# Shortlisted problems

301

302

307

315

316

324

332

338

# Index

|  |  |  |  |
| --- | --- | --- | --- |
| 301 | recursion; iteration+stack | medium | classic good example for prunning |
| 302L | bfs/dfs; binary search | interesting | classic |
| 303 | dp | trivial |  |
| 304 | dp | trivial |  |
| 305L | union-find | easy | has difficult followup |
| 306 | enumeration | trivial |  |
| 307 | interval tree | difficult | classic |
| 308L | interval tree | difficult | untested |
| 309 | dp | easy |  |
| 310 | recursion | trivial |  |
| 311L | sparse rep. + merge sort | trivial | untested |
| 312 | dp | easy |  |
| 313 | merge sort | medium | tricky impl. |
| 314L | recursion | easy | triky impl. |
| 315 | bst; merge sort; interval tree | difficult !! | classic & super tricky |
| 316 | iteration+stack | difficult | classic need better understanding |
| 317L | bfs | trivial | untested |
| 318 | bit tricks | trivial |  |
| 319 | brain teaser(parity) | easy |  |
| 320L | Recursion; dp | trivial |  |
| 321\*\* | iteration+stack | difficult | need better understanding |
| 322 | recursion | trivial |  |
| 323L | union-find | easy | untested |
| 324 | rewiring+partition | difficult !! | classic & super tricky |
| 325L | hash | trivial | untested |
| 326 | math trick | interesting |  |
| 327 | bst/merge sort | difficult !!! | super tricky |
| 328 | linked list | trivial |  |
| 329 | dfs | trivial |  |
| 330 | greedy | medium | Needs better understanding |
| 331 | graph theory | interesting |  |
| 332 | dfs | difficult | classic |
| 333L | recursion | trivial | untested |
| 334 | dp | trivial |  |
| 335\* | iteration | medium | A good stretch of mind |
| 336\* | trie | medium | Impl. tricky |
| 337 | recursion | easy |  |
| 338 | bit trick | interesting | classic |
| 339L | recursion | trivial | untested |
| 340L | sliding window | easy | untested |
| 341 | stack | trivial |  |
| 342 | math tricks | easy |  |
| 343 | dp | trivial |  |
| 344 | Two pointer | trivial |  |
| 345 | Two pointer | trivial |  |
| 346L | sliding window | trivial | untested |
| 347 | heap | trivial |  |
| 348L | bitset | trivial | untested |
| 349 | hash | trivial |  |
| 350 | hash | trivial |  |

# 301. Remove Invalid Parenthesis

|  |
| --- |
| Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.  Note: The input string may contain letters other than the parentheses ( and ). |
| Examples:  "" -> ["()()()", "(())()"]  "(a)())()" -> ["(a)()()", "(a())()"]  ")(" -> [""] |

**Solution1: DFS**

问题简化: 暂时忽略要求删除最少的要求，把问题看成是逐步创建一个合法串的过程。因此和enumerate 输入的子串是一样的，不合法的前缀会直接被剪枝。

合法前缀条件: (1) 左括号数>=右括号数。(2)后续右括号数>=当前左括号数-当前右括号数

递归状态：

* 输入串下一个read位置 first
* 当前串左右括号数 nl, nr
* 当前前缀prefix (用于最后输出结果）

初始状态：

* (first, nl, nr, prefix)=(0, 0, 0, "")

递归结构：(first, nl, nr, prefix)的下一个状态可能是

* s[first]=='(': (first+1, nl+1, nr, prefix+'(') 或 (first+1, nl, nr, prefix)
* s[first]==')': (first+1, nl, nr+1, prefix+')') 或 (first+1, nl, nr, prefix)
* s[first]为其它值：(first+1, nl, nr, prefix+s[first])

递归终止条件：

* first == s.size(): 输入串已扫完，如果当前nr=nl，则prefix是合法串，增加到结果
* nl < nr: 当前前缀不合法

例子：

Input: ()())

(

()

(

Input: **(**)())

Input: (**)**())

()(

()

Input: ()**(**))

((

(

Input: ()(**)**)

()()

()(

()

(()

((

()

(

Input: ()()**)**

()()

()()

()(

()

(())

(()

(()

((

()

()

(

(

()

(

()

()

(

最少删除：在DFS过程中，我们优先搜索那些不删除括号的路径，这样第一个产生的结果一定是最长的。

剪枝：找到第一个串后我们就知道了目标长度

* 假如当前可输入的字符数量不足以填满目标长度，可以直接返回。
* 假如当前可输入的字符数量多于目标长度，可直接返回。

避免重复串：比较简单的办法是用一个unordered\_set解决。

|  |
| --- |
| **vector<string> removeInvalidParentheses(string s)** {  unordered\_set<string> result;  removeInvalidParentheses\_recur(s, "", 0, 0, 0, result);  vector<string> vec\_res(result.begin(), result.end());  return vec\_res;  }  void removeInvalidParentheses\_recur(const string &s, string prefix, int first, int nl, int nr, unordered\_set<string> &result) {    // early stopping considering length of solution / invalid prefix  if (!result.empty() && prefix.size() > result.begin()->size()) return;  if (!result.empty() && prefix.size() + s.size() - first < result.begin()->size()) return;  if (nl < nr) return;    // add alphabetic characters until seeing end of string or a bracket  for (; first < s.size() && isalpha(s[first]); first++) prefix += s[first];    // reached end of string s  if (first == s.size()) {  if (nl == nr) result.insert(prefix);  return;  }    // recursion  if (s[first] == '(') removeInvalidParentheses\_recur(s, prefix+'(', first+1, nl+1, nr, result);  else removeInvalidParentheses\_recur(s, prefix+')', first+1, nl, nr+1, result);  removeInvalidParentheses\_recur(s, prefix, first+1, nl, nr, result);  } |

**Solution2: BFS**

在上述例子中我们发现每层有很多重复，通过BFS+unordered\_set可以有效避免这种重复。

我们可以先经过一轮bfs找到最大深度，从而对BFS剪枝。

Input: ()())()

(

()

(

Input: **(**)())()

Input: (**)**())()

()(

()

Input: ()**(**))()

((

Input: ()(**)**)()

()()

()(

(()

Input: ()()**)**()

()()

(())

()()(

()()()

Input: ()())(**)**

Input: ()())**()**

(())(

(())()

|  |
| --- |
| **vector<string> removeInvalidParentheses(string s)** {  int max\_len, cnt\_pair, cnt\_left, cnt\_right;  tie(max\_len, cnt\_pair, cnt\_left, cnt\_right) = compute\_max\_length(s);  if (max\_len == 0) return vector<string>{""};  unordered\_map<string, pair<int, int>> pre;  pre[""] = make\_pair(0, 0);  for (char c : s) {  unordered\_map<string, pair<int, int>> cur;  for (auto &item : pre) {  string cur\_str = item.first+c;  int nl = item.second.first, nr = item.second.second;  if (nl > cnt\_pair || cnt\_left + nl < cnt\_pair || cnt\_right + nr < cnt\_pair)  continue;  if (c == '(' || c == ')') cur.insert(item);  cur[cur\_str] = make\_pair(nl + (c=='('?1:0), nr + (c==')'&&nl>nr?1:0));  }  swap(cur, pre);  if (c =='(') --cnt\_left;  if (c ==')') --cnt\_right;  }  vector<string> result;  for (auto & item : pre) {  if (item.first.size() == max\_len && item.second.first == item.second.second)  result.push\_back(item.first);  }  return result;  }  tuple<int, int, int, int> compute\_max\_length(const string &s) {  stack<char> stk;  int count\_remove = 0, count\_pair = 0, count\_left = 0, count\_right = 0;  for (char c : s) {  if (c == '(') {  stk.push(c);  ++count\_left;  }  if (c == ')') {  ++count\_right;  if (stk.empty()) ++count\_remove; //extra parenthesis found  else { //parenthesis matched  stk.pop();  ++count\_pair;  }  }  }  count\_remove += stk.size();  return make\_tuple(s.size() - count\_remove, count\_pair, count\_left, count\_right);  } |

# 

# 

# 302. Smallest Rectangle Enclosing Black Pixels [Locked]

|  |
| --- |
| An image is represented by a binary matrix with 0 as a white pixel and 1 as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location (x, y) of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.  For example, given the following image:  [  "0010",  "0110",  "0100"  ]  and x = 0, y = 2,Return 6. |

**Solution 1：DFS**

Floodfill整个联通分支，并维护遍历过程中行和列的最大值和最小值。

|  |
| --- |
| int minArea(vector<vector<char>>& img, int row, int col) {  if (img.size() == 0 || img[0].size() == 0) return 0;  int minr = row, maxr = row, minc = col, maxc = col;  minArea\_recur(img, row, col, minr, maxr, minc, maxc);  return (maxr - minr + 1) \* (maxc - minc + 1);  }  inline void minArea\_recur(vector<vector<char>>& img, int r, int c,  int& minr, int& maxr, int& minc, int& maxc) {  if (c < 0 || r < 0 || c >= img[0].size() || r >= img.size() || img[r][c] != '1') return;  img[r][c] = ' ';  minr = min(minr, r); maxr = max(maxr, r); minc = min(minc, c); maxc = max(maxc, c);  minArea\_recur(img, r-1, c, minr, maxr, minc, maxc);  minArea\_recur(img, r+1, c, minr, maxr, minc, maxc);  minArea\_recur(img, r, c-1, minr, maxr, minc, maxc);  minArea\_recur(img, r, c+1, minr, maxr, minc, maxc);  } |

**Solution 2: Binary search**

如果我们对image作一维投影，该投影也是联通的。因此我们用二分法搜索左右上下边界（从(r, c)向4个方向散去第一个全0的行或列）。

|  |
| --- |
| inline bool check(vector<vector<char>>& img, int id, bool check\_row) {  if (check\_row) { for (char c: img[id]) if (c == '1') return true; }  else { for (vector<char> &v: img) if (v[id] == '1') return true; }  return false;  }  //b1: id of row/col which contains 1, b0: id of row/col which contains 0  int binary\_search(vector<vector<char>>& img, int b1, int b0, bool check\_row) {  while (abs(b1-b0) > 1) {  int mid = (b1+b0)/2;  if (check(img, mid, check\_row)) b1 = mid; else b0 = mid;  }  return b1;  }  int minArea(vector<vector<char>>& img, int row, int col) {  int minc = binary\_search(img, col, -1, false);  int minr = binary\_search(img, row, -1, true);  int maxc = binary\_search(img, col, img[0].size(), false);  int maxr = binary\_search(img, row, img.size(), true);  return (maxr - minr + 1) \* (maxc - minc + 1);  } |

# 303. Range Sum Query - Immutable

|  |
| --- |
| Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.  Note:  You may assume that the array does not change.  There are many calls to sumRange function. |
| Example:  Given nums = [-2, 0, 3, -5, 2, -1]  sumRange(0, 2) -> 1  sumRange(2, 5) -> -1  sumRange(0, 5) -> -3 |

**Solution: cumsum**

存储nums的所有前缀和sum[k+1] = nums[0]+...+nums[k]，则sumRange(i,j)=sum[j]-sum[i-1]

|  |
| --- |
| **class NumArray** {  public:  **NumArray(vector<int> &nums)** {  for (int k = 0; k < nums.size(); ++k)  sum.push\_back( (k==0? 0 : sum.back()) + nums[k] );  }  **int sumRange(int i, int j)** {  return sum[j] - (i==0? 0 : sum[i-1]);  }  vector<int> sum;  }; |

# 304. Range Sum Query 2D - Immutable

|  |
| --- |
| Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).  Note:  You may assume that the matrix does not change.  There are many calls to sumRegion function.  You may assume that row1 ≤ row2 and col1 ≤ col2. |

**Solution: cumsum**

在2D数组中存储矩阵左上角到每个点(i,j)的累加和.

则sumRegion(row1, col1, row2, col2) =

sum(row2, col2)+sum(row1-1,col1-1)-sum(row2, col1-1)-sum(row1-1, col2)

|  |
| --- |
| **class NumMatrix** {  public:  **NumMatrix(vector<vector<int>> &matrix)** {  for (int n = 0; n < matrix.size(); ++n) {  area.push\_back(vector<int>{});  for (int m = 0; m < matrix[0].size(); ++m) {  if (m == 0 && n == 0) area[n].push\_back(matrix[n][m]);  else if (n == 0) area[n].push\_back(area[n][m-1] + matrix[n][m]);  else if (m == 0) area[n].push\_back(area[n-1][m] + matrix[n][m]);  else area[n].push\_back(-area[n-1][m-1] + area[n][m-1]  + area[n-1][m] + matrix[n][m]);  }  }  }  **int sumRegion(int row1, int col1, int row2, int col2)** {  return get\_area(row1-1, col1-1) + get\_area(row2, col2)  - get\_area(row1-1, col2) - get\_area(row2, col1-1);  }    **inline int get\_area(int r, int c)** {  return (r < 0 || c < 0)? 0 : area[r][c];  }    vector<vector<int>> area;  }; |

# 305. Number of Islands II [Locked]

|  |
| --- |
| A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.  Example: Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].  Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).  Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.  Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.  Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.  Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.  0 0 0 1 0 0 1 1 0 1 1 0 1 1 0  0 0 0 0 0 0 0 0 0 0 0 1 0 0 1  0 0 0 0 0 0 0 0 0 0 0 0 0 1 0  We return the result as an array: [1, 1, 2, 3] |

**Solution: Union find**

|  |
| --- |
| class Islands {  public:  Islands(int m, int n) : board(m, vector<int>(n, 0)), nrow(m), ncol(n), prev(1, 0) {};    int add(int r, int c) {  if (board[r][c] != 0) return nislands;    unordered\_set<int> roots;  int min\_id = prev.size();  for (int k = 0; k < 4; ++k) {  int id = get\_id(r+shifts[k].first, c+shifts[k].second);  if (id > 0) {  roots.insert(id);  min\_id = min(id, min\_id);  }  }  if (!roots.empty()) for (int v : roots) prev[v] = min\_id;  else prev.push\_back(min\_id);  board[r][c] = min\_id;  nislands -= (roots.size()-1);  return nislands;  }    private:  int get\_id(int r, int c) {  if (r < 0 || c < 0 || r >= nrow || c >= ncol) return 0;  int id = board[r][c];  while (prev[id] != id) id = prev[id];  return id;  }  vector<vector<int>> board;  vector<int> prev;  int nrow, ncol;  int nislands = 0;  const vector<pair<int, int>> shifts = {{0, 1}, {0, -1}, {-1, 0}, {1, 0}};  };  **vector<int> numIslands2(int m, int n, vector<pair<int, int>>& positions)** {  if ( m == 0 || n == 0) return vector<int>{};  Islands cnt(m, n);    vector<int> result(positions.size(), 0);  for (int k = 0; k < positions.size(); ++k)  result[k] = cnt.add(positions[k].first, positions[k].second);  return result;  } |

# 306. Additive Number

|  |
| --- |
| Additive number is a string whose digits can form additive sequence.  A valid additive sequence:   * should contain at least three numbers. * Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two. * Numbers in the additive sequence cannot have leading zeros, so sequence 1, 2, 03 or 1, 02, 3 is invalid.   Given a string containing only digits '0'-'9', write a function to determine if it's an additive number. Method should be able to handle overflow.  Example: "112358" is an additive number because the digits can form an additive sequence: 1, 1, 2, 3, 5, 8. |

**Solution: iterative enumertion**

算法：穷举前两个数，验证后序的字符是否和计算得到的和一致。

边界条件：

* skip 0开始的数
* 字符串可能很长，因此不能用加法，而要自己实现字符串上的加法，注意可能带来进位

|  |
| --- |
| **bool isAdditiveNumber(string num)** {  if (num.size() <=2) return false;  for (int k = 1; k < num.size()-1; ++k) {  if (num[0] == '0' && k > 1) return false;  for (int j = k+1; j < num.size(); ++j) {  if (num[k] == '0' && j>k+1) break;  if (check(num, k, j)) return true;  }  }  return false;  }  inline bool check(const string &num, int k, int j) {  string pre = num.substr(0, k), last = num.substr(k, j - k);  for (int l = j; l < num.size(); l += last.size()) {  string next = addString(pre, last);  if (l+next.size() > num.size() || num.substr(l, next.size()).compare(next) != 0)  return false;  pre = last; last = next;  }  return true;  }  inline string addString(string &s1, string &s2) {  string result("");  int overflow = 0;  auto it1 = s1.rbegin(), it2 = s2.rbegin();  while (it1 != s1.rend() || it2 !=s2.rend()) {  int sum = overflow + (it1 == s1.rend()? 0: (\*it1 - '0'))  + (it2 == s2.rend()? 0: (\*it2 - '0'));  result += ('0' + sum%10);  overflow = sum/10;  if (it1 != s1.rend()) it1++;  if (it2 != s2.rend()) it2++;  }  if (overflow) result += '1';  reverse(result.begin(), result.end());  return result;  } |

# 307. Range Sum Query - Mutable

|  |
| --- |
| Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.  The update(i, val) function modifies nums by updating the element at index i to val.  Example: Given nums = [1, 3, 5]  sumRange(0, 2) -> 9  update(1, 2)  sumRange(0, 2) -> 8  Note:  The array is only modifiable by the update function.  You may assume the number of calls to update and sumRange function is distributed evenly. |

**Solution1: Fenwick Tree (Binary Index Tree).**

设计思想：通过把cumsum的值分散保存在idx在树中的路径上，并用位运算映射树结构，使检索更新cumsum快速简单

例子：下图圆圈中蓝字为下标idx，黑字为值tree[idx]。红线指向后继next，黑线指向parent.

结点关系映射：

第k层的index有k位1.

黑线－从子结点到父结点index的映射: **parent(n) = n & (n-1)**，即最后一位置0.

红线－从结点到它后继结点的映射： **next(n) = n + (n & -n)**. 左移最后一位1,或把最后一段0111...11结构变为1000...00

例子：

**idx** 0 1 2 3 4 5 6 7 8 9 10 11 12

(binary) 0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100

parent[idx] - 0 0 2 0 4 4 6 0 8 8 10 8

(binary) - 0 0 10 0 100 100 110 0 1000 1000 1010 1000

next[idx] - 2 4 4 8 6 8 8 - 10 12 12 -

(binary) - 10 100 100 1000 1q0 1000 1000 - 1010 1100 1100 -

对后继映射的进一步解释：

后继结点定义为下一个兄弟结点，或如果当前结点n是父亲的最右结点，后继是n第一个非最右结点祖先的后继结点。

对第一种情况，注意到(n&-n)取了n最右一位就可以得到结论。对第二种情况，注意到所有最右结点都以x0 11...1100..00结尾，它的祖先结点中最低的不是最右结点的key是x0 10...0，而它的右结点值是x1 00...1100..00，正好还是n + (n & -n)。

数据映射

tree[idx] = cumsum[idx] - cumsum[parent[idx]]

例如上图对应下表：

**idx** 0 1 2 3 4 5 6 7 8 9 10 11 12

input[idx] 2 1 1 3 2 3 4 5 6 7 8 9

cumsum[idx] 0 2 3 4 7 9 12 16 21 27 34 42 51

**tree[idx]** 0 2 3 1 7 2 5 4 9 6 13 8 30

初始化：tree[idx]初始化为长为数组长度+1的全零数组。

检索：检索cumsum只需把node n+1路径上所有结点加起来，求range sum就是两个cumsum之差。

更新input[n]：假设增加d, 影响的元素是结点n+1以及它到根结点路径上所有结点的后续兄弟结点。这些元素都需要加d

例如更新input[4]，则需更新结点5(对应input[4]), 6, 8.

最后注意Fenwick tree不直接存储data，因此如果问题需要access data方便起见最好另存一份。

|  |
| --- |
| **class NumArray** {  public:  **NumArray(vector<int> &nums)** : data(nums), tree(nums.size()+1, 0) {  for (int i = 0; i < data.size(); ++i)  for (int key = i+1; key < tree.size(); key += (key&-key)) tree[key] += data[i];  }  **void update(int i, int val)** {  int diff = val - data[i];  for (int key = i+1; key < tree.size(); key += (key&-key)) tree[key] += diff;  data[i] = val;  }  **int sumRange(int i, int j)** {  return cumSum(j) - cumSum(i-1);  }  private:  int cumSum(int i) {  if (i < 0) return 0;  int sum = 0;  for (int key = i + 1; key > 0; key &= (key-1)) sum += tree[key];  return sum;  }  vector<int> tree;  vector<int> data;  }; |

**Solution 2：Segment Tree with Fenwick tree like implementation.**

线段树是一棵二叉树。每个结点对应一个区间[l,r),结点的两个子结点把这个区间的值分为均匀的两段。每个结点还存储这个区间里的sum。根结点对应[0, n)。

这里我们使用数组存储一棵二叉完全树。为index方便，我们把n扩展到2的整数次幂M = 2k. 上面的例子对应的Segment Tree如下图。

5

**[4,6)**

51

**[0,16)**

21

**[0,8)**

30

**[8,16)**

7

**[0,4)**

14

**[4,8)**

3

**[0,2)**

1

**[2,3)**

3

**[5,6)**

30

**[8,12)**

9

**[6,8)**

6

**[8,9)**

17

**[10,12)**

9

**[11,12)**

2

**[0,1)**

1

**[1,2)**

3

**[3,4)**

2

**[4,5)**

4

**[6,7)**

5

**[7,8)**

7

**[9,10)**

8

**[10,11)**

4

**[2,4)**

13

**[8,10)**

0

**[12,16)**

0

**[12,14)**

0

**[14,16)**

**1(1)**

**2(10)**

**3(11)**

**4(100)**

**5(101)**

**6(101)**

**7(111)**

**8(1000)**

**9(1001)**

**10(1010)**

**11(1011)**

**12(1100)**

**13(1101)**

**14(1110)**

**15(1111)**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28,29**

**30,31**

其中的红字表示结点的index。

结点关系映射：注意到index的父结点是子结点的前缀。因此

* parent(n) = n>>1;
* left\_child(n) = n\*2;
* right\_child(n) = n\*2+1;

数据映射：

define BASE = next\_power\_of\_two(size of input)

tree[n] = tree[left\_child[n)] + tree[right\_child(n)] for n < BASE

tree[n] = input[n-BASE) for n >= BASE

初始化：

这样构造树时，则把输入复制从tree[M]开始的空间。然后从tree[M-1]开始，用tree[n] = tree[n\*2] + tree[n\*2+1]构造。

检索：

当我们需要检索区间(j, k)时(即区间[j+1, k-1])，相关结点都在j, k的最小公共子树里，然后寻找到j的path上所有左结点的右兄弟之和，和到k的path上所有右结点的左兄弟之和。我们只需要迭代指向j, k的父亲，直到两者指向同一结点。

更新：当更新A[k]时，只要相应更新根结点到结点[k,k]路径上所有结点的value.这通过迭代访问parent(n)即可做到。

边界条件：开区间访问需要检索A[-1]和A[end+1]，因此我们增加两个元素，同时把index向右平移1.

|  |
| --- |
| inline unsigned int next\_power\_of\_two(unsigned int v) {  --v;  v |= (v >> 1);  v |= (v >> 2);  v |= (v >> 4);  v |= (v >> 8);  v |= (v >> 16);  return v+1;  }  **class NumArray** {  public:  **NumArray(vector<int> &nums)** : M(next\_power\_of\_two(nums.size()+2)) {  tree.resize(M<<1, 0);  copy(nums.begin(), nums.end(), tree.begin()+M+1);  for (int k = M-1; k>0; k--) tree[k] = tree[k<<1] + tree[k<<1|1];  }  **void update(int i, int val)** {  i++; //number is 1 based  int diff = val - tree[M+i];  for (int key = M+i; key > 0; key >>= 1) tree[key] += diff;  }  **int sumRange(int i, int j)** {  int sum = 0;  for (i += M, j += M+2; i ^ j ^ 1; i >>= 1, j >>= 1) {  if (!(i&1)) sum += tree[i^1];  if (j&1) sum += tree[j^1];  }  return sum;  }  private:  vector<int> tree; //tree stored in array, size is M\*2  int M; //smallest power of 2 greater or equal than nums.size()+2  }; |

# 308. Range Sum Query 2D - Mutable [Locked]

|  |
| --- |
| Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).  Note:  The matrix is only modifiable by the update function.  You may assume the number of calls to update and sumRegion function is distributed evenly.  You may assume that row1 ≤ row2 and col1 ≤ col2. |
| Example: Given matrix = [  [3, 0, 1, 4, 2],  [5, 6, 3, 2, 1],  [1, 2, 0, 1, 5],  [4, 1, 0, 1, 7],  [1, 0, 3, 0, 5]  ]  sumRegion(2, 1, 4, 3) -> 8  update(3, 2, 2)  sumRegion(2, 1, 4, 3) -> 10 |

**Solution: Segment tree**

把1D的Segment Tree（或Fenwick Tree）扩展到2D。办法是类似1D segment tree划分结点。

**Step 3**

**Step 1**

**Step 2**

数据结构：上左图中红色区域表示数组，蓝色区域表示tree的数据，记录某个2D region的sum，黄色块和红蓝线表示结点间的父子关系。不难看出这个2D矩阵的每行、每列都是一个1D Segment tree

初始化： 首先把数据copy到矩阵右下角，然后先按行初始化data块左边的tree数组，然后按列更新前一半行的tree 数组。如上右图。

更新：

如上左图。但更新结点时因为data是2D的。因此结点(i, j)的更新要传播要所有i的祖先和j的祖先的笛卡尔积上。

导致的变化在于每个结点需要更新log(m)\*log(n)个结点，

检索：

首先对每行做range sum。然后对每列做range sum

类似的，把fenwick tree扩展到2D也可以解决这个问题

|  |
| --- |
| inline unsigned int next\_power\_of\_two(unsigned int v) {  --v;  v |= (v >> 1);  v |= (v >> 2);  v |= (v >> 4);  v |= (v >> 8);  v |= (v >> 16);  return v+1;  }    class NumMatrix {  public:  NumMatrix(vector<vector<int>> &matrix)  : row\_base(next\_power\_of\_two( matrix.size()+2 )),  col\_base(next\_power\_of\_two( matrix.size()?(matrix[0].size()+2) : 2 ),  tree(row\_base \* 2, vector<int>(col\_base \* 2, 0)) {  for (int i = 0; i < matrix.size(); ++i) {  //copy a row of matrix to tree  copy(matrix[i].begin(), matrix[i].end(), tree[row\_base+i+1].begin()+col\_base+1);  //compute bottom-left part of tree matrix  for (int j = col\_base -1; j>0; --j)  tree[row\_base+i+1][j] = tree[row\_base+i+1][j<<1]+tree[row\_base+i+1][j<<1|1];  }  //compute top part of the tree matrix  for (int i = row\_base-1; i >0; --i)  for (int j = col\_base\*2-1; j >0; --j)  tree[i][j] = tree[i<<1][j] + tree[i<<1|1][j];    }  void update(int row, int col, int val) {  ++row, ++col; // the matrix are stored in tree with 1 based index  int diff = val - tree[row\_base+row][col\_base+col];  // update all ancestors of node (row\_base+row, col\_base+col)  for (int k = row\_base+row; k > 0; k >>=1) {  for (int j = col\_base+col; j >0; j>>=1) {  tree[k][j] += diff;  }  }  }  int sumRegion(int row1, int col1, int row2, int col2) {  int sum = 0;  //sum over row ranges  for (int i1 = row1+row\_base, i2 = row2+row\_base+2; i1^i2^1; i1>>=1, i2>>=1) {  if (!(i1&1)) sum += sumRow(i1^1, col1, col2);  if (i2&1) sum += sumRow(i2^1, col1, col2);  }  return sum;  }  private:  inline int sumRow(int i, int col1, int col2) { //sum over column ranges  vector<int>& t = tree[i];  int sum = 0;  for (int j1 = col1+col\_base, j2 = col2+col\_base+2; j1^j2^1; j1>>=1, j2>>=1) {  if (!(j1&1)) sum += t[j1^1];  if (j2&1) sum += t[j2^1];  }  return sum;  }    const int row\_base, col\_base; //half of tree matrix size  vector<vector<int>> tree; //2D tree matrix  }; |

# 309. Best Time to Buy and Sell Stock with Cooldown

|  |
| --- |
| Say you have an array for which the ith element is the price of a given stock on day i.  Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:  You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).  After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day) |
| Example:  prices = [1, 2, 3, 0, 2]  maxProfit = 3  transactions = [buy, sell, cooldown, buy, sell] |

**Solution: DP**

用local[i]存储最后一次卖出在第i天的最大利益，global[i]存储到第i天的最大利益。

因此：

local[i] = max(global[i-3] + prices[i]-prices[i-1], local[i-1]+prices[i]-prices[i-1])

global[i] = max(global[i-1], local[i])

|  |
| --- |
| **int maxProfit(vector<int>& prices)** {  if (prices.size() == 0) return 0;    vector<int> local(prices.size(), 0);  vector<int> global(prices.size(), 0);    for (int k = 1; k < prices.size(); ++k) {  int last = k < 3? 0 : global[k-3];  local[k] = max(last + prices[k]-prices[k-1], local[k-1] + prices[k]-prices[k-1]);  global[k] = max(global[k-1], local[k]);  }  return global.back();  } |

# 310. Minimum Height Trees

|  |
| --- |
| For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.  Format  The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).  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.  Example 1: Given n = 4, edges = [[1, 0], [1, 2], [1, 3]], return [1]  Example 2: Given n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]], return [3, 4]  0 0 1 2  | \ | /  1 3  / \ |  2 3 4  |  5 |

**Solution: Graph theory + Topological sort**

对树状结构，根结点的度数是1.

因此我们使用类似拓扑排序的方式，每次扫掉一层。最后扫到的一层即为可能的根结点。

|  |
| --- |
| **vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges)** {  vector<int> degree(n, 0);  vector<list<int>> neighbor(n, list<int>{});    for (pair<int, int> p : edges) {  degree[p.first]++;  degree[p.second]++;  neighbor[p.first].push\_back(p.second);  neighbor[p.second].push\_back(p.first);  }  vector<int> pre;  for (int k = 0; k < n; ++k) if (degree[k]<=1) pre.push\_back(k);    int count = n;  while (count > pre.size()) {  vector<int> cur;  for (int src : pre) {  for (int dst : neighbor[src]) if (--degree[dst] == 1) cur.push\_back(dst);  --count;  }  swap(pre, cur);  }  return pre;  } |

# 311. Sparse Matrix Multiplication [Locked]

|  |
| --- |
| Given two sparse matrices A and B, return the result of AB.  You may assume that A's column number is equal to B's row number.  Example:  [ 7, 0, 0 ] [ 7 0 0 ]  A = [ 1, 0, 0] B = [ 0, 0, 0 ], AB = [ 1 0 0 ] x [ 0 0 0 ] = [ 7 0 0 ]  [-1, 0, 3] [ 0, 0, 1 ] [ -1 0 3 ] [ 0 0 1 ] [ -7 0 3 ] |

**Solution: merge sort + row-col-nonzero val rep.**

用一个vector<vector<int>>存储两个矩阵的非零元素位置。A的第一层为行，B的第一层为列。这样我们只需要计算输出矩阵中的非零元素。

假设A大小为mxn, B大小为nxk, 抽取非零元素的复杂度是O(mn+nk)。

假设输出矩阵非零元素个数为p，A, B每行非零元素个数约为q个,则计算输出矩阵的复杂度为O(pq)。假设p是O(n)级别的，则计算步骤复杂度大致是O(n)级别的。（因为q为常数）。

|  |
| --- |
| vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B) {  vector<pair<int, vector<int>>> Anz, Bnz;  //make compact representation of A  for (int i = 0; i < A.size(); ++i) {  vector<int> nz;  for (int j = 0; j < A[0].size(); ++j)  if (A[i][j] != 0) nz.push\_back(j);  if (!nz.empty()) Anz.push\_back(make\_pair(i, nz));  }  //make compact representation of B's transpose  for (int j = 0; j < B[0].size(); ++j) {  vector<int> nz;  for (int i = 0; i < B.size(); ++i)  if (B[i][j] != 0) nz.push\_back(i);  if (!nz.empty()) Bnz.push\_back(make\_pair(j, nz));  }  //do multiplication  vector<vector<int>> result(A.size(), vector<int>(B[0].size(), 0));  for (auto p : Anz)  for (auto q: Bnz) result[p.first][q.first] = inner\_prod(A, B, p, q);  return result;  }  inline int inner\_prod(const vector<vector<int>>& A, const vector<vector<int>>& B,  const pair<int, vector<int>>& p, const pair<int, vector<int>>& q) {  const vector<int>& posA = p.second;  const vector<int>& posB = q.second;  int r = p.first, c = q.first, sum = 0;  for (int i = 0, j = 0; i < posA.size() && j < posB.size(); ++i, ++j) {  if (posA[i] == posB[j]) sum += A[r][posA[i]] \* B[posB[j]][c];  else if (posA[i] < posB[j]) --j;  else --i;  }  return sum;  } |

# 312. Burst Baloons

|  |
| --- |
| Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] \* nums[i] \* nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.  Find the maximum coins you can collect by bursting the balloons wisely.  Note:  (1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.  (2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100 |
| Example: Given [3, 1, 5, 8] Return 167  nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []  coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167 |

**Solution: DP**

用2D矩阵存储范围[i,j]被burst产生的最大收入。从长度为1的范围开始计算。对一个长度为k的range，遍历所有最后一个戳破的气球编号。

|  |
| --- |
| int maxCoins(vector<int>& nums) {  if (nums.size() == 0) return 0;    vector<vector<int>> coins(nums.size(), vector<int>(nums.size(), 0));    for (int k = 1; k <= nums.size(); ++k) { //number of neighboring baloons bursted  for (int j = 0; j+k <= nums.size(); ++j) { // first baloon id  int cl = j == 0? 1: nums[j-1], cr = j+k==nums.size()? 1 : nums[j+k];    for (int l = j; l < j+k; ++l) { //last baloon bursted  int c1 = (l==j)?0:coins[j][l-1];  int c2 = (l==j+k-1)?0:coins[l+1][j+k-1];  int c3 = cl \* cr \* nums[l];  coins[j][j+k-1] = max(coins[j][j+k-1], c1+c2+c3);  }  }  }    return coins[0].back();  } |

# 313. Super Ugly Number

|  |
| --- |
| Write a program to find the nth super ugly number.  Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k.  Note:  (1) 1 is a super ugly number for any given primes.  (2) The given numbers in primes are in ascending order.  (3) 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000. |
| For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4. |

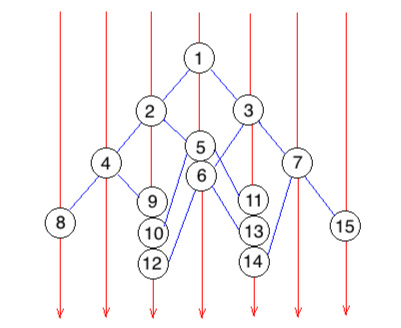
**Solution: merge sort**

维护k个列表，第j个存储因子含primes[j]。在实现中我们维护长为n的数组history和一个长为k个元素的数组candidate，表示列表的下一个数字是history[candidate[j]]\*primes[j]。每次我们把最小表头minv的加入history，并把所有表头=minv的表对应的candidate加1.

|  |
| --- |
| **int nthSuperUglyNumber(int n, vector<int>& primes)** {  vector<int> candidate(primes.size(), 0);  vector<int> history(n, 1);    for (int k = 1; k < n; ++k) {  int min\_val = INT\_MAX;  for (int k = 0; k < candidate.size(); ++k)  min\_val = min(min\_val, history[candidate[k]] \* primes[k]);  history[k] = min\_val;  for (int k = 0; k < candidate.size(); ++k)  if(history[candidate[k]] \* primes[k] == min\_val) candidate[k]++;  }  return history.back();  } |

# 314. Binary Tree Vertical Order Traversal [Locked]

|  |
| --- |
| Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).  If two nodes are in the same row and column, the order should be from left to right. |



**Solution: Pre-order traversal**

这个题的意思看着有点晕，leetcode上的几个例子太简单，不太说明问题。网上搜到一个比较复杂的例子。

算法：

前序遍历，同时维护当前的column和depth。用两个vector维护column id为正／为负的所有columns。插入的元素还包括元素深度。

树遍历完后对每个column内部按照深度stable sort。

|  |
| --- |
| bool cmp (pair<int, int>p1, pair<int, int>p2) {  return p1.first < p2.first;  }  vector<vector<int>> verticalOrder(TreeNode\* root) {  vector<vector<pair<int, int>>> col\_pos, col\_neg;  verticalOrder\_recur(col\_pos, col\_neg, root, 0, 0);  vector<vector<int>> result;  for (auto it = col\_neg.rbegin(); it != col\_neg.rend(); ++it) {  result.push\_back(vector<int>{});  stable\_sort(it->begin(), it->end(), cmp);  for (auto &v : \*it) result.back().push\_back(v.second);  }  for (auto it = col\_pos.begin(); it != col\_pos.end(); ++it) {  result.push\_back(vector<int>{});  stable\_sort(it->begin(), it->end(), cmp);  for (auto &v : \*it) result.back().push\_back(v.second);  }  return result;  }  void verticalOrder\_recur(vector<vector<pair<int, int>>>& col\_pos,  vector<vector<pair<int, int>>>& col\_neg,  TreeNode\* root, int col\_idx, int depth) {  if (root == NULL) return;  if (col\_idx >= 0) {  if (col\_pos.size() == col\_idx) col\_pos.push\_back(vector<pair<int, int>>{});  col\_pos[col\_idx].emplace\_back(depth, root->val);  }  else {  if (col\_neg.size() == -col\_idx-1) col\_neg.push\_back(vector<pair<int, int>>{});  col\_neg[-col\_idx-1].emplace\_back(depth, root->val);  }    verticalOrder\_recur(col\_pos, col\_neg, root->left, col\_idx-1, depth+1);  verticalOrder\_recur(col\_pos, col\_neg, root->right, col\_idx+1, depth+1);  } |

# 315. Count of Smaller Numbers After Self

|  |
| --- |
| You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i]. |
| Example: Given nums = [5, 2, 6, 1]  To the right of 5 there are 2 smaller elements (2 and 1).  To the right of 2 there is only 1 smaller element (1).  To the right of 6 there is 1 smaller element (1).  To the right of 1 there is 0 smaller element.  Return the array [2, 1, 1, 0]. |

**Solution1: insert-only BST**

用二叉树维护已经出现的数字。从右到左每次插入一个数字，并返回在二叉树中比当前数子小的数。最坏情况O(n2).

递归insert: 完成插入操作，维护树结构，并返回<v的元素个数

1. ++r->size
2. 根据比较v和tree node r的值决定插入在r的左侧还是右侧。
3. 如果插入右侧的话把r左子树的size计入count，r本身也可能计入count

边界条件：

* r == NULL: 此时在r引用的位置创建一个新node
* r->val == v: v插入右侧，从而保证每个节点左子树的值都<=根，右子树>根。

|  |
| --- |
| class BSTree { //a simple BST that does not guarantee balanced structure  struct Node {  Node\* left, \* right;  int size, val;  Node(int v) : left(nullptr), right(nullptr), val(v), size(1) {};  ~Node() { delete left; delete right; left = right = nullptr; }  };  static int insert\_recur(Node\*& r, int v) {  if (r == NULL) {  r = new Node(v);  return 0;  }  else {  ++r->size;  if (v > r->val) return insert\_recur(r->right, v) + 1  + (r->left? r->left->size : 0);  else return insert\_recur(r->left, v);  }  }  Node \*root = nullptr;  public:  ~BSTree() { delete root; }  int insert(int v) {  return insert\_recur(root, v);  }  };  **vector<int> countSmaller(vector<int>& nums)** {  vector<int> result(nums.size(), 0);  BSTree tree;  for (int k = nums.size()-1; k >=0; --k)  result[k] = tree.insert(nums[k]);  return result;  } |

**Solution2: insert-only balanced BST (e.g. Treap)**

二叉树的问题在于它可能很不平衡。平衡二叉树有AVL树、红黑树等，但比较好实现的是treap(随机二叉搜索树）。treap结点除了向父、子结点的指针和自身的值，最重要的属性是一个随机生成的priority。当插入一个新值时，需要通过rotation使树的priority符合heap属性。

递归insert的实现：

和BSTree几乎一样，只是除了插入后可能进行左旋或右旋来keep priority上的heap属性。对augmented tree node，例如此题中带size信息的treenode，还需要更新因为结构调整带来的treenode更新。

rotate的实现

当把node v 插入node g的子树p后，可能导致p的priority>g的，此时需要旋转。相关的node除了g, p，还包括p在另一个方向的child c。

这样只需要把BST改成treap，复杂度就下降到O(logn)

|  |
| --- |
| class Treap {  struct Node {  int key, priority, size;  Node \*left, \*right;  Node (int k, int p): key(k), priority(p), size(1), left(NULL), right(NULL) {};  ~Node() { delete left; delete right; }  };  inline static int get\_size(Node \*node) { return node? node->size : 0;}    void rotate(Node\*& gref, Node\*& pref, Node\* &cref) {  //rotate  Node\* const g = gref, \* const p = pref, \* const c = cref;  pref = c, cref = g;  //update size  swap(p->size, g->size);  g->size = get\_size(g->left) + get\_size(g->right) + 1;  gref = p; //update reference to new root  }  int insert\_recur(Node\* &nref, int key, int priority) {  int count = 0;  if (nref == NULL) nref = new Node(key, priority);  else {  bool go\_left = key <= nref->key;  //insert and count #nodes < key  Node\*& child = go\_left? nref->left : nref->right;  count += insert\_recur(child, key, priority);  if (!go\_left) count += get\_size(nref->left)+1;  //rotate and update size  ++nref->size;  if (child->priority > nref->priority)  rotate(nref, child, go\_left? child->right : child->left);  }  return count;  }  Node\* root = NULL;    public:  ~Treap() { delete root; }  int insert(int key) { return treap\_insert(root, key, rand()); }  };  **vector<int> countSmaller(vector<int>& nums) {**  vector<int> result(nums.size(), 0);  Treap tree;  for (int k = nums.size()-1; k>=0; --k)  result[k] = tree.insert(nums[k]);  return result;  } |

**Solution3: merge sort**

对nums合并排序，在合并的过程中左边子数组的count增加当前已经被合并的右边子数组元素个数，右边子数组的结果不变。

例如：

Merge [5], [2] -> [2 5], count = [0, 1], idx = [1 0]

Merge [6], [1] -> [1 6], count = [0, 1], idx = [3 2]

Merge [2 5] [1 6] -> [1 2 5 6], count = [0 1 2 1], idx = [3 1 0 2]

Reorder [2 1 1 0] (result[idx[k]] = count[k])

|  |
| --- |
| **vector<int> countSmaller(vector<int>& nums)** {  vector<int> result(nums.size(), 0);  vector<int> idx(nums.size(), 0);  for (int k = 0; k < nums.size(); ++k) idx[k] = k;    for (int t = 2; t <= nums.size()\*2; t \*= 2) {  vector<int> result\_copy(result);  vector<int> idx\_copy(idx);  int first = 0, mid = t/2, last = t;  for (; mid < nums.size(); first+= t, mid += t, last += t) {   //merge [first, mid), [mid, last)  if (last > nums.size()) last = nums.size();    int k = first, i = first, j = mid, c2 = 0;  while (i < mid || j < last) {  if (j == last || (i < mid && nums[idx\_copy[i]] <= nums[idx\_copy[j]])) {  idx[k] = idx\_copy[i];  result[k++] = result\_copy[i++] + c2;  }  else {  idx[k] = idx\_copy[j];  result[k++] = result\_copy[j++];  c2++;  }  }  }  }    for (int k = 0; k < nums.size(); ++k) nums[idx[k]] = result[k];  return nums;  } |

**Solution 4: Fenwick tree**

我们把数字按照从小到大stable\_sort（这样同样值的code后面的比较小），用sort后的位置encode。

例如：[5 2 6 1 1] 的编码是 [3 2 4 0 1].

这样我们从右到左用Fenwick Tree maintain一个histogram，并在位置k返回[0, code[k]-1]的cumsum.

|  |
| --- |
| **vector<int> countSmaller(vector<int>& nums)** {  vector<int> code = encode(nums);  vector<int> tree(nums.size()+1, 0);  vector<int> result(nums.size(), 0);  for (int k = nums.size()-1; k >=0; --k) {  for (int v = code[k]; v>0; v &= (v-1)) result[k] += tree[v];  for (int v = code[k]+1; v<=nums.size(); v += (v&-v)) ++tree[v];  }  return result;  }  inline vector<int> encode(vector<int>& nums) {  vector<int> idx(nums.size(), 0);  iota(idx.begin(), idx.end(), 0);  stable\_sort(idx.begin(), idx.end(), [&nums](int i, int j){return nums[i] < nums[j];});  vector<int> code(nums.size(), 0);  for (int k = 0; k < nums.size(); ++k) code[idx[k]] = k;  return code;  } |

# 

# 

# 316. Remove Duplicate Letters

|  |
| --- |
| Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results. |
| Example:  Given "bcabc" Return "abc"  Given "cbacdcbc" Return "acdb" |

**Solution: iteration + stack**

无重复字母序最小串条件：对任意0<=k < s.size()，我们把字串拆分成两部分

* prefix: 出现在前k个字符里的字母
* suffix: 出现在后k个字符里的字母

则prefix满足三个条件：

1. 未出现字母在s[k+1…]中出现
2. 无重复字母
3. 在s[0...k]所有满足条件(1)(2)的子串中，prefix字母序最小

递归结构：假设已知prefix[k], c = s[k+1]

prefix[k] = minj {prefix[0...j]+c | prefix[0...j]+c满足条件(1)(2)}

stack状更新：若c已在prefix[k]中出现过，则满足条件(2)的prefix[0...j]+c必然是prefix的前缀，在字母序上我们认为prefix更小，因此直接跳过该字母。

若c未出现过，则可以通过逐渐删除prefix末尾>c的字符，直到末字符若删除会导致条件(1)不满足。

这个过程类似于stack，但我们知道stack最长为unique字母个数，且从不超过这个长度。

复杂度：因为每个元素进出输出字符串(stack)一次，因此复杂度为O(n), n为输入字符串长度

|  |
| --- |
| **string removeDuplicateLetters(string s)** {  int hist[26] = {0};  int nchar = 0;  for (char c : s) if (++hist[c-'a']==1) ++nchar;    bool visited[26] = {false};  string result(nchar, 0); int pos = 0    for (char c:s) {  hist[c-'a']--;  if (!visited[c-'a']) {  while (pos > 0 && c < result[pos-1] && hist[result[pos-1]-'a']>0 )  visited[result[--pos]-'a'] = false;  result[pos++] = c;  visited[c-'a'] = true;  }  }  return result;  } |

# 317. Shortest Distance from All Buildings [Locked]

|  |
| --- |
| You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:   * Each 0 marks an empty land which you can pass by freely. * Each 1 marks a building which you cannot pass through. * Each 2 marks an obstacle which you cannot pass through.   For example, given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2):  1 - 0 - 2 - 0 - 1  | | | | |  0 - 0 - 0 - 0 - 0  | | | | |  0 - 0 - 1 - 0 - 0  The point (1,2) is an ideal empty land to build a house, as the total travel distance of 3+3+1=7 is minimal.  So return 7.  Note:  There will be at least one building. If it is not possible to build such house according to the above rules, return -1. |

**Solution: BFS**

从每个building出发作BFS，从而得到每个building到每个empty land的距离。最后比较所有距离的和，取最小的。

|  |
| --- |
| inline  void add\_to\_cur(vector<int>& cur, vector<bool>& visited, int i, int j, int M, int N) {  int id = i \* N + j;  if (i < 0 || j < 0 || i >= M || j >=N || visited[id]) return;  visited[id] = true;  cur.push\_back(id);  }  inline  void update\_dist(vector<vector<int>> &grid, vector<vector<int>> &dist, int i0, int j0) {  const int M = grid.size(), N = M==0? 0 : grid[0].size();  vector<bool> visited(M\*N, false);  vector<int> pre;  grid[i0][j0] = 0;  add\_to\_cur(pre, visited, i0, j0, M, N);  for (int l = 0; !pre.empty(); ++l) {  vector<int> cur;  for (int id : pre) {  int i = id/N, j = id%N;  if (grid[i][j]>0 || dist[i][j]==INT\_MAX) continue;  //cannot go through wall or house or unreachable cells  dist[i][j] += l;  add\_to\_cur(cur, visited, i-1, j, M, N);  add\_to\_cur(cur, visited, i+1, j, M, N);  add\_to\_cur(cur, visited, i, j-1, M, N);  add\_to\_cur(cur, visited, i, j+1, M, N);  }  swap(pre, cur);  }  grid[i0][j0] = 1;    for (int id = 0; id < M\*N; ++id) {  if (visited[id]==false) dist[id/N][id%N] = INT\_MAX;  }  }  int shortestDistance(vector<vector<int>> grid) {  vector<vector<int>> total\_dist(grid.size(), vector<int>(grid[0].size(), 0));  for (int i = 0; i < grid.size(); ++i) {  for (int j = 0; j < grid[0].size(); ++j) {  if (grid[i][j] == 1) update\_dist(grid, total\_dist, i, j);  }  }  int min\_dist = INT\_MAX;  for (int i = 0; i < grid.size(); ++i) {  for (int j = 0; j < grid[0].size(); ++j) {  if (grid[i][j]==0) min\_dist = min(min\_dist, total\_dist[i][j]);  }  }  return min\_dist==INT\_MAX? -1:min\_dist;  } |

# 318. Maximum Product of Word Lengths

|  |
| --- |
| Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0. |

**Solution1: bitset + prunning**

bit mask对单词编码，这样可以很快知道两个单词是否有共同字母。通过事先对word排序可以提前退出循环。

|  |
| --- |
| int maxProduct(vector<string>& words) {  sort(words.begin(), words.end(),  [](string s1, string s2) { return s1.size()< s2.size();} );  vector<int> code(words.size(), 0);  for (int k = 0; k < words.size(); ++k)  for (char c : words[k]) code[k] |= 1 << (c-'a');  int maxlen = 0;  for (int k = words.size()-1; k>=0; --k) {  if (words[k].size() \* words[k].size() <= maxlen) break;  for (int j = k-1; j>=0; --j) {  if (words[j].size() \* words[k].size() <= maxlen) break;  if ((code[k] & code[j]) == 0)  maxlen = max(maxlen, (int)(words[k].size() \* words[j].size()));  }  }  return maxlen;  } |

**Solution2: bitset + bin sort**

Maintain一个从编码到最长单词的mapping。然后把code按从大到小排序。

然后对每两个&操作为0的code，把它们的最长单词乘积用于update最大乘积。

|  |
| --- |
| int maxProduct(vector<string>& words) {  unordered\_map<int, int> len;  for (string &w : words) {  int idx = 0;  for (char c : w) idx = idx | (1 << (c-'a'));  len[idx] = max(len[idx], static\_cast<int>(w.size()));  }  vector<pair<int, int>> invert\_map;  for (auto &p : len) invert\_map.emplace\_back(p.second, p.first);  sort(invert\_map.begin(), invert\_map.end());  int max\_len = 0;  for (int k = invert\_map.size()-1; k > 0; --k) {  if (invert\_map[k].first \* invert\_map[k].first <= max\_len) break;  for (int j = k-1; j >=0; --j) {  int prod = invert\_map[k].first \* invert\_map[j].first;  if (prod <= max\_len) break;  if ((invert\_map[k].second & invert\_map[j].second) == 0)  max\_len = max(max\_len, prod);  }  }  return max\_len;  } |

# 319. Bulb Switcher

|  |
| --- |
| There are n bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the ith round, you toggle every i bulb. For the nth round, you only toggle the last bulb. Find how many bulbs are on after n rounds. |
| Example: Given n = 3.  At first, the three bulbs are [off, off, off].  After first round, the three bulbs are [on, on, on].  After second round, the three bulbs are [on, off, on].  After third round, the three bulbs are [on, off, off].  So you should return 1, because there is only one bulb is on. |

**Solution: parity**

开关第n个bulb的bulbs包括n的所有因子。所以我们只要看每个n的因子数是奇是偶。

令n分解为pow(a1, k1)\*pow(a2, k2)\*...\*pow(an, kn)，则n的因子个数为(k1+1)(k2+1)...(kn+1)。

只有当k1k2...kn都是偶数时才为奇数（bulb on)。因此n必须是完全平方数bulb才会亮。

因此我们计算[1, n]范围中的完全平方数个数就好了。

|  |
| --- |
| int bulbSwitch(int n) {  return sqrt(n);  } |

# 320. Generalized Abbreviation [Locked]

|  |
| --- |
| Write a function to generate the generalized abbreviations of a word.  Example:  Given word = "word", return the following list (order does not matter):  ["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"] |

**Solution: recursion**

递归结构：假设前k个字母已被prefix表示，当前的可能性包括

* prefix末字母为数字，则当前输出必须是字母
* 否则可以选择把word[k]开始的前j个字母改为数字。

|  |
| --- |
| vector<string> generateAbbreviations(string word) {  vector<string> result;  string prefix;  generateAbbr\_recur(word, 0, prefix, result);  return result;  }  void generateAbbr\_recur(const string &word, int k, string prefix, vector<string> &result) {  if (prefix.size() > 0 && isdigit(prefix.back()) && k<word.size()) prefix += word[k++];  if (k == word.size()) {  result.push\_back(prefix);  return;  }    for (int j = 1; k+j <= word.size(); ++j)  generateAbbr\_recur(word, k+j, prefix + to\_string(j), result);  } |

**Solution: DP**

可以cache word从第k个字母开始的压缩，分为以数字开始的S1和不以数字开始的S2：

递归结构：S1[k] = {num2str(j)+str | str in S2[k+j]} S2[k] = {word[k]+str |str in S1[k+1] U S2[k+1]}

DP初始条件：k = word.size()时S1={""}, S2={""}

|  |
| --- |
| vector<string> generateAbbreviations(string word) {  vector<vector<string>> set\_num(word.size()+1, vector<string>{});  vector<vector<string>> set\_char(word.size()+1, vector<string>{});  set\_num.back().push\_back("");  set\_char.back().push\_back("");  for (int k = word.size()-1; k>=0; --k) {  for (int j = 1; k+j <= word.size(); ++j)  for (string &s : set\_char[k+j]) set\_num[k].push\_back(to\_string(j)+s);  for (string &s : set\_num[k+1]) set\_char[k].push\_back(word[k]+s);  if (k < word.size()-1)  for (string &s : set\_char[k+1]) set\_char[k].push\_back(word[k]+s);  }  vector<string> result;  result.insert(result.end(), set\_num[0].begin(), set\_num[0].end());  result.insert(result.end(), set\_char[0].begin(), set\_char[0].end());  return result;  } |

# 321. Create Maximum Number

|  |
| --- |
| Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity. |
| Example 1:  nums1 = [3, 4, 6, 5] nums2 = [9, 1, 2, 5, 8, 3] k = 5  return [9, 8, 6, 5, 3]  Example 2:  nums1 = [6, 7] nums2 = [6, 0, 4] k = 5  return [6, 7, 6, 0, 4]  Example 3:  nums1 = [3, 9] nums2 = [8, 9] k = 3  return [9, 8, 9] |

**Solution: stack + merge sort**

问题分解：

如果数字串是字母序最小的，那么它的字符中属于l1的字符构成的子串也是字母序最小的。所以我们可以遍历数字串中属于l1的个数，从而把问题转化为对pair(l, k-l):

(1)找l1中字母序最小的长为l 的串s1

(2)找l2中字母序最小的长为k-l 的串s2

(3)找s1, s2的字母序最小的merge

对所有l种可能性，再选字母序最小的即可。

子问题算法：

对子问题(1)，我们可以用316的stack方法来做。

对子问题(2)，我们每次都要确定当前数字为可选的最小数字。这就是对两个串进行merge sort的merge步骤。

|  |
| --- |
| vector<int> maxNumber(vector<int>& nums1, vector<int>& nums2, int k) {  vector<int> result(k, 0);  for (int i = 0; i <= k; ++i) {  if (i > nums1.size() || k-i > nums2.size()) continue;  vector<int> v = mergeNumber(maxNumber(nums1, i), maxNumber(nums2, k-i));  result = greater(result, v, 0, 0)? result : v;  }  return result;  }  // return vector representing larger number  bool greater(vector<int>& nums1, vector<int>& nums2, int i, int j) {  for (; i < nums1.size(); ++i, ++j) {  if (j >= nums2.size()) return true;  if (nums1[i] > nums2[j]) return true;  if (nums1[i] < nums2[j]) return false;  }  return false;  }  // return max number formed by two arrays, assuming total length equals k  vector<int> mergeNumber(vector<int> nums1, vector<int> nums2) {  vector<int> result(nums1.size()+nums2.size(), 0);  for (int i = 0, j = 0, k = 0; i < nums1.size() || j < nums2.size(); ++k) {  result[k] = greater(nums1, nums2, i, j)? nums1[i++] : nums2[j++];  }  return result;  }  // return max number formed by single array  vector<int> maxNumber(vector<int> &nums, int k) {  vector<int> res(k, 0);  for (int i = 0, j = 0; j < nums.size(); ++j) {  while (i > 0 && nums[j] > res[i-1] && nums.size()+i-j>k) --i;  if (i < k) res[i++] = nums[j];  }  return res;  } |

# 322. Coin Change

|  |
| --- |
| You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.  Note:  You may assume that you have an infinite number of each kind of coin |
| Example 1: coins = [1, 2, 5], amount = 11 return 3 (11 = 5 + 5 + 1)  Example 2: coins = [2], amount = 3 return -1. |

**Solution: DP**

用C[n]存储组成n最少所需要的coins: C[n] = 1+minj C[n-pj]

|  |
| --- |
| int coinChange(vector<int>& coins, int amount) {  if (amount < 0 || coins.size() == 0) return -1;  vector<int> count(amount+1, INT\_MAX);  count[0] = 0;  for (int k = 1; k <= amount; ++k) {  for (int j = 0; j < coins.size(); ++j) {  if (k >= coins[j] && count[k-coins[j]]<INT\_MAX)  count[k] = min(count[k], 1+count[k-coins[j]]);  }  }  return count.back()==INT\_MAX?-1:count.back();  } |

# 323. Number of Connected Components in an Undirected Graph [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 find the number of connected components in an undirected graph.  Example 1:  0 3  | |  1 --- 2 4  Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.  Example 2:  0 4  | |  1 --- 2 --- 3  Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1.  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. |

**Solution 1：DFS/BFS.**

算法：先构造adjacency list，然后dfs 或bfs.

|  |
| --- |
| **int countComponents(int n, vector<pair<int, int>>& edges)** {  vector<vector<int>> adj(n, vector<int>{});  for (auto &e : edges) {  adj[e.first].push\_back(e.second);  adj[e.second].push\_back(e.first);  }  int count = 0;  vector<bool> visited(n, false);  for (int k = 0; k < n; ++k) {  if (visited[k] == false) {  ++count;  queue<int> q;  q.push(k);  while(!q.empty()) {  int i = q.front();  q.pop();  visited[i] = true;  for (int j : adj[i]) if (!visited[j]) q.push(j);  }  }  }  return count;  } |

**Solution 2：Union Find.**

Fast union:每次union edge两个端点所在分支。

* 检查两点是否已在同一分支
* 如果不在，把root(id1) attach给root（id2)，联通分支数-1

|  |
| --- |
| **int countComponents(int n, vector<pair<int, int>>& edges)** {  vector<int> prev(n, 0);  iota(prev.begin(), prev.end(), 0);    int count = n;  for (auto e : edges) {  int r1 = get\_root(e.first, prev);  int r2 = get\_root(e.second, prev);  prev[r1] = r2;  count -= r1==r2?0:1;  }  return count;  }  inline int get\_root(int n, vector<int>& prev) {  int r;  for (r = n; r != prev[r]; r = prev[r]);  return r;  } |

Union-find 的两个优化策略：

* path compression: 把id路径上的所有node attach给parent
* Weighted union: 另开一个数组记录每个node的子孙数，每次union时把weight小的attach到大的下面。

|  |
| --- |
| **int countComponents(int n, vector<pair<int, int>>& edges)** {  vector<int> prev(n, 0);  vector<int> weight(n, 1);  iota(prev.begin(), prev.end(), 0);    int count = n;  for (auto e : edges) {  int r1 = get\_root(e.first, prev);  int r2 = get\_root(e.second, prev);  if (r1 != r2) {  --count;  int rs = weight[r1]<weight[r2]? r1:r2; //smaller root  int rl = weight[r1]<weight[r2]? r2:r1; //larger root  prev[rs] = rl;  weight[rl] += rs;  }  }  return count;  }  inline int get\_root(int n, vector<int>& prev) {  int r;  for (r = n; r != prev[r]; r = prev[r]);  for (int i = n; r != i; i = prev[i]) prev[i] = r; //compression  return r;  } |

# 

# 324. Wiggle Sort II

|  |
| --- |
| Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....  Note:  You may assume all input has valid answer.  Follow Up:  Can you do it in O(n) time and/or in-place with O(1) extra space? |
| Example:  (1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is [1, 4, 1, 5, 1, 6].  (2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is [2, 3, 1, 3, 1, 2]. |

**Solution1 : sort +greedy**

把数组按从大到小序为input[0...n]，我们可以以如下方式重新组织input

input[0...mid-1]->output[1 3 ...mid\*2-1]

input[mid...N-1]->output[0 2 4 ...]

这可以用通项公式input[i]->output[(i\*2+1)/(N|1)]来表示。

|  |
| --- |
| void wiggleSort(vector<int>& nums) {  vector<int> input(nums);  sort(input.begin(), input.end(), greater<int>());  for (int i = 0; i < nums.size(); ++i) nums[(i\*2+1)%(nums.size()|1)] = input[i];  } |

**Solution2: partition + rewiring**

Partition: Solution1提示我们，其实我们只需要知道每个数在前一半还是后一半。因为中位数都可能出现，我们把它放在大的一半的右侧和小的一半的左侧。

Rewiring: 我们把数组index 0, 1, 2, ...映射到1, 3, 5, ...0,2,4…，这时直接在reindex的虚拟index上操作，得到的partition结果即为wiggle sort

|  |
| --- |
| void wiggleSort(vector<int>& nums) {  auto mid\_iter = nums.begin() + nums.size()/2;  nth\_element(nums.begin(), mid\_iter, nums.end());  int mid\_val = \*mid\_iter;  const int N = nums.size();  // P[v] = [> mid] [= mid] [< mid],  // with v[i]=(i\*2+1)%(N|1) be rewired index of i  for (int i = 0, j = N-1, k = 0; k <= j; ++k) {  if (P(nums, k, N) < mid\_val) swap(P(nums, k--, N), P(nums, j--, N));  else if (P(nums, k, N) > mid\_val) swap(P(nums, k, N), P(nums, i++, N));  }  }  inline int& P(vector<int>& nums, int i, const int N) {  return nums[(i\*2 + 1)%(N|1)];  } |

# 325. Maximum Size Subarray Sum equals k ［Locked]

|  |
| --- |
| Given an array nums and a target value k, find the maximum length of a subarray that sums to k. If there isn't one, return 0 instead.  Example 1:  Given nums = [1, -1, 5, -2, 3], k = 3,  return 4. (because the subarray [1, -1, 5, -2] sums to 3 and is the longest)  Example 2:  Given nums = [-2, -1, 2, 1], k = 1,  return 2. (because the subarray [-1, 2] sums to 1 and is the longest)  Follow Up:  Can you do it in O(n) time? |

**Solution: cumsum + unordered\_map**

从左到右计算cumsum存入unordered\_map，map存储cumsum最左位置，在每个位置搜索当前cumsum-k。

|  |
| --- |
| int maxSubArrayLen(vector<int>& nums, int k) {  unordered\_map<int, int> cumsum;  int sum = 0, maxlen = 0;  for (int j = 0; j < nums.size(); ++j) {  auto it = cumsum.find(sum);  if (it == cumsum.end()) cumsum[sum] = j;  sum += nums[j];  it = cumsum.find(sum - k);  if (it != cumsum.end())  maxlen = max(maxlen, j - it->second + 1);  }  return maxlen;  } |

# 326. Power of Three

|  |
| --- |
| Given an integer, write a function to determine if it is a power of three.  Follow up:  Could you do it without using any loop / recursion? |

**Solution1: iteration**

迭代地除以3,直到数字不能被3整除。最后检查剩下的factor是不是1.

|  |
| --- |
| bool isPowerOfThree(int n) {  if (n<=0) return false;  while (n%3==0) n/=3;  return n==1;  } |

**Solution 2：unordered\_set**

在一个set中存储所有3的整数幂。

|  |
| --- |
| bool isPowerOfThree(int n) {  Static unordered\_set<int> valid({1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, 177147, 531441, 1594323, 4782969, 14348907, 43046721, 129140163, 387420489, 1162261467});  return valid.find(n) != valid.end();  } |

**Solution 3: mod**

int范围内最大的3的整数幂是1162261467。检查它是否整除n即可。

|  |
| --- |
| bool isPowerOfThree(int n) {  return n > 0 && 1162261467 % n == 0;  } |

# 327. Count of Range Sum

|  |
| --- |
| Given an integer array nums, return the number of range sums that lie in [lower, upper] inclusive.  Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j (i ≤ j), inclusive.  Note:  A naive algorithm of O(n2) is trivial. You MUST do better than that. |
| Example:  Given nums = [-2, 5, -1], lower = -2, upper = 2,  Return 3.  The three ranges are : [0, 0], [2, 2], [0, 2] and their respective sums are: -2, -1, 2. |

**Solution: cumsum + balanced tree/merge sort**

令C[i] = sum A[0...i]，则S(i, j) = C[j]-C[i-1]。若lower <= S(i, j) <= upper，则：

1. C[j] >= C[i-1] + lower
2. C[j] <= C[i-1] + upper

因此我们得到一个类似315题Count of Smaller Numbers After Self的问题。解决办法之一是通过维护一个排好序的C[0]...C[j-1]结构（例如二叉搜索树）来解决此问题。复杂度为O(nlogn)。

另一个办法是通过merge sort。只是这次在merge时，我们要为每个C[j]，先计算C[j]>=C[i-1]+lower和C[j]>C[i-1]+upper的i的数量，并把两者相减。

|  |
| --- |
| int countRangeSum(vector<int>& nums, int lower, int upper) {  vector<long> sums(nums.size()+1, 0);  for (int k = 0; k < nums.size(); ++k) sums[k+1] = sums[k] + nums[k];  return countRangeSum(sums, 0, sums.size(), lower, upper);  }  int countRangeSum(vector<long>& sums, int first, int last, int lower, int upper) {  if (first == last-1) return 0;  int mid = (first + last)/2;  int count = countRangeSum(sums, first, mid, lower, upper)  + countRangeSum(sums, mid, last, lower, upper);  vector<long> input(sums.begin()+first, sums.begin()+last);  int k = first, m = mid-first, n = mid-first;  for (int i = 0, j = mid-first; i < mid-first; i++, count += n-m) {  while (m < last-first && input[m] < input[i]+lower) m++;  while (n < last-first && input[n] <= input[i]+upper) n++;  while (j < last-first && input[j] < input[i]) sums[k++] = input[j++];  sums[k++] = input[i];  }  return count;  } |

# 328. Odd Even Linked List

|  |
| --- |
| Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.  You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.  Note:  The relative order inside both the even and odd groups should remain as it was in the input.  The first node is considered odd, the second node even and so on ... |
| Example:  Given 1->2->3->4->5->NULL,  return 1->3->5->2->4->NULL. |

**Solution: linked list basic ops**

先把输入linked list分裂成odd, even list，然后再把它们串联起来。

|  |
| --- |
| ListNode\* oddEvenList(ListNode\* head) {  ListNode sentinel\_odd(0), sentinel\_even(0);  ListNode \*tail\_odd = &sentinel\_odd;  ListNode \*tail\_even = &sentinel\_even;  for (int k = 1; head; k++, head = head->next) {  ListNode \*&tail = k%2==1? tail\_odd : tail\_even;  tail->next = head;  tail = tail->next;  }  tail\_odd->next = sentinel\_even.next;  tail\_even->next = NULL;  return sentinel\_odd.next;  } |

# 329. Longest Increrasing Path in a Matrix

|  |
| --- |
| Given an integer matrix, find the length of the longest increasing path.  From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed). |
| Example 1:  nums = [  [9,9,4],  [6,6,8],  [2,1,1]  ]  Return 4 The longest increasing path is [1, 2, 6, 9].  Example 2:  nums = [  [3,4,5],  [3,2,6],  [2,2,1]  ]  Return 4 The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed. |

**Solution: Memorized recursion**

从每个node出发DFS（后序），递归地计算从后继node出发的最长路径，以得到从当前node出发的最长路径。

|  |
| --- |
| **int longestIncreasingPath(vector<vector<int>>& matrix)** {  if (matrix.size()==0 || matrix[0].size()==0) return 0;  vector<vector<int>> lp(matrix.size(), vector<int>(matrix[0].size(), 0));  int maxl = 1;  for (int i = 0; i < matrix.size(); ++i) {  for (int j = 0; j < matrix[0].size(); ++j) {  maxl = max(maxl, longestIncreasingPath\_recur(matrix, lp, i, j));  }  }  return maxl;  }  int longestIncreasingPath\_recur(vector<vector<int>>& matrix, vector<vector<int>> &lp,  int i, int j) {  if (i < 0 || j < 0 || i>=matrix.size() || j>=matrix[0].size()) return 0;  if (lp[i][j] > 0) return lp[i][j];    int maxl = 1;  if (getVal(matrix, i-1, j) > matrix[i][j])  maxl = max(longestIncreasingPath\_recur(matrix, lp, i-1, j)+1, maxl);  if (getVal(matrix, i+1, j) > matrix[i][j])  maxl = max(longestIncreasingPath\_recur(matrix, lp, i+1, j)+1, maxl);  if (getVal(matrix, i, j-1) > matrix[i][j])  maxl = max(longestIncreasingPath\_recur(matrix, lp, i, j-1)+1, maxl);  if (getVal(matrix, i, j+1) > matrix[i][j])  maxl = max(longestIncreasingPath\_recur(matrix, lp, i, j+1)+1, maxl);  lp[i][j] = maxl;  return maxl;  }    inline int getVal(vector<vector<int>>& matrix, int i, int j) {  if (i < 0 || j < 0 || i>=matrix.size() || j>=matrix[0].size()) return INT\_MIN;  else return matrix[i][j];  } |

# 330. Patching Array

|  |
| --- |
| Given a sorted positive integer array nums and an integer n, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required. |
| Example 1:  nums = [1, 3], n = 6  Return 1.  Combinations of nums are [1], [3], [1,3], which form possible sums of: 1, 3, 4.  Now if we add/patch 2 to nums, the combinations are: [1], [2], [3], [1,3], [2,3], [1,2,3].  Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].  So we only need 1 patch.  Example 2:  nums = [1, 5, 10], n = 20  Return 2.  The two patches can be [2, 4].  Example 3:  nums = [1, 2, 2], n = 5  Return 0. |

**Solution: greedy**

最短patch数组要求对任意k

* 它的前k个数能构造连续范围[1, n]
* 前k个数中来自原数组的数尽量多。
* 它的前k个数能构造的范围尽量大。

假设patched数组前k个数能构造的范围是[1, n]。假设nums中下一个数为a

* a <= n: 最大范围更新为[1, n+a]
* 考虑加入1<=b<=n+1，从而使范围连续。根据上面第三条原则，b=n+1

|  |
| --- |
| int minPatches(vector<int>& nums, int n) {  long miss = 1, count = 0, i = 0;  while (miss <= n) {  if (i < nums.size() && nums[i] <= miss) miss += nums[i++];  else {  count++;  miss += miss;  }  }  return count;  } |

# 331. Verify Preorder Serialization of a Binary Tree

|  |
| --- |
| One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.  \_9\_  / \  3 2  / \ / \  4 1 # 6  / \ / \ / \  # # # # # #  For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.  Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.  Each comma separated value in the string must be either an integer or a character '#' representing null pointer.  You may assume that the input format is always valid, for example it could never contain two consecutive commas such as "1,,3". |
| Example 1: "9,3,4,#,#,1,#,#,2,#,6,#,#" Return true  Example 2: "1,#" Return false  Example 3: "9,#,#,1" Return false |

**Solution1: Graph theory (count degree)**

所有结点（除了根结点）入度为1，非叶结点(numbers)出度为2,叶子结点出度为0.在前序遍历时，当我们还未加入当前结点的出度，则边对应的出度都已经被count，但入度可能没有被count。所以入度和应该小于出度和。当遍历结束后，入度和应该等于出度和。

|  |
| --- |
| bool isValidSerialization(string preorder) {  int out\_degree = 0, in\_degree = -1; // total fan in and fan out  istringstream ss(preorder);  string token;  while (getline(ss, token, ',')) {  ++in\_degree;  // at each node, in degree >= out degree, when out degree of the node not counted  if (in\_degree > out\_degree) return false;  if (token[0] != '#') out\_degree += 2;  }  return (in\_degree == out\_degree);  } |

**Solution2: Graph theory (count leaf/nonleaf)**

二叉树非叶结点的个数＝叶结点个数-1。当前序遍历时，假设未被访问的右子树都是叶子，则非叶结点个数应该>叶结点个数-1，直到最后一个结点时，两者相等。

|  |
| --- |
| bool isValidSerialization(string preorder) {  int n\_null = 0, n\_num = 0; // total fan in and fan out  istringstream ss(preorder);  string token;  while (n\_null != n\_num + 1 && !ss.eof()) {  getline(ss, token, ',');  if (token[0] == '#') ++n\_null; else ++n\_num;  }  return ss.eof() && (n\_null == n\_num+1);  } |

# 332. Reconstruct Itinerary

|  |
| --- |
| Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.  Note:  If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].  All airports are represented by three capital letters (IATA code).  You may assume all tickets form at least one valid itinerary. |
| Example 1:  tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]  Return ["JFK", "MUC", "LHR", "SFO", "SJC"].  Example 2:  tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]  Return ["JFK","ATL","JFK","SFO","ATL","SFO"].  Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"]. But it is larger in lexical order. |

1

2

3

4

5

**Solution: post-order traversal**

把每个结点的后继按照字母序排序。并且DFS时先访问字母序小的后继，并删去已访问的边。但对多于一个后继的结点，意味着路径应该先从该结点开始绕若干圈，然后到达整个行程的尾端。

如右图，我们希望的访问顺序是：S A S C S B。

一种可能性是我们按照字母序：先后访问从A, B, C开始的路径，并检测到B所在路径不是一个环，因此把S到B的路径append到最后。但这样实现起来比较复杂。因此我们考虑另外一种“后序”递归方式：只有当一个结点的所有后继被访问完时才输出该结点。这样可以得到我们期望顺序的逆序。

例如右图的例子先后访问的path为: 1 2 3 (3) 4 5 (5) (4) (2) (1) (括号表示沿着边回退）。关键点在于从B回退到S后因为S还有未访问的后继，因此先访问S的后继。这样我们得到序列BSCSAS，再逆转以下这个序列即可。

递归版本：

|  |
| --- |
| vector<string> findItinerary(vector<pair<string, string>> tickets) {  unordered\_map<string, multiset<string>> adjacency;  for (pair<string, string> p : tickets) adjacency[p.first].insert(p.second);  vector<string> result;  findItinerary\_recur("JFK", adjacency, result);  reverse(result.begin(), result.end());  return result;  }  void findItinerary\_recur(string src, unordered\_map<string, multiset<string>>& adjacency, vector<string>& result) {  multiset<string> &next\_list = adjacency[src];  while (next\_list.size() > 0) {  string next = \*(next\_list.begin());  next\_list.erase(next\_list.begin());  findItinerary\_recur(next, adjacency, result);  }  result.push\_back(src);  } |

非递归版本：

|  |
| --- |
| vector<string> findItinerary(vector<pair<string, string>> tickets) {  unordered\_map<string, multiset<string>> adjacency;  for (pair<string, string> p : tickets) adjacency[p.first].insert(p.second);  vector<string> result;  stack<string> stk;  stk.push("JFK");    while (!stk.empty()) {  string src = stk.top();  multiset<string> &next\_list = adjacency[src];  if (next\_list.size() > 0) {  stk.push(\*next\_list.begin());  next\_list.erase(next\_list.begin());  }  else {  stk.pop();  result.push\_back(src);  }  }  reverse(result.begin(), result.end());  return result;  } |

# 333. Largest BST Subtree [Locked]

|  |
| --- |
| Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.  Note:  A subtree must include all of its descendants.  Here's an example:  10  / \  5 15  / \ \  1 8 7  The Largest BST Subtree in this case is the highlighted one.  The return value is the subtree's size, which is 3.  Hint:  You can recursively use algorithm similar to 98. Validate Binary Search Tree at each node of the tree, which will result in O(nlogn) time complexity.  Follow up:  Can you figure out ways to solve it with O(n) time complexity? |

**Solution: post-order traversal**

后序遍历。返回左右子树的大小、范围。同时还返回目前最大的BST.

|  |
| --- |
| int largestBSTSubtree(TreeNode\* root) {  if (root == NULL) return 0;  int max\_sz = 0;  largestBSTSubtree\_recur(root, max\_sz);  return max\_sz;  }  tuple<int, int, int, bool> largestBSTSubtree\_recur(TreeNode\* root, int &max\_sz) {  if (root == NULL) return make\_tuple(INT\_MAX, INT\_MIN, 0, true);    int minv1, minv2, maxv1, maxv2, n1, n2;  bool is\_bst1, is\_bst2, is\_bst;  tie(minv1, maxv1, n1, is\_bst1) = largestBSTSubtree\_recur(root->left, max\_sz);  tie(minv2, maxv2, n2, is\_bst2) = largestBSTSubtree\_recur(root->right, max\_sz);    is\_bst = is\_bst1 && is\_bst2 && (maxv1 <= root->val) && (minv2 >= root->val);  if (is\_bst) max\_sz = max(max\_sz, n1 + n2 + 1);    return make\_tuple(min(root->val, minv1), max(root->val, maxv2), n1+n2+1, is\_bst);  } |

# 334. Increasing Triplet Subsequence

|  |
| --- |
| Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.  Formally the function should:  Return true if there exists i, j, k  such that arr[i] < arr[j] < arr[k] given 0 ≤ i < j < k ≤ n-1 else return false.  Your algorithm should run in O(n) time complexity and O(1) space complexity. |
| Examples:  Given [1, 2, 3, 4, 5],  return true.  Given [5, 4, 3, 2, 1],  return false. |

**Solution: DP**

计算每个数字之前的最小数和之后的最大数。如果对某个位置这三个数字是单调增的，则返回true，否则返回false.

|  |
| --- |
| bool increasingTriplet(vector<int>& nums) {  if (nums.size()<3) return false;  vector<int> minv(nums.size(), 0);  minv[0] = nums[0];  for (int k = 1; k < nums.size(); ++k) minv[k] = min(minv[k-1], nums[k]);  int maxv = nums.back();  for (int k = nums.size()-1; k > 0; --k) {  if (maxv > nums[k] && nums[k] > minv[k-1]) return true;  maxv = max(maxv, nums[k]);  }  return false;  } |

# 335. Self Crossing

|  |
| --- |
| You are given an array x of n positive numbers. You start at point (0,0) and moves x[0] metres to the north, then x[1] metres to the west, x[2] metres to the south, x[3] metres to the east and so on. In other words, after each move your direction changes counter-clockwise.  Write a one-pass algorithm with O(1) extra space to determine, if your path crosses itself, or not. |
| Example 1:  Given x = [2, 1, 1, 2], Return true (self crossing)  Example 2:  Given x = [1, 2, 3, 4], Return false (not self crossing)  Example 3:  Given x = [1, 1, 1, 1], Return true (self crossing) |

**Solution: observe + modularization**

没有self crossing的路径有三种pattern:向外转（左一）、向内转（左二）、或向外转后再向内转。self crossing在向内转时发生。

向外转改为向内转的过程中可能产生self crossing的情况如右一、右二图。

|  |
| --- |
| **bool isSelfCrossing(vector<int>& x)** {  if (x.size() <= 3) return false;  int last = scanOuter(x);  if (last == x.size()-1) return false;  if (last >= 3 && last < x.size()-1) {  int prev = last < 4? 0 : x[last-4];  if (x[last]+prev<x[last-2] && x[last+1]>=x[last-1]) return true;  if (x[last]+prev>=x[last-2] && x[last+1]+x[last-3]>=x[last-1]) return true;  }    return scanInner(x, last+1);  }  int scanOuter(vector<int>& x) {  int k = 2;  while (k < x.size() && x[k] > x[k-2]) ++k;  return k;  }  bool scanInner(vector<int>& x, int k) {  while (k < x.size() && x[k] < x[k-2]) ++k;  return k < x.size();  } |

# 336. Palindrome Pairs

|  |
| --- |
| Given a list of unique words. Find all pairs of distinct indices (i, j) in the given list, so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome. |
| Example 1:  Given words = ["bat", "tab", "cat"]  Return [[0, 1], [1, 0]]  The palindromes are ["battab", "tabbat"]  Example 2:  Given words = ["abcd", "dcba", "lls", "s", "sssll"]  Return [[0, 1], [1, 0], [3, 2], [2, 4]]  The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"] |

**Solution: trie tree**

算法：假设palindrome pair为s1, s2，则：

(1)若两者一样长：s1,s2互为reverse  
(2)若s1较长：reverse(s2)为s1的前缀，剩余的s1后缀为回文。

(3)若s2较长：s1为reverse(s2)的前缀，剩余的reverse(s2)部分为回文。

Trie tree结构：因此我们可以构造一棵由所有word的reverse构成的trie树，每个前缀node保存一个list记录以这个前缀开头的单词里有哪些相应的后缀是回文，它本身如果是字典里的一个单词，相应的index（或-1，表示不是一个单词）。

这样，当我们lookup字典里的每个单词i，如果看到前缀是单词j的reverse，则检查自身剩下的后缀是不是回文，如果是则添加(i,j)。访问结束后，再把相应node存储的所有 list里的单词j对应的pair(i,j).

边界条件

* s1, s2长度相等的时候只应该被count一次，因此每个结点的list不包括这个节点本身。
* 剔除回文串导致添加(i, i)的情况。

|  |
| --- |
| struct TrieNode {  TrieNode\* children[26] = {NULL};  int idx; //word index or -1 if prefix is not a word  list<int> pchildren; //index of words = this prefix + some palindrome suffix    TrieNode(): idx(-1) {};  ~TrieNode() {  for (int k = 0; k < 26; ++k) delete children[k];  }  };  **vector<vector<int>> palindromePairs(vector<string>& words)** {  vector<vector<int>> result;  TrieNode \*root = new TrieNode();  for (int k = 0; k < words.size(); ++k) add(root, words[k], k);  for (int k = 0; k < words.size(); ++k) query(root, words[k], k, result);  delete root;  return result;  }  //lookup word in trie tree:  //- for each prefix id, if it is a word append (idx, id)  //- at node for word s(if it exists) for all words id in node's pchildren append (idx, id)  //- avoid adding (idx, idx)  inline void query(TrieNode \*root, string &s, int idx, vector<vector<int>> &result) {  TrieNode \*t = root;  int j = 0;  for (; t != NULL && j<=s.size(); ++j) {  if (t->idx>=0 && ispalindrome(s, j, s.size()-1))  result.push\_back(vector<int>{idx, t->idx});  if (j < s.size()) t = t -> children[s[j]-'a'];  }  if (t == NULL) return;  if (idx == t->idx) result.pop\_back(); //remove (i, i) for palindrome string  for (int id : t->pchildren) result.push\_back(vector<int>{idx, id});  }  //insert word into trie tree:  //- create relevant prefix nodes if necessary  //- add word index to prefix nodes where corresponding nonempty suffix is palindrome  //- set idx of word node as word index  inline void add(TrieNode \*root, string &s, int idx) {  TrieNode \*t = root;    for (int k = s.size()-1; k >= 0; --k) {  if (ispalindrome(s, 0, k)) t->pchildren.push\_back(idx);  char c = s[k];  if (t->children[c-'a'] == NULL) t->children[c-'a'] = new TrieNode();  t = t->children[c-'a'];  }  t->idx = idx;  }  inline bool ispalindrome(string &s, int l, int r) {  for (; l < r; l++, r--) if (s[l] != s[r]) return false;  return true;  } |

# 337. House Robber III

|  |
| --- |
| The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.  Determine the maximum amount of money the thief can rob tonight without alerting the police. |
| Example 1:  3  / \  2 3  \ \  3 1  Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.  Example 2:  3  / \  4 5  / \ \  1 3 1  Maximum amount of money the thief can rob = 4 + 5 = 9. |

**Solution: post-order traversal**

对每棵子树：递归地返回rob根结点和不rob根结点得到的最大收益。

|  |
| --- |
| int rob(TreeNode\* root) {  return rob\_recur(root).first;  }  // first: max sum including self; second: max sum not including self  pair<int, int> rob\_recur(TreeNode\* root) {  if (root == NULL) return make\_pair(0, 0);    pair<int, int> pl = rob\_recur(root->left);  pair<int, int> pr = rob\_recur(root->right);    pair<int, int> ret\_val;  ret\_val.second = pl.first + pr.first;  ret\_val.first = max(pl.second + pr.second + root->val, ret\_val.second);    return ret\_val;  } |

# 338. Counting Bits

|  |
| --- |
| Given a non negative integer number num. For every numbers i in the range 0 ≤ i ≤ num calculate the number of 1's in their binary representation and return them as an array.  Follow up:  It is very easy to come up with a solution with run time O(n\*sizeof(integer)). But can you do it in linear time O(n) /possibly in a single pass?  Space complexity should be O(n).  Can you do it like a boss? Do it without using any builtin function like \_\_builtin\_popcount in c++ or in any other language. |
| Example:  For num = 5 you should return [0,1,1,2,1,2]. |

**Solutoin1: iteration**

每增加一个bit，复制之前的sequence，并把相应的count加一。

|  |
| --- |
| vector<int> countBits1(int num) {  if (num == 0) return vector<int>{0};    vector<int> result({0, 1});  while (result.size()-1 < num) {  int len = min(result.size(), num - result.size() + 1);  vector<int> next(result.begin(), result.begin() + len);  for (int &v : next) result.push\_back(v+1);  }  return result;  } |

**Solution 2: DP**

result[i]＝result[i&(i-1)]+1

|  |
| --- |
| vector<int> countBits2(int num) {  vector<int> result(num+1, 0);  for (int k = 1; k <= num; ++k) result[k] = result[k & (k-1)] + 1;  return result;  } |

# 339. Nested List Weight Sum [Locked]

|  |
| --- |
| Given a nested list of integers, return the sum of all integers in the list weighted by their depth.  Each element is either an integer, or a list -- whose elements may also be integers or other lists. |
| Example 1:  Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)  Example 2:  Given the list [1,[4,[6]]], return 27. (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4\*2 + 6\*3 = 27) |

**Solution: recursion**

递归过程中传入depth即可。

|  |
| --- |
| int depthSum(vector<NestedInteger>& nestedList) {  return depthSum\_recur(nestedList, 1);  }  int depthSum\_recur(vector<NestedInteger>& nestedList, int depth) {  int sum = 0;  for (auto ni : nestedList)  sum += ni.isInteger()?  ni.getInteger() \* depth : depthSum\_recur(ni.getList(), depth+1);  return sum;  } |

# 340. Longest Substring with At Most K Distinct Characters [Locked]

|  |
| --- |
| Given a string, find the length of the longest substring T that contains at most k distinct characters.  For example, Given s = “eceba” and k = 2,  T is "ece" which its length is 3. |

**Solution: sliding window+set**

用一个数组maintain每个字母最后出现的位置。当数组刚出现k+1个字符时，假设数组最小数为s，则窗口从s+1开始。

|  |
| --- |
| int lengthOfLongestSubstringKDistinct(string s, int k) {  if (s.size() == 0) return 0;  array<int, 256> last\_pos;  last\_pos.fill(-1);  set<int> pos;    int len = 0;  for (int j = 0, start = -1; j < s.size(); ++j) {  char c = s[j];  if (last\_pos[c] >= 0) pos.erase(last\_pos[c]);  last\_pos[c] = j;  pos.insert(j);  if (pos.size() == k+1) {  start = \*(pos.begin());  char c\_del = s[start];  last\_pos[c\_del] = -1;  pos.erase(start);  }  len = max(len, j - start);  }  return len;  } |

# 341. Flatten Nested List Iterator

|  |
| --- |
| Given a nested list of integers, implement an iterator to flatten it.  Each element is either an integer, or a list -- whose elements may also be integers or other lists.  // This is the interface that allows for creating nested lists.  // You should not implement it, or speculate about its implementation  class NestedInteger {  public:  // Return true if this NestedInteger holds a single integer, rather than a nested list.  bool isInteger() const;  // Return the single integer that this NestedInteger holds, if it holds a single integer  // The result is undefined if this NestedInteger holds a nested list  int getInteger() const;  // Return the nested list that this NestedInteger holds, if it holds a nested list  // The result is undefined if this NestedInteger holds a single integer  const vector<NestedInteger> &getList() const;  }; |
| Example 1: Given the list [[1,1],2,[1,1]], By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].  Example 2: Given the list [1,[4,[6]]], By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,4,6]. |

**Solution: recursion with stack**

这个算法类似DFS访问一棵树，用一个stack维护待展开的结点。

|  |
| --- |
| **class NestedIterator {**  public:  **NestedIterator(vector<NestedInteger> &nestedList)** {  for (int k = nestedList.size()-1; k>=0; --k) stk.push(nestedList[k]);  }  **int next()** {  NestedInteger ni = stk.top();  stk.pop();  return ni.getInteger();  }  **bool hasNext()** {  while (!stk.empty() && !stk.top().isInteger()) {  const vector<NestedInteger> nestedList = stk.top().getList();  stk.pop();  for (int k = nestedList.size()-1; k>=0; --k) stk.push(nestedList[k]);  }  return !stk.empty();  }    private:  stack<NestedInteger> stk;  }; |

# 342. Power of Four

|  |
| --- |
| Given an integer (signed 32 bits), write a function to check whether it is a power of 4. |
| Example:  Given num = 16, return true. Given num = 5, return false.  Follow up: Could you solve it without loops/recursion? |

**Solution: bit-trick**

首先确定n的偶数bit都是0,然后确定n是2的整数次幂。

|  |
| --- |
| **bool isPowerOfFour(int num)** {  return (num > 0) && (num & 0x55555555) == num && (num & (num-1)) == 0;  } |

# 343. Integer Break

|  |
| --- |
| Given a positive integer n, break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.  Note: you may assume that n is not less than 2. |
| For example, given n = 2, return 1 (2 = 1 + 1); given n = 10, return 36 (10 = 3 + 3 + 4). |

**Solution:static DP**

用P[n]记录n的integer break。因为问题和输入基本无关，因此可以预存计算过的P[n]

|  |
| --- |
| int integerBreak(int n) {  static vector<int> prod({0,0,1});  if (n+1 <= prod.size()) return prod[n];    for (int k = prod.size(); k <= n; ++k) {  int pk = 0;  for (int j = 1; j<=k-j; ++j)  pk = max(max(max(pk, prod[k-j]\*j), prod[j]\*(k-j)), (k-j)\*j);  prod.push\_back(pk);  }  return prod.back();  } |

# 344. Reverse String

|  |
| --- |
| Write a function that takes a string as input and returns the string reversed. |
| Example:  Given s = "hello", return "olleh". |

**Solution: two-pointer**

用两个指针从字符串两段向中间扫描。同时把这两个指针对应的字符对换。

|  |
| --- |
| string reverseString(string s) {  for (int l = 0, r = s.size()-1; l < r; ++l, --r) swap(s[l], s[r]);  return s;  } |

# 345. Reverse Vows of a String

|  |
| --- |
| Write a function that takes a string as input and reverse only the vowels of a string. |
| Example 1:  Given s = "hello", return "holle".  Example 2:  Given s = "leetcode", return "leotcede". |

**Solution: two-pointer**

类似344题，只是指针每次只指向aeiou等字母。

|  |
| --- |
| string reverseVowels(string s) {  bool isvowel[256] = {false};  isvowel['a'] = isvowel['e'] = isvowel['i'] = isvowel['o'] = isvowel['u'] = true;  isvowel['A'] = isvowel['E'] = isvowel['I'] = isvowel['O'] = isvowel['U'] = true;    for (int l = 0, r = s.size()-1; l < r; ++l, --r) {  while (l < r && !isvowel[s[l]]) ++l;  while (l < r && !isvowel[s[r]]) --r;  swap(s[l], s[r]);  }  return s;  } |

# 346. Moving Average from Data Stream [Lock]

|  |
| --- |
| Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.  For example,  MovingAverage m = new MovingAverage(3);  m.next(1) = 1  m.next(10) = (1 + 10) / 2  m.next(3) = (1 + 10 + 3) / 3  m.next(5) = (10 + 3 + 5) / 3 |

**Solution: queue**

维护当前的sum和count。以及用一个queue存放sliding wondow里的内容

|  |
| --- |
| class MovingAverage {  public:  MovingAverage(int size) : sz(size) {}    double next(int val) {  if (window.size() == sz) {  sum -= window.front();  window.pop();  }  sum += val;  window.push(val);  return (double)sum/(double)(window.size());  }    private:  int sum = 0, sz;  queue<int> window;  }; |

# 347. Top K Frequent Elements

|  |
| --- |
| Given a non-empty array of integers, return the k most frequent elements.  Note:  You may assume k is always valid, 1 ≤ k ≤ number of unique elements.  Your algorithm's time complexity must be better than O(n log n), where n is the array's size. |
| For example,  Given [1,1,1,2,2,3] and k = 2, return [1,2]. |

**Solution1: unordered\_map + heap**

首先记录直方图hist。然后取hist的前k个元素。

用一个大小为k的heap/priority queue。复杂度为O(nlogk).

|  |
| --- |
| vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> hist;  for (int v : nums) ++hist[v];    priority\_queue<pair<int, int>> freq;  for (auto it = hist.begin(); it != hist.end(); it++) {  freq.push(make\_pair(-it->second, it->first));  if (freq.size() > k) freq.pop();  }    vector<int> result(k, 0);  for (int j = 0; j < k; ++j) {  result[j] = freq.top().second;  freq.pop();  }  return result;  } |

**Solution 2: unordered\_map + map**

用一个map<int, vector<int>>存储frequency为k的数组。时间为O(nlogm)，m为unique char的个数。

|  |
| --- |
| vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> hist;  for (int v : nums) ++hist[v];    map<int, vector<int>> items;  for (auto it = hist.begin(); it != hist.end(); ++it)  items[it->second].push\_back(it->first);    vector<int> result;  for (auto it = items.rbegin(); it != items.rend() && result.size() < k; ++it)  result.insert(result.end(), it->second.begin(), it->second.end());  return result;  } |

# 348. Design Tic-Tac-Toe [Locked]

|  |
| --- |
| Design a Tic-tac-toe game that is played between two players on a n x n grid.  You may assume the following rules:   * A move is guaranteed to be valid and is placed on an empty block. * Once a winning condition is reached, no more moves is allowed. * A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.   Example: Given n = 3, assume that player 1 is "X" and player 2 is "O" in the board.  TicTacToe toe(3);  toe.move(0, 0, 1); // Returns 0 (no one wins)  toe.move(0, 2, 2); // Returns 0 (no one wins)  toe.move(2, 2, 1); // Returns 0 (no one wins)  toe.move(1, 1, 2); // Returns 0 (no one wins)  toe.move(2, 0, 1); // Returns 0 (no one wins)  toe.move(1, 0, 2); // Returns 0 (no one wins)  toe.move(2, 1, 1); // Returns 1 (player 1 wins)  |X| | | |X| |O| |X| |O| |X| |O| |X| |O| |X| |O| |X| |O|  | | | | | | | | | | | | | |O| | | |O| | |O|O| | |O|O| |  | | | | | | | | | | |X| | | |X| |X| |X| |X| |X| |X|X|X|  Follow up: Could you do better than O(n2) per move() operation? |

**Solution: bit sets**

类似8皇后，维护每行、每列、每对角线x每个player的count。

|  |
| --- |
| class TicTacToe {  private:  const int N;  vector<vector<char>> board;  vector<int> rows[2], cols[2];  int diag1[2] = {0}, diag2[2] = {0};  int last\_player = -1;  bool finish = false;    public:  //assumes n is valid: n > 0  TicTacToe (int n) : N(n), board(N, vector<char>(N, ' ')) {  rows[0].resize(N, 0), rows[1].resize(N, 0);  cols[0].resize(N, 0), cols[1].resize(N, 0);  };    int move(int row, int col, int player) {  //make sure input is valid  last\_player = player;    board[row][col] = '0' + player;  rows[player-1][row]++;  cols[player-1][col]++;  if (row == col) diag1[player-1]++;  if (row == N-1-col) diag2[player-1]++;  if (rows[player-1][row] == N || cols[player-1][col] == N ||  diag1[player-1] == N || diag2[player-1] == N) finish = true;    return finish? player : 0;  }  }; |

# 349. Intersection of Two Arrays

|  |
| --- |
| Given two arrays, write a function to compute their intersection.  Note:  Each element in the result must be unique.  The result can be in any order. |
| Example:  Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2]. |

**Solution: unordered\_set**

用一个unordered\_set记录nums1中的数。对nums2中的每个数，检查是否在set中

|  |
| --- |
| vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {  unordered\_set<int> history(nums1.begin(), nums1.end());  vector<int> result;  for (int v : nums2) if (history.erase(v)) result.push\_back(v);  return result;  } |

# 350. Intersection of Two Arrays II

|  |
| --- |
| Given two arrays, write a function to compute their intersection.  Example:  Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].  Note:  Each element in the result should appear as many times as it shows in both arrays.  The result can be in any order.  Follow up:  What if the given array is already sorted? How would you optimize your algorithm?  What if nums1's size is small compared to nums2's size? Which algorithm is better?  What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once? |

**Solution 1: unordered\_multiset**

用一个unordered\_multiset存nums1中的元素。

|  |
| --- |
| vector<int> intersect1(vector<int>& nums1, vector<int>& nums2) {  unordered\_multiset<int> history(nums1.begin(), nums1.end());  vector<int> result;  for (int v : nums2) {  auto it = history.find(v);  if (it != history.end()) {  history.erase(it);  result.push\_back(v);  }  }  return result;  } |

**Solution 2: merge sort**

对两个数组分别排序。然后用类似merge sort的方法找到成对的元素。

|  |
| --- |
| vector<int> intersect2(vector<int>& nums1, vector<int>& nums2) {  sort(nums1.begin(), nums1.end());  sort(nums2.begin(), nums2.end());  vector<int> result;  for (int i = 0, j = 0; i < nums1.size() && j < nums2.size(); ++i, ++j) {  if (nums1[i]==nums2[j]) result.push\_back(nums1[i]);  else if (nums1[i] < nums2[j]) --j;  else --i;  }  return result;  } |

# 