size() == 0) return;  if (phrase.size() == 0) phrase = word;  else phrase += ' ' + word;  } |

# 274. H-Index

|  |
| --- |
| Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index. |
| According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each."  For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, his h-index is 3.  Note: If there are several possible values for h, the maximum one is taken as the h-index. |

**Solution: radix sort**

如果一个作者有N篇文章，那么他的H-index最多为N。因此我们划分N+1个bin，第k个bin记录citation为k的数。最后一个bin记录citation>=N的数。然后从右向左累加并搜索，就可以知道citation大于等于k的篇数。

|  |
| --- |
| **int hIndex(vector<int>& citations)** {  vector<int> count(citations.size()+1, 0);  for (int v : citations) {  count[min(v, (int)citations.size())]++;  }  for (int k = citations.size(), sum = 0; k>0; k--) {  sum += count[k];  if (sum >= k) return k;  }  return 0;  } |

# 275. H-Index II

|  |
| --- |
| Follow up for H-Index: What if the citations array is sorted in ascending order? Could you optimize your algorithm? |

**Solution: binary search**

问题转化为寻找最小的index k满足citation[k]>=N-k。可以通过二分查找解决。

|  |
| --- |
| **int hIndex(vector<int>& citations)** {  //let N = citation.size()  //find minimal k: citation[k]+k >= N  const int N = citations.size();  if (N == 0) return 0;  if (citations[0]>=N) return N;  if (citations[N-1]==0) return 0; //acceleration  //find last k that citation[k]+k < N  int first = 0, last = N-1;  while (first < last-1) {  int mid = (last+first)/2;  int sum = citations[mid]+mid;  if (sum >= N) last = mid-1;  else first = mid;  }  if (last > first && citations[last]+last<N) first++;  return N - first - 1;  } |

# 276. Paint Fence [Locked]

|  |
| --- |
| There is a fence with n posts, each post can be painted with one of the k colors.  You have to paint all the posts such that no more than two adjacent fence posts have the same color.  Return the total number of ways you can paint the fence.  Note: n and k are non-negative integers. |

**Solution: DP**

递归结构 (c1, c2), 其中c1是最后两个fence不一样的可能性，c2是最后两个fence一样的可能性

递推关系 c1=(c1+c2)\*(k-1), c2 = c1

初始条件 c1[0]=k, c2[0]=0

|  |
| --- |
| int numWays(int n, int k) {  if (n <= 0) return 0;  int c1 = k, c2 = 0;  for (int j = 2; j <= n; ++j) {  int total = c1+c2;  c2 = c1;  c1 = total \* (k-1);  }  return c1+c2;  } |

**Solution2: analytical solution of recursive equation**

递推关系化简：令c[t] = c1[t], c[t-1] = c2[t]，则我们有c[t] = (k-1)c[t-1] + (k-1)c[t-2]

c[t]的通项表达式式是 (at-bt)/(a-b)\*k，a+b=(k-1), a\*b = 1-k

这样c1[t]=c[t] = (at-bt)/(a-b)\*k，c2[t]=c[t-1]= (at-1-bt-1)/(a-b)\*k

|  |
| --- |
| int numWays(int n, int k) {  if (n <= 0 || k == 0) return 0;  if (k == 1) return n<3;  double root = (k-1)\*(k-1)/4.0+k-1;  double a = (k-1)/2.0 + sqrt(root);  double b = (k-1)/2.0 - sqrt(root);  int c1 = round((pow(a, n) - pow(b, n))\*k/(a-b));  int c2 = round((pow(a, n-1) - pow(b, n-1))\*k/(a-b));  return (c1+c2);  } |

# 277. Find the Celebrity [Locked]

|  |
| --- |
| Suppose you are at a party with n people (labeled from 0 to n - 1) and among them, there may exist one celebrity. The definition of a celebrity is that all the other n - 1people know him/her but he/she does not know any of them.  Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).  You are given a helper function bool knows(a, b) which tells you whether A knows B. Implement a function int findCelebrity(n), your function should minimize the number of calls to knows.  Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return -1. |

**Solution: divide and conquer**

如果knows(a, b)返回true，那么a一定不是celebrity，b可能是也可能不是。如果knows(a, b)返回false，那么b一定不是celebrity，a可能是可能不是。因此每调用一次knows可以排除一个人。所以我们连续调用n-1次，可以得到一个潜在的celebrity。最后还需要调用2n次验证是否这个潜在的celebrity不认识每个人且被每个人认识。

|  |
| --- |
| int findCelebrity(int n) {  int cand = 0, last\_cand = -1;  for (int k = 1; k < n; ++k) {  bool ans = knows(cand, k);  last\_cand = ans? cand : last\_cand;  cand = ans? k : cand;  }  for (int k = 0; k < cand; ++k) //have asked if cand knows those after cand  if (knows(cand, k)) return -1;    for (int k = 0; k < n; ++k)  if (last\_cand != k && !knows(k, cand)) return -1;  return cand;  } |

# 278. Find Bad Version

|  |
| --- |
| You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.  Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.  You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API. |

**Solution: binary search**

用二分法找到第一个bad: 循环不变量，第一个bad在[first, last]中。

(1)如果mid为good，则left=mid+1

(3)如果mid为bad，则right = mid (因为mid有可能是第一个bad)

|  |
| --- |
| int firstBadVersion(int n) {  int left = 1, right = n;  while (left < right) {  int mid = left+(right-left)/2; //avoid overflow  if (isBadVersion(mid)) right = mid;  else left = mid+1;  }  return left;  } |

# 279. Perfect Squares

|  |
| --- |
| Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n. |
| For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n = 13, return 2 because 13 = 4 + 9. |

**Solution1: DP**

|  |
| --- |
| int numSquares(int n) {  if (n <= 0) return 0;  vector<int> ns(n, INT\_MAX);  for (int k = 1; k <= n/k; ++k) ns[k\*k-1] = 1;    for (int j = 1; j <= n; ++j) {  if (ns[j-1] == INT\_MAX) { //complete square  for (int l = 1; l <= j/l; ++l) ns[j-1] = min(ns[j-1], ns[l\*l-1] + ns[j-l\*l-1]);  }  }    return ns.back();  } |

**Solution2: Memorized Recursion**

|  |
| --- |
| int numSquares(int n) {  if (n <= 0) return 0;  vector<int> ns(n+1, INT\_MAX);  return numSquares\_recur(n, ns);  }  int numSquares\_recur(int n, vector<int> &ns) {  if (n == 0) return 0;  if (ns[n] < INT\_MAX) return ns[n];  int min\_ns = INT\_MAX;  for (int j = 1; j <= n/j; ++j)  min\_ns = min(min\_ns, 1 + numSquares\_recur(n - j\*j, ns));  ns[n] = min\_ns;  return min\_ns;  } |

**Solution3: Static memorized recursion**

|  |
| --- |
| int numSquares(int n) {  if (n <= 0) return 0;  static vector<int> ns({0});  if (n+1 > ns.size()) ns.insert(ns.end(), n+1-ns.size(), INT\_MAX);    return numSquares\_recur(n, ns);  }  int numSquares\_recur(int n, vector<int> &ns) {  if (ns[n] < INT\_MAX) return ns[n];  int min\_ns = INT\_MAX;  for (int j = 1; j <= n/j; ++j)  min\_ns = min(min\_ns, 1 + numSquares\_recur(n - j\*j, ns));  ns[n] = min\_ns;  return min\_ns;  } |

**Solution4: number theory**

这在interview时应该不太可能想到

依赖两个数学定理：

Lagrange's Four Square Theorem：解只有可能是1,2,3,4

Legendre's Three Square Theorem: 只有在n = 4k (8m+7)时解才可能为4

因此我们分别检查是否解为1,2,4，否则为3

|  |
| --- |
| int numSquares(int n) {  if (n <= 0) return 0;  if (is\_square(n)) return 1;  if (is\_sum\_of\_four\_squares(n)) return 4;  if (is\_sum\_of\_two\_squares(n)) return 2;  return 3;  }  inline bool is\_square(int n) {  int sqr = sqrt(n);  return n== sqr\*sqr;  }  inline bool is\_sum\_of\_four\_squares(int n) {  while (n % 4 == 0) n/=4;  return n % 8 == 7;  }  inline bool is\_sum\_of\_two\_squares(int n) {  for (int j = 1; j < n/j; ++j) {  if (is\_square(n-j\*j)) return true;  }  return false;  } |

# 280. Wiggle Sort [Locked]

|  |
| --- |
| Given an unsorted array nums, reorder it in-place such that nums[0] <= nums[1] >= nums[2] <= nums[3]....  For example, given nums = [3, 5, 2, 1, 6, 4], one possible answer is [1, 6, 2, 5, 3, 4]. |

**Sol: observe & prove**

把数组元素两两组合。如果第2k个元素比第2k+1个元素大，则调转它们的顺序。

此时如果第2k+1个元素比第2k+2个元素小，则调换这两个元素。

|  |
| --- |
| void wiggleSort(vector<int>& nums) {  for (int k = 0; k+1 < nums.size(); k +=2)  if (nums[k] > nums[k+1]) swap(nums[k], nums[k+1]);  for (int k = 1; k+1 < nums.size(); k +=2)  if (nums[k] < nums[k+1]) swap(nums[k], nums[k+1]);  } |

# 281. ZigZag Iterator [Locked]

|  |
| --- |
| Given two 1d vectors, implement an iterator to return their elements alternately.  For example, given two 1d vectors:  v1 = [1, 2]  v2 = [3, 4, 5, 6]  By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1, 3, 2, 4, 5, 6].  Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?  The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic".  For example, given the following input:  v1 = [1,2,3]  v2 = [4,5,6,7]  v3 = [8,9]  It should return [1,4,8,2,5,9,3,6,7]. |

**Solution: queue+iterator**

把所有vector的iterator扔进一个队列。每次从队列取出一个vector取当前iterator指向的元素，并令iterator指向下一个元素。同时把当前iterator转移到队尾。如果iterator已经指向最后一个元素，则不再插入队列。直到队列为空。

|  |
| --- |
| class ZigzagIterator {  public:  ZigzagIterator(vector<int>& v1, vector<int>& v2) {  if (v1.begin() != v1.end()) q.emplace(v1.begin(), v1.end());  if (v2.begin() != v2.end()) q.emplace(v2.begin(), v2.end());  }  int next() {  auto it = q.front().first;  auto end\_it = q.front().second;  q.pop();  if (it+1 != end\_it) q.emplace(it+1, end\_it);  return \*it;  }  bool hasNext() {  return !q.empty();  }  private:  queue<pair<vector<int>::iterator, vector<int>::const\_iterator>> q;  }; |

# 282. Expression Add Operators

|  |
| --- |
| Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or \* between the digits so they evaluate to the target value.  Examples:  "123", 6 -> ["1+2+3", "1\*2\*3"]  "232", 8 -> ["2\*3+2", "2+3\*2"]  "105", 5 -> ["1\*0+5","10-5"]  "00", 0 -> ["0+0", "0-0", "0\*0"]  "3456237490", 9191 -> [] |

**Solution: recursion**

递归取出当前串的前k个数字加入表达式，并同时计算表达式的值。如果当前表达式为空，则加入数字后表达式值已得到，否则考虑在数字前加入+，-，\*的三种可能性。其中对\*的可能性，还需要知道乘法项\*之前的evaluation，把它从当前expression中扣除。

|  |
| --- |
| vector<string> addOperators(string num, int target) {  vector<string> result;  string prefix("");  addOperators\_recur(num, target, result, 0, prefix, 0);  return result;  }    void addOperators\_recur(const string &num, long target, vector<string> &result,  long first, string &prefix, long term) {  if (first == num.size()) {  if (target == 0) result.push\_back(prefix);  return;  }    int len\_prefix = prefix.size();  for (int k = first+1; k <= num.size(); ++k) {  if (num[first] == '0' && k>first+1) break;  string num\_str = num.substr(first, k-first);  long val = atol(num\_str.c\_str());    if (first == 0) {  prefix += num\_str;  addOperators\_recur(num, target - val, result, k, prefix, val);  }  else {  prefix += '+'; prefix += num\_str;  addOperators\_recur(num, target - val, result, k, prefix, val);  prefix[len\_prefix] = '-';  addOperators\_recur(num, target + val, result, k, prefix, -val);  prefix[len\_prefix] = '\*';  addOperators\_recur(num, target + term - val\*term, result, k, prefix, term\*val);  }  prefix.resize(len\_prefix);  }  } |

**Note:**Leetcode上最快的solution<20ms，但trick在于把string用char[]代替。上面的impl是C++的，运行时间112ms，beats 96.88%。

# 283. Move Zeros

|  |
| --- |
| Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.  Note:  You must do this in-place without making a copy of the array.  Minimize the total number of operations. |
| For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0]. |

**Solution: read-write pointer**

用一对读写指针即可。

|  |
| --- |
| void moveZeroes(vector<int>& nums) {  int iw = 0;  for (int ir = 0; ir < nums.size(); ++ir)  if (nums[ir] != 0) nums[iw++] = nums[ir];  for (; iw < nums.size(); ++iw) nums[iw] = 0;  } |

# 284. Peeking Iterator

|  |  |
| --- | --- |
| Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek() operation -- it essentially peek() at the element that will be returned by the next call to next().  Below is the interface for Iterator, which is already defined for you.   |  | | --- | | class Iterator {  struct Data;  Data\* data;  public:  Iterator(const vector<int>& nums);  Iterator(const Iterator& iter);  virtual ~Iterator();  int next(); // Returns the next element in the iteration.  bool hasNext() const; // Returns true if the iteration has more elements.  }; |   Follow up: How would you extend your design to be generic and work with all types, not just integer? |
| Example:  Assume that the iterator is initialized to the beginning of the list: [1, 2, 3].  Call next() gets you 1, the first element in the list.  Now you call peek() and it returns 2, the next element. Calling next() after that still return 2.  You call next() the final time and it returns 3, the last element. Calling hasNext() after that should return false. |

**Solution1：prefetch**

维护next()的值，用一个bool标记next()是否存在。

|  |
| --- |
| **class PeekingIterator : public Iterator** {  public:  **PeekingIterator(const vector<int>& nums) : Iterator(nums)** {  if (Iterator::hasNext()) {  mLast = false;  mNext = Iterator::next();  }  else mLast = true;  }  **int peek()** {  assert(!mLast); return mNext;  }  **int next()** {  int retval = mNext;  if (Iterator::hasNext()) mNext = Iterator::next(); else mLast = true;  return retval;  }  **bool hasNext() const** {  return !mLast;  }    private:  int mNext;  bool mLast;  }; |

**Solution2: copy constructor**

在Peek函数中获得一个对iterator的copy，并取next()。其余函数使用父类实现即可。

|  |
| --- |
| **class PeekingIterator : public Iterator** {  public:  **PeekingIterator(const vector<int>& nums)** : Iterator(nums) {}  **int peek()** {  return Iterator(\*this).next();  }  // do not define next() and hasNext() here so that they inherit the behavior of Iterator  }; |

# 285. Inorder Successor in BST [Locked]

|  |
| --- |
| Given a binary search tree and a node in it, find the in-order successor of that node in the BST.  Note: If the given node has no in-order successor in the tree, return null. |

**Solution: in-order traversal**

如果当前node有右子树，返回当前node的右子树的最左根结点。否则从根结点出发寻找node，并维护路径上最后一个向左走的parent，返回此parent。

|  |
| --- |
| **TreeNode\* inorderSuccessor(TreeNode\* root, TreeNode\* p) {**  if (p->right != NULL) {  p = p->right;  while (p->left != NULL) p = p->left;  return p;  }  TreeNode \*candidate = NULL;  while (root !=p)  root = (p->val > root->val)? root->right : (candidate = root)->left;  return candidate;  } |

# 286. Walls and Gates [Locked]

|  |
| --- |
| You are given a m x n 2D grid initialized with these three possible values.  -1 - A wall or an obstacle.  0 - A gate.  INF - Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.  Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF. |

**Solution: BFS**

|  |
| --- |
| **void wallsAndGates(vector<vector<int>>& rooms) {**  if (rooms.size() == 0 || rooms[0].size() == 0) return;  const int M = rooms.size(), N = rooms[0].size();  vector<int> pre;  for (int i = 0; i < M; ++i)  for (int j = 0; j < N; ++j)  if (rooms[i][j]==0) pre.push\_back(i\*N+j);  for (int l = 1; !pre.empty(); ++l) {  vector<int> cur;  for (int idx : pre) {  int i = idx/N, j = idx%N;  expand(rooms, cur, i-1, j, l);  expand(rooms, cur, i+1, j, l);  expand(rooms, cur, i, j-1, l);  expand(rooms, cur, i, j+1, l);  }  swap(pre, cur);  }  }  inline void expand(vector<vector<int>>& rooms, vector<int>& cur, int i, int j, int l) {  const int M = rooms.size(), N = rooms[0].size();  if (i < 0 || j < 0 || i >= M || j >= N || rooms[i][j] != INT\_MAX) return;  rooms[i][j] = l;  cur.push\_back(i\*N+j);  } |

# 287. Find the Duplicate Number

|  |
| --- |
| Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.  Note:  You must not modify the array (assume the array is read only).  You must use only constant, O(1) extra space.  Your runtime complexity should be less than O(n2).  There is only one duplicate number in the array, but it could be repeated more than once. |

**Solution: binary search**

考虑范围[first, last]中的数。因为duplicate number的数量多于missing number的数量，因此如果duplicate number在其中，则在这个范围中的数多于last-first+1个。因此我们可以用二分搜索，复杂度为O(nlogn)

|  |
| --- |
| **int findDuplicate(vector<int>& nums)** {  int first = 1, last = nums.size(); //1 and n+1  while (first < last-1) {  int mid = (first + last)/2;  int c1 = 0, c2 = 0; //goal: mid-first, last - mid  for (int v : nums) {  if (v < first || v >= last) continue;  if (v < mid) c1++; else c2++;  }  if (c1 > mid-first) last = mid;  else first = mid;  }  return first;  } |

# 288. Unique Word Abbreviation [Locked]

|  |
| --- |
| An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:  a) it --> it (no abbreviation)  b) d|o|g --> d1g  c) i|nternationalizatio|n --> i18n  d) l|ocalizatio|n --> l10n  Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation. |
| Example:  Given dictionary = [ "deer", "door", "cake", "card" ]  isUnique("dear") -> false  isUnique("cart") -> true  isUnique("cane") -> false  isUnique("make") -> true |

**Solution: unordered\_map**

用一个unordered\_map维护缩写对应的单词信息，包括：

1. 单词是否只出现一次
2. 第一个缩写为abbr的单词

边界条件：

(1)dictionary里有重复单词时，删除之

(2)isUnique检索的单词可能不存在于dictionary中

|  |
| --- |
| class ValidWordAbbr {  public:  ValidWordAbbr(vector<string> &dictionary) {  for (string s : dictionary) {  string abbr = getAbbr(s);  auto it = unique.find(abbr);  if (it == unique.end()) unique[abbr] = make\_pair(true, s);  else {  if (s.compare(it->second.second) != 0) it->second.first = false;  }  }  }  bool isUnique(string word) {  string abbr = getAbbr(word);  auto it = unique.find(abbr);  if (it == unique.end()) return true;  if (it->second.first == false) return false;  return !word.compare(it->second.second);  }  private:  string getAbbr(string &s) {  if (s.size()<3) return s;  else return s.front() + to\_string(static\_cast<int>(s.size())-2) + s.back();  }  unordered\_map<string, pair<bool, string>> unique;  }; |

# 289. Game of Life

|  |
| --- |
| According to the Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."  Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):  Any live cell with fewer than two live neighbors dies, as if caused by under-population.  Any live cell with two or three live neighbors lives on to the next generation.  Any live cell with more than three live neighbors dies, as if by over-population..  Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.  Write a function to compute the next state (after one update) of the board given its current state.  Follow up:  Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.  In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems? |

**Solution: bit mask**

用每个cell的第二位保存更新后的值。cell[i][j]在cell[i+1][j+1]计算后可以更新。cell[i][end]在cell[i+1][end]计算后可以更新。cell[end][i]在cell[end][i-1]计算后可以更新。

|  |
| --- |
| **void gameOfLife(vector<vector<int>>& board)** {  if (board.size() == 0 || board[0].size() == 0) return;    int next\_bit[18] = {0, 0, 0, 1, 0, 0, 0, 0, 0,  0, 0, 1, 1, 0, 0, 0, 0, 0};  int di[8] = {-1, -1, -1, 0, 0, 1, 1, 1};  int dj[8] = {-1, 0, 1, -1, 1, -1, 0, 1};    for (int i = 0; i < board.size(); ++i) {  for (int j = 0; j < board[0].size(); ++j) {  int count = 0;  for (int k = 0; k < 8; ++k) {  int i1 = i + di[k], j1 = j + dj[k];  if (i1 < 0 || j1 < 0 || (i1 >= board.size()) || (j1 >= board[0].size()))  continue;  if (board[i1][j1]%2 == 1) count++;  }  board[i][j] += next\_bit[board[i][j] \* 9 + count] \* 2;  if (i > 0) board[i-1][j-1] /=2;  if (i == board.size()-1) board[i][j-1] /= 2;  }  if (i > 0) board[i-1].back()/=2;  if (i == board.size()-1) board[i].back()/=2;  }  } |

# 

# 

# 290. Word Pattern

|  |
| --- |
| Given a pattern and a string str, find if str follows the same pattern.  Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.  Examples:  pattern = "abba", str = "dog cat cat dog" should return true.  pattern = "abba", str = "dog cat cat fish" should return false.  pattern = "aaaa", str = "dog cat cat dog" should return false.  pattern = "abba", str = "dog dog dog dog" should return false.  Notes:  You may assume pattern contains only lowercase letters, and str contains lowercase letters separated by a single space. |

**Solution: unordered\_set**

用一个array of string存储字母到字符串的映射。用一个unordered\_set存储已经出现过的单词。

|  |
| --- |
| **bool wordPattern(string pattern, string str)** {  istringstream iss(str);  string map[256] = {""};  unordered\_set<string> history;  for (char c : pattern) {  string word;  if (!(iss >> word)) return false;  if (map[c].size() == 0) {//new  if (history.insert(word).second == false) return false;  map[c] = word;  }  else if (word.compare(map[c]) != 0) return false;  }    return (iss.eof());  } |

# 291. Word Pattern II [Locked]

|  |
| --- |
| Given a pattern and a string str, find if str follows the same pattern.  Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty substring in str.  Notes: You may assume both pattern and str contains only lowercase letters. |
| Examples:  pattern = "abab", str = "redblueredblue" should return true.  pattern = "aaaa", str = "asdasdasdasd" should return true.  pattern = "aabb", str = "xyzabcxzyabc" should return false. |

**Solution: recursion + prunning + hashing**

递归结构：R(p\_first, s\_first, cmap, words)

p\_first, s\_first为当前等待parse的pattern和str中的第一个字符

cmap, words为pattern[0...p\_first)和str[0...s\_first)之间构建的一个mapping

R为是否存在一个和cmap, words consistent的mapping

递推关系：

1. 如果pattern[p\_first]=c在cmap中已经出现过，且cmap[c]不是str[s\_first…]的前缀，则返回false
2. 如果pattern[p\_first]=c在cmap中已经出现过，且cmap[c]是str[s\_first…]的前缀，则返回R(p\_first+1, s\_first+len, cmap, words)，其中len是cmap[c]的长度
3. 如果pattern[p\_first]=c在cmap中没有出现过，则穷举str[s\_first, …]中没有在word中出现的前缀，作为c的mapping。只要其中有一个满足R(p\_first+1, s\_first+prefix.len, cmap U (c, prefix), wordsUprefix)=true，就返回true

结束条件：p\_first==pattern.size()，返回是否str已扫完：s\_first==str.size()

优化：

1. 扁平化for loop的第二种可能性。因为只有一个branch，可以用for loop解决
2. pattern剩下长度必须多于str剩下的长度(如果记录pattern的histogram可以进一步优化，但实现上过于复杂）
3. 用string[]而不是unorderted\_map<char, string>存储char到string的映射
4. 用str.compare(pos, len, word)来进行子串比较，从而避免用substr操作创建新串

|  |
| --- |
| **bool wordPatternMatch(string pattern, string str)** {  string cmap[256]; //map from pattern char to string  unordered\_set<string> words; //set of strings that already been mapped  return wordPatternMatch\_recur(pattern, str, 0, 0, cmap, words);  }  bool wordPatternMatch\_recur(const string& pattern, const string& str,  int p\_first, int s\_first,  string cmap[],  unordered\_set<string>& words) {  //skip (char, word) pairs that already has a mapping  for ( ; p\_first < pattern.size(); ++p\_first) {  string& word = cmap[pattern[p\_first]];  if (word.size()>0) {  int word\_len = word.size();  if (s\_first + word\_len > str.size() ||  str.compare(s\_first, word\_len, word) != 0) return false;  s\_first += word\_len;  }  else break; //key is new  }  if (p\_first == pattern.size()) return s\_first == str.size(); //finish scanning pattern    //char in pattern is new: try different possibilities of mapped substring  //remaining substr of str should be longer than pattern  int max\_len = int(str.size()) - int(pattern.size()) + p\_first + 1 - s\_first;  for (int word\_len = 1; word\_len <= max\_len; ++word\_len) {  string word = str.substr(s\_first, word\_len);  if (words.find(word) != words.end()) continue; //word already mapped, skip  cmap[pattern[p\_first]] = word;  words.insert(word);  if (wordPatternMatch\_recur(pattern, str, p\_first+1, s\_first+word\_len, cmap, words))  return true;  cmap[pattern[p\_first]] = "";  words.erase(word);  }  return false;  } |

# 292. Nim Game

|  |
| --- |
| You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.  Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.  For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend. |

**Solution: Mathematical reduction**

如果桌上的石头有1－3块，则已经赢了。如果桌上有4块石头，则余下1－3块，必输。

若桌上有5-7块石头，可以留下4块，对手必输。如果桌上有8块，则轮到对手时桌上有5-7块石头，自己必输。

推广上述观察：石头是4的倍数时会输。

若桌上有4k+m块石头(m>0),则可以取走m块石头，这样对手取走1－3块后，仍剩下4(k-1)+n块石头(n>0)。若桌上有4k块石头，则取走1－3块后对手剩下4(k+1)+m(m>0)块石头，因此对手必胜。

|  |
| --- |
| **bool canWinNim(int n)** {  return n % 4;  } |

# 293. Flip Game [Locked]

|  |
| --- |
| You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.  Write a function to compute all possible states of the string after one valid move.  For example, given s = "++++", after one move, it may become one of the following states:  [  "--++",  "+--+",  "++--"  ]  If there is no valid move, return an empty list []. |

**Solution:iteration**

从左到右扫一遍，并检查是否右两个连续的++。

|  |
| --- |
| **vector<string> generatePossibleNextMoves(string s)** {  vector<string> result;  for (int k = 0; k + 1 < s.size(); ++k) {  if (s[k] == '+' && s[k+1] == '+') {  result.push\_back(s);  result.back()[k] = result.back()[k+1] = '-';  }  }  return result;  } |

# 294. Flip Game II [Locked]

|  |
| --- |
| You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.  Write a function to determine if the starting player can guarantee a win.  For example, given s = "++++", return true. The starting player can guarantee a win by flipping the middle "++" to become "+--+".  Follow up:  Derive your algorithm's runtime complexity. |

**Solution1: memorized recursion**

|  |
| --- |
| **bool canWin(string s)** {  unordered\_map<string, bool> cache;  return canWin\_recur(s, cache);  }  bool canWin\_recur(string& s, unordered\_map<string, bool>& cache) {  auto it = cache.find(s);  if (it != cache.end()) return it->second;  for (int k = 0; k+1 < s.size(); ++k) {  if (s[k] == '+' && s[k+1] == '+') {  s[k] = s[k+1] = '-';  if (!canWin(s)) { cache[s] = true; return true;}  s[k] = s[k+1] = '+';  }  }  cache[s] = false;  return false;  } |

**Solution2: memorized recursion + bit tricks**

|  |
| --- |
| **bool canWin(string s)** {  assert(s.size() < 64);  uint64\_t code = 0;  for (char c : s) code = (code<<1) | (c == '+'?1:0);  unordered\_map<uint64\_t, bool> cache;  return canWin\_recur(code, cache);  }  bool canWin\_recur(uint64\_t code, unordered\_map<uint64\_t, bool>& cache) {  auto it = cache.find(code);  if (it != cache.end()) return it->second;  for (uint64\_t left = code; left; left = left & (left-1)) {  uint64\_t digit = left & -left;  digit = digit | (digit << 1);  if ((code & digit)==digit && !canWin\_recur(code^digit, cache)) {  return cache[code]=true;  }  }  return cache[code] = false;  } |

# 295. Find Median from Data Stream

|  |
| --- |
| Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.  E.g. the median of [2,3,4] is 3, the median of [2,3] is (2 + 3) / 2 = 2.5.  Design a data structure that supports the following two operations:  void addNum(int num) - Add a integer number from the data stream to the data structure.  double findMedian() - Return the median of all elements so far. |
| For example:  add(1)  add(2)  findMedian() -> 1.5  add(3)  findMedian() -> 2 |

**Solution: heap**

维护两个heap，heap1维护较大的一半数，heap2维护较小的一半数。保证heap1中的数比heap2中的多0－1个。这样取中位数只用取heap1的最小数，以及总数为奇数时取heap2的最大数。进来一个数时，如果它小于等于当前中位数，则进heap2，否则进heap1。如果heap1中的数太多，则把最小数pop放进heap2，同理当heap2数太多则pop最大数放进heap1.

|  |
| --- |
| class MedianFinder {  public:  // Adds a number into the data structure.  void addNum(int num) {  if (max\_heap.size() == 0) min\_heap.push(num);  else {//min\_heap and max\_heap are both non-empty  int vmin = min\_heap.top(), vmax = max\_heap.top();  if (num > vmax) max\_heap.push(num);  else min\_heap.push(num);  }    if (min\_heap.size() - max\_heap.size() == 2) {  //move max of min\_heap to max\_heap  int vmin = min\_heap.top(); min\_heap.pop();  max\_heap.push(vmin);  }  else if (min\_heap.size() - max\_heap.size() == -1) {  //move min of max\_heap to min\_heap  int vmax = max\_heap.top(); max\_heap.pop();  min\_heap.push(vmax);  }  }  // Returns the median of current data stream  double findMedian() {  int vmin = min\_heap.top();  if ((min\_heap.size() + max\_heap.size()) % 2 == 1) return min\_heap.top();  else return (min\_heap.top() + max\_heap.top())/2.0;  }    private:  priority\_queue<int,vector<int>,less<int> > min\_heap;  priority\_queue<int,vector<int>,greater<int> > max\_heap;  }; |

# 296. Best Meeting Point [Locked]

|  |
| --- |
| A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.  For example, given three people living at (0,0), (0,4), and (2,2):  1 - 0 - 0 - 0 - 1  | | | | |  0 - 0 - 0 - 0 - 0  | | | | |  0 - 0 - 1 - 0 - 0  The point (0,2) is an ideal meeting point, as the total travel distance of 2+2+2=6 is minimal. So return 6. |

**Solution: 2D-to-1D separation + linear programming**

最优位置的选择x和y是分开的。

所以把问题简化为在1d空间寻找最优的meeting point。通过写出并展开距离的表达式，我们可以发现距离先单调减再单调增，最小值位置在所有起点的中位数位置。

|  |
| --- |
| int minTotalDistance(vector<vector<int>>& grid) {  vector<int> x, y;  for (int i = 0; i < grid.size(); ++i) {  for (int j = 0; j < grid[0].size(); ++j) {  if (grid[i][j]==0) continue;  x.push\_back(j); y.push\_back(i);  }  }  nth\_element(x.begin(), x.begin()+y.size()/2, x.end());  int mx = x[x.size()/2], my = y[y.size()/2];  int dist = 0;  for (int v : x) dist += abs(mx - v);  for (int v : y) dist += abs(my - v);  return dist;  } |

# 297. Serialize and Deserialize Binary Tree

|  |
| --- |
| Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.  Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure. |

**Solution: pre-order traversal**

遇到NULL时输出#，用空格作为分隔符

|  |
| --- |
| class Codec {  public:  // Encodes a tree to a single string: nodes are separated by space, and '#' denotes NULL  string serialize(TreeNode\* root) {  string result("");  serialize\_recur(root, result);  return result;  }    // Decodes your encoded data to tree.  TreeNode\* deserialize(string data) {  istringstream iss(data);  return deserialize\_recur(iss);  }    private:  void serialize\_recur(TreeNode \*root, string &s) {  if (s.size()) s += ' ';  if (root) s += to\_string(root->val);  else {s += '#'; return;}    serialize\_recur(root->left, s);  serialize\_recur(root->right, s);  }    TreeNode\* deserialize\_recur(istringstream &iss) {  string word;  iss >> word;  if (word[0] == '#') return NULL;    TreeNode \*root = new TreeNode(atoi(word.c\_str()));  root->left = deserialize\_recur(iss);  root->right = deserialize\_recur(iss);  return root;  }  }; |

# 298. Binary Tree Longest Consecutive Sequence [Locked]

|  |
| --- |
| Given a binary tree, find the length of the longest consecutive sequence path.  The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse). |
| For example,  1 2  \ \  3 3  / \ /  2 4 2  \ /  5 1  Left: Longest consecutive sequence path is 3-4-5, so return 3.  Right: Longest consecutive sequence path is 2-3,not3-2-1, so return 2. |

**Solution: post-order traversal**

对每个node，计算：

1. 以这个node为起点的最长连续sequence
2. node所在子树的最长sequence

|  |
| --- |
| int longestConsecutive(TreeNode\* root) {  return longestConsecutive\_recur(root).second;  }  pair<int, int> longestConsecutive\_recur(TreeNode\* root) {  if (root == NULL) return make\_pair(0, 0);  pair<int, int> lp = longestConsecutive\_recur(root->left);  pair<int, int> rp = longestConsecutive\_recur(root->right);  int l = 1;  if (root->left && root->left->val == root->val+1) l = lp.first + 1;  if (root->right && root->right->val == root->val+1) l = max(l, rp.first + 1);  return make\_pair(l, max(l, max(lp.second, rp.second)));  } |

# 299. Bulls and Cows

|  |
| --- |
| You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number. |
| For example:  Secret number: "1807"  Friend's guess: "7810"  Hint: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.)  Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows. In the above example, your function should return "1A3B".  Please note that both secret number and friend's guess may contain duplicate digits, for example:  Secret number: "1123"  Friend's guess: "0111"  In this case, the 1st 1 in friend's guess is a bull, the 2nd or 3rd 1 is a cow, and your function should return "1A1B".  You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal. |

**Solution: histogram**

用一个histogram记录每个值的count，bull不记入histogram，但是记入bull对应的计数器。最后histogram的和是未匹配的数字。字符串总数-bull个数-histogram之和因此是cow个数。

|  |
| --- |
| **string getHint(string secret, string guess)** {  int na = 0;  vector<int> cnt(10, 0);  for (int k = 0; k < secret.size(); ++k) {  if (secret[k] == guess[k]) na++;  else {  cnt[secret[k] - '0']++;  cnt[guess[k] - '0']--;  }  }    int nb = secret.size() - na;  for (int k = 0; k <= 9; ++k) nb -= max(cnt[k], 0);  return to\_string(na) + 'A' + to\_string(nb) + 'B';  } |

# 300. Longest Increasing Subsequences

|  |
| --- |
| Given an unsorted array of integers, find the length of longest increasing subsequence.  For example,  Given [10, 9, 2, 5, 3, 7, 101, 18],  The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note that there may be more than one LIS combination, it is only necessary for you to return the length.  Your algorithm should run in O(n2) complexity.  Follow up: Could you improve it to O(n log n) time complexity? |

**Solution1:DP**

从右到左计算数组每个右缀的LIS长度。对第k个元素，检查所有可能append在它之后的子序列。

|  |
| --- |
| **int lengthOfLIS(vector<int>& nums)** {  if (nums.size() == 0) return 0;    int maxlen = 1;  vector<int> len(nums.size(), 1);  for (int k = nums.size()-2; k >= 0; --k) {  for (int j = k+1; j < nums.size(); ++j) {  if (nums[j] > nums[k]) len[k] = max(len[k], len[j]+1);  }  maxlen = max(maxlen, len[k]);  }  return maxlen;  } |

**Solution2: iteration + binary search**

上述动态规划浪费指出在于len[1..k]可能寸了多个长度为l的串。其中只有尾数最小的串真正对更新len[k]起作用。因此我们改存前k个数字产生的所有长度为l的子串中尾数最小的，把它们的尾数纪录在数组min\_tail[]里。这样更新发生在min\_tail[l-1]<nums[k]<min\_tail[l]的情况。这提示我们考虑min\_tail是否可能是单调的（因为如果是则只发生一处更新，且可以二分搜到）。此时可以考虑长为l、尾数最小的LIS的倒数第二个数，它一定小于min\_tail[l]且大于min\_tail[l-1]，因此min\_tail是单调增的。

|  |
| --- |
| **int lengthOfLIS(vector<int>& nums)** {  vector<int> min\_tail;  for (int v : nums) {  auto it = lower\_bound(min\_tail.begin(), min\_tail.end(), v);  if (it == min\_tail.end()) min\_tail.push\_back(v);  else (\*it) = v;  }  return min\_tail.size();  } |

# 