# 101. Symmetric Tree

|  |
| --- |
| Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center). |
| For example, this binary tree [1,2,2,3,4,4,3] is symmetric:  1  / \  2 2  / \ / \  3 4 4 3  But the following [1,2,2,null,3,null,3] is not:  1  / \  2 2  \ \  3 3  Note:  Bonus points if you could solve it both recursively and iteratively. |

**Sol1：recursion**

判断是否root为空，或递归是否左子树t1为右子树t2的镜像（即两者根结点都为空或值相等且t1的左子树的镜像为t2右子树，且t1右子树也为t2左子树镜像）。

|  |
| --- |
| **bool isSymmetric(TreeNode\* root)** {  return !root || isReflection(root->left, root->right);  }  bool isReflection(TreeNode \*t1, TreeNode \*t2) {  if (!t1 && !t2) return true;  if ((!t1 && t2) || (t1 && !t2) || (t1->val != t2->val)) return false;  return isReflection(t1->left, t2->right) && isReflection(t1->right, t2->left);  } |

**Sol2: iteration with stack**

遍历左子树，同时维护右子树中的镜像结点，直到发现冲突（返回false)，或遍历结束(返回true)。

|  |
| --- |
| **bool isSymmetric(TreeNode\* root)** {  if (!root) return true;  stack<pair<TreeNode \*, TreeNode \*>> stk;  stk.push(make\_pair(root->left, root->right));    while (!stk.empty()) {  TreeNode \*t1 = stk.top().first;  TreeNode \*t2 = stk.top().second;  stk.pop();  if (!t1 && !t2) continue;  if ((t1 && !t2) || (t2 && !t1) || t1->val != t2->val) return false;  stk.push(make\_pair(t1->left, t2->right));  stk.push(make\_pair(t1->right, t2->left));  }  return true;  } |

# 102. Binary Tree Level Order

|  |
| --- |
| Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level). |
| For example:  Given binary tree [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7  return its level order traversal as:  [  [3],  [9,20],  [15,7]  ] |

**Sol:BFS**

维护pre: 当前层结点和cur:下一层结点

外层while loop用于切换到下一层

|  |
| --- |
| **vector<vector<int>> levelOrder(TreeNode\* root)** {  vector<vector<int>> result;  vector<TreeNode \*> pre;  if (root == NULL) return result;  else {  result.push\_back(vector<int>{root->val});  pre.push\_back(root);  }  while (!pre.empty()) {  vector<TreeNode \*> cur;  vector<int> cur\_val;  for (TreeNode \*t : pre) {  if (t->left) {  cur.push\_back(t->left);  cur\_val.push\_back(t->left->val);  }  if (t->right) {  cur.push\_back(t->right);  cur\_val.push\_back(t->right->val);  }  }  swap(pre, cur);  if (!cur\_val.empty()) result.push\_back(cur\_val);  }  return result;  } |

# 103. Binary Tree Zigzag Level Order Traversal

|  |
| --- |
| Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).  For example:  Given binary tree [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7  return its zigzag level order traversal as:  [  [3],  [20,9],  [15,7]  ] |

**Sol:BFS**

修改上述算法，用一个bool控制是否当前行需要先逆转再输入result。

|  |
| --- |
| **vector<vector<int>> zigzagLevelOrder(TreeNode\* root)** {  vector<vector<int>> result;  vector<TreeNode \*> pre;  if (root == NULL) return result;  else {  result.push\_back(vector<int>{root->val});  pre.push\_back(root);  }    bool rev = true;  for (; !pre.empty(); rev = !rev) {  vector<TreeNode \*> cur;  vector<int> cur\_val;  for (TreeNode \*t : pre) {  if (t->left) {  cur.push\_back(t->left);  cur\_val.push\_back(t->left->val);  }  if (t->right) {  cur.push\_back(t->right);  cur\_val.push\_back(t->right->val);  }  }  swap(pre, cur);  if (!cur\_val.empty()) {  if (rev) reverse(cur\_val.begin(), cur\_val.end());  result.push\_back(cur\_val);  }  }  return result;  } |

# 104. Maximum Depth of Binary Tree

|  |
| --- |
| Given a binary tree, find its maximum depth.  The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node. |

**Sol: recursion**

递归结构：返回当前root对应子树的高度(为左右子树高度较大值＋1)。

递归结束条件：root==NULL。

|  |
| --- |
| **int maxDepth(TreeNode\* root)** {  return (root == NULL)? 0 : max(maxDepth(root->left), maxDepth(root->right)) + 1;  } |

# 105. Construct Binary Tree from Preorder and Inorder Traversal

|  |
| --- |
| Given preorder and inorder traversal of a tree, construct the binary tree.  Note:  You may assume that duplicates do not exist in the tree |

**Sol: recursion**

树的根结点为preorder[0]。在inorder中找到preorder[0]则把数组分为两半，一半对应左子树，一半对应右子树。由inorder的划分可以得到preorder得划分（因为子数组长度相同，排列不同）。因此可以递归构造树。

|  |
| --- |
| **TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder)** {  return buildTree\_recur(preorder, inorder, 0, preorder.size(), 0, inorder.size());  }  TreeNode\* buildTree\_recur(const vector<int>& preorder, const vector<int>& inorder,  int l1, int r1, int l2, int r2) {  if (l1 == r1) return NULL;  TreeNode \*root = new TreeNode(preorder[l1]);  size\_t mid2 = find(inorder.begin()+l2, inorder.begin()+r2, preorder[l1])  - inorder.begin();  root->left = buildTree\_recur(preorder, inorder, l1+1, l1+mid2-l2+1, l2, mid2);  root->right = buildTree\_recur(preorder, inorder, l1+mid2-l2+1, r1, mid2+1, r2);  return root;  } |

# 106. Construct Binary Tree from Inorder and Postorder Traversal

|  |
| --- |
| Given inorder and postorder traversal of a tree, construct the binary tree.  Note:  You may assume that duplicates do not exist in the tree. |

**Sol: recursion**

和前一题一样，只是根结点是后序遍历的最后一个数

|  |
| --- |
| **TreeNode\* buildTree(vector<int>& inorder, vector<int>& postorder)** {  return buildTree\_recur(inorder, postorder, 0, inorder.size(), 0, postorder.size());  }  TreeNode\* buildTree\_recur(const vector<int>& inorder, const vector<int>& postorder,  int l1, int r1, int l2, int r2) {  if (l1 == r1) return NULL;  TreeNode \*root = new TreeNode(postorder[r2-1]);  size\_t mid1 = find(inorder.begin()+l1, inorder.begin()+r1, postorder[r2-1])  - inorder.begin();  root->left = buildTree\_recur(inorder, postorder, l1, mid1, l2, l2+mid1-l1);  root->right = buildTree\_recur(inorder, postorder, mid1+1, r1, l2+mid1-l1, r2-1);  return root;  } |

# 107. Binary Tree Level Order

|  |
| --- |
| Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).  For example:  Given binary tree [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7  return its bottom-up level order traversal as:  [  [15,7],  [9,20],  [3]  ] |

**Sol: BFS**

计算出Binary Tree Level Order的结果再reverse一下。

|  |
| --- |
| **vector<vector<int>> levelOrderBottom(TreeNode\* root)** {  vector<vector<int>> result;  vector<TreeNode \*> pre;  if (root == NULL) return result;  else {  result.push\_back(vector<int>{root->val});  pre.push\_back(root);  }  while (!pre.empty()) {  vector<TreeNode \*> cur;  vector<int> cur\_val;  for (TreeNode \*t : pre) {  if (t->left) {  cur.push\_back(t->left);  cur\_val.push\_back(t->left->val);  }  if (t->right) {  cur.push\_back(t->right);  cur\_val.push\_back(t->right->val);  }  }  swap(pre, cur);  if (!cur\_val.empty()) result.push\_back(cur\_val);  }  reverse(result.begin(), result.end());  return result;  } |

# 108. Convert Sorted Array to Binary Search Tree

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

**Sol: recursion**

递归结构：构造子数组nums[l]...nums[r-1]对应的BST：以子数组的中位数为根，前一半为左子树，后一半为右子树。

结束条件: l = r (子树为空)

|  |
| --- |
| **TreeNode\* sortedArrayToBST(vector<int>& nums)** {  return sorted\_array\_to\_bst\_recur(nums, 0, nums.size());  }  TreeNode\* sorted\_array\_to\_bst\_recur(const vector<int>& nums, int l, int r) {  if (l == r) return NULL;  int mid = (l+r)/2;  TreeNode \*root = new TreeNode(nums[mid]);  root->left = sorted\_array\_to\_bst\_recur(nums, l, mid);  root->right = sorted\_array\_to\_bst\_recur(nums, mid+1, r);  return root;  } |

# 109. Convert Sorted List to Binary Search Tree

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

**Sol: recursion**

每次链表长度折半。为了不反复遍历整个链表，先走一遍以存放长度

|  |
| --- |
| **TreeNode\* sortedListToBST(ListNode\* head)** {  int count = 0;  for (ListNode \*p = head; p; p = p->next) count++;  return sortedListToBST\_Recur(head, count);  }  TreeNode\* sortedListToBST\_Recur(ListNode \*head, int count) {  if (count == 0) return NULL; //recursion ends with empty list    //determine mid position  int mid = count/2;  ListNode sentinel(0); sentinel.next = head;  ListNode \*p = head, \*t = &sentinel;  for (int k = 0; k < mid; ++k, t = t->next, p = p->next);  t->next = NULL; //split list into two parts  TreeNode \*root = new TreeNode(p->val);  root->left = sortedListToBST\_Recur(head, mid);  root->right = sortedListToBST\_Recur(p->next, count - mid - 1);  t->next = p; //restore linked list to avoid dangling pointers  return root;  } |

# 110. Balanced Binary Tree

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

**Sol: recursion**

修改之前求树深度的递归解。只是在递归过程中判断一下是否平衡的条件满足。

|  |
| --- |
| **bool isBalanced(TreeNode\* root)** {  return isBalanced\_recur(root) >= 0;  }  int isBalanced\_recur(TreeNode \*root) {  if (!root) {  return 0;  }  else {  int hl = isBalanced\_recur(root->left);  int hr = isBalanced\_recur(root->right);  return (hl < 0 || hr < 0 || abs(hr-hl) > 1)? -1 : (max(hl, hr)+1);  }  } |

# 111. Minimum Depth of Binary Tree

|  |
| --- |
| Given a binary tree, find its minimum depth.  The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. |

**Sol: recursion**

自上而下计算每个结点的深度。当当前结点为根结点时，更新min\_depth.

|  |
| --- |
| **int minDepth(TreeNode\* root)** {  if (!root) return 0;  int min\_depth = INT\_MAX;  minDepth\_recur(root, 0, min\_depth);  return min\_depth;  }  void minDepth\_recur(TreeNode\* root, int depth, int &min\_depth) {  if (!root->left && !root->right) min\_depth = min(min\_depth, depth+1);  if (root->left) minDepth\_recur(root->left, depth+1, min\_depth);  if (root->right) minDepth\_recur(root->right, depth+1, min\_depth);  } |

# 112. Path Sum

|  |
| --- |
| Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. |
| For example:  Given the below binary tree and sum = 22,  5  / \  4 8  / / \  11 13 4  / \ \  7 2 1  return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22. |

**Sol: recursion**

递归结构：某个子树存在path sum = sum :

(1) 左子树非空且存在path sum == sum - root->val

(2) 右子树非空且存在path sum == sum - root->val

递归结束条件：当前root是叶子时，判断是否sum==0

|  |
| --- |
| **bool hasPathSum(TreeNode\* root, int sum)** {  return root? hasPathSum\_recur(root, sum):false;  }  bool hasPathSum\_recur(TreeNode \*root, int sum) {  sum -= root->val;  if (!root->left && !root->right) return sum==0;  return (root->left && hasPathSum\_recur(root->left, sum))  || (root->right && hasPathSum\_recur(root->right, sum));  } |

# 113. Path Sum II

|  |
| --- |
| Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum. |
| For example:  Given the below binary tree and sum = 22,  5  / \  4 8  / / \  11 13 4  / \ / \  7 2 5 1  return  [  [5,4,11,2],  [5,8,4,5]  ] |

**Sol: recursion**

上面递归的interface增加两个参数即可:

- vector<int> &prefix:从根到当前root的路径

- vector<vector<int>> &result: 返回值

|  |
| --- |
| **vector<vector<int>> pathSum(TreeNode\* root, int sum)** {  vector<vector<int>> result;  vector<int> prefix;  if (root) pathSum\_recur(root, sum, prefix, result);  return result;  }  void pathSum\_recur(TreeNode \*root, int sum, vector<int> &prefix, vector<vector<int>> &result) {  sum -= root->val;  prefix.push\_back(root->val);  if (!root->left && !root->right && sum==0) result.push\_back(prefix);  if (root->left) pathSum\_recur(root->left, sum, prefix, result);  if (root->right) pathSum\_recur(root->right, sum, prefix, result);  prefix.pop\_back();  } |

# 114. Flatten Binary Tree

|  |
| --- |
| Given a binary tree, flatten it to a linked list in-place.  For example,  Given  1  / \  2 5  / \ \  3 4 6  The flattened tree should look like:  1  \  2  \  3  \  4  \  5  \  6 |

**Sol: preorder tree traversal**

非递归解比较好写。

|  |
| --- |
| **void flatten(TreeNode\* root)** {  stack<TreeNode\*> stk;  TreeNode sentinel(0);  TreeNode \*tail = &sentinel;  stk.push(root);  while (!stk.empty()) {  root = stk.top();  stk.pop();  if (root) {  stk.push(root->right);  stk.push(root->left);  root->left = NULL;  root->right = NULL;  tail->right = root;  tail = root;  }  }  root = sentinel.right;  } |

# 115. Distinct Subsequence

|  |
| --- |
| Given a string S and a string T, count the number of distinct subsequences of T in S.  A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not). |
| Here is an example:  S = "rabbbit", T = "rabbit"  Return 3. |

**Sol1: recursion**

每次考虑s[0...i]是否包含t[0...j]，它分解为s[i]是否对应t[j]两种可能性。

|  |
| --- |
| **int numDistinct(string s, string t)** {  int count = 0;  numDistinct\_recur(s, t, 0, 0, count);  return count;  }  void numDistinct\_recur(const string &s, const string&t, int is, int it, int &count) {  if (is == s.size() || it == t.size() || t.size()-it > s.size()-is) return;    numDistinct\_recur(s, t, is+1, it, count);  if (s[is] == t[it]) {  if (it == t.size()) ++count;  else numDistinct\_recur(s, t, is+1, it+1, count);  }  } |

**Sol2: DP**

Sol1的问题在于递归时is=i, it=j将被多次调用，而它增加的count是一样多的。

用C[i][j]扫描s[0...i]包含t[0...j]的所有可能性。

|  |
| --- |
| **int numDistinct(string s, string t)** {  if (t.size() == 0) return 0;    vector<int> count(t.size(), 0);  count[0] = s[0] == t[0]?1:0;    for (int i = 1; i < s.size(); ++i) {  int pre = 1;  for (int j = 0; j <= i && j < t.size(); ++j) {  int last = count[j];  count[j] += (s[i] == t[j]? pre:0);  pre = last;  }  }  return count.back();  } |

# 116. Populating Next Right Pointers in Each Node

|  |
| --- |
| Given a binary tree  struct TreeLinkNode {  TreeLinkNode \*left;  TreeLinkNode \*right;  TreeLinkNode \*next;  };  Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.  Initially, all next pointers are set to NULL.  Note:  You may only use constant extra space.  You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).  For example,  Given the following perfect binary tree,  1  / \  2 3  / \ / \  4 5 6 7  After calling your function, the tree should look like:  1 -> NULL  / \  2 -> 3 -> NULL  / \ / \  4->5->6->7 -> NULL |

**Sol: BFS.**

在每一层类似连接链表一样把treeNode通过next field连接起来。用sentinel Node可以简化代码。

|  |
| --- |
| **void connect(TreeLinkNode \*root)** {  for (TreeLinkNode \*head; head; head = head->left) {  TreeLinkNode sentinel(0);  for (TreeLinkNode \*t = head, \*tail = &sentinel; t; t=t->next) {  if (!t->left) break;  tail->next = t->left;  t->left->next = t->right;  tail = t->right;  }  }  } |

# 117. Populating Next Right Pointers in Each Node II

|  |
| --- |
| Follow up for problem "Populating Next Right Pointers in Each Node".  What if the given tree could be any binary tree? Would your previous solution still work?  Note:  You may only use constant extra space.  For example,  Given the following binary tree,  1  / \  2 3  / \ \  4 5 7  After calling your function, the tree should look like:  1 -> NULL  / \  2 -> 3 -> NULL  / \ \  4-> 5 -> 7 -> NULL |

**Sol: BFS**

和上题基本一样

关键是head = head->left变为head = sentinel->next.以及跳过空的子树。

|  |
| --- |
| **void connect(TreeLinkNode \*root)** {  for (TreeLinkNode \*head = root; head;) {  TreeLinkNode sentinel(0);  for (TreeLinkNode \*t = head, \*tail = &sentinel; t; t=t->next) {  if (t->left) {  tail->next = t->left;  tail = t->left;  }  if (t->right) {  tail->next = t->right;  tail = t->right;  }  }  head = sentinel.next;  }  } |

# 118. Pascal's Triangle

|  |
| --- |
| Given numRows, generate the first numRows of Pascal's triangle. |
| For example, given numRows = 5,  Return  [  [1],  [1,1],  [1,2,1],  [1,3,3,1],  [1,4,6,4,1]  ] |

**Sol: simple for-loops**

|  |
| --- |
| **vector<vector<int>> generate(int numRows)** {  vector<vector<int>> result;  for (int k = 0; k < numRows; ++k) {  if (k==0) result.push\_back(vector<int>{1});  else {  vector<int> cur(1, 1);  vector<int> &last = result.back();  for (int j = 0; j < last.size()-1; ++j) {  cur.push\_back(last[j] + last[j+1]);  }  cur.push\_back(1);  result.push\_back(cur);  }  }  return result;  } |

# 119. Pascal Triangle II

|  |
| --- |
| Given an index k, return the kth row of the Pascal's triangle.  For example, given k = 3,  Return [1,3,3,1].  Note:  Could you optimize your algorithm to use only O(k) extra space? |

**Sol: simple for loop**

只需要保存最后两行即可（其实一行也行）。

|  |
| --- |
| **vector<int> getRow(int rowIndex)** {  if (rowIndex == 0) return vector<int>{1};  vector<int> cur(1, 1);  for (int k = 1; k <= rowIndex; ++k) {  cur.push\_back(1);//cur has k\*2+1 elements  for (int j = k-1; j>0; --j) cur[j] += cur[j-1];  cur[0] = 1;  }  return cur;  } |

# 120. Triangle

|  |
| --- |
| Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below. |
| For example, given the following triangle  [  [2],  [3,4],  [6,5,7],  [4,1,8,3]  ]  The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).  Note:  Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle. |

**Sol: simple for loop**

用矩阵sum存储到每行每个元素的最小值。但只需要维护最后两行数组。

|  |
| --- |
| **int minimumTotal(vector<vector<int>>& triangle)** {  if (triangle.size() == 0) return 0;  vector<int> sum(triangle.size(), INT\_MAX);  sum[0] = triangle[0][0];    for (int k = 1; k < triangle.size(); ++k) {  for (int j = k; j >0; --j) {  sum[j] = min(sum[j], sum[j-1]) + triangle[k][j];  }  sum[0] += triangle[k][0];  }  int minpath = INT\_MAX;  for (int k = 0; k < sum.size(); ++k) minpath = min(minpath, sum[k]);  return minpath;  } |

# 121. Best Time to Buy and Sell Stock

|  |
| --- |
| Say you have an array for which the ith element is the price of a given stock on day i.  If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit. |

**Sol: DP**

第k天完成交易的最大profit是price[k] - min(price[0...k])。因此维护到每一天的最低价格和最大利润即可。

|  |
| --- |
| **int maxProfit(vector<int>& prices)** {  int minp = INT\_MAX, max\_profit = INT\_MIN;  for (int p : prices) {  minp = min(minp, p);  max\_profit = max(p - minp, max\_profit);  }  return prices.size() == 0? 0:max\_profit;  } |

# 122. Best Time to Buy and Sell Stock II

|  |
| --- |
| 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). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again). |

**Sol: Divide and conquer**

一个n天的交易可以拆成每天买、每天卖的交易。因此只需要看单天的利润（价格差）是否大于0即可。

|  |
| --- |
| **int maxProfit(vector<int>& prices)** {  int max\_profit = 0;  for (int k = 1; k < prices.size(); ++k)  max\_profit += max(0, prices[k] - prices[k-1]);  return max\_profit;  } |

# 123. Best Time to Buy and Sell Stock III

|  |
| --- |
| 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 at most two transactions.  Note:  You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again). |

**Sol:DP**

前面的第121题我们计算了到第k天为止的最大利益。类似地如果我们从右向左扫描也可以计算从第k天开始的最大利益（通过维护第k天开始的最高价格）。这样扫描两遍即可得到两次交易的最大利益。

|  |
| --- |
| **int maxProfit(vector<int>& prices)** {  vector<int> first\_prof(prices.size(), 0);  int minp = INT\_MAX, maxp = 0;    for (int n = 0; n < prices.size(); ++n) {  minp = min(minp, prices[n]);  first\_prof[n] = n==0? 0 : max(first\_prof[n-1], prices[n] - minp);  }    int max\_prof = 0;  for (int n = prices.size()-1; n>=0; --n) {  maxp = max(maxp, prices[n]);  max\_prof = max(max\_prof, first\_prof[n] + maxp - prices[n]);  }  return max\_prof;  } |

# 124. Binary Tree Maximum Path Sum

|  |
| --- |
| Given a binary tree, find the maximum path sum.  For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.  For example:  Given the below binary tree,  1  / \  2 3  Return 6. |

**Sol:Post-order travesal**

递归结构:寻找从每个子结点到它叶子结点的最大path

(1) 向左右子结点传根到自己的path sum

(2) 递归计算左右子树max path sum

(3) 返回两者间较大值＋root->val

递归结束条件：叶子结点，此时返回max path sum=root->val.

|  |
| --- |
| **int maxPathSum(TreeNode\* root)** {  if (!root) return 0;  int max\_path = INT\_MIN;  maxPathSum\_recur(root, max\_path);  return max\_path;  }  int maxPathSum\_recur(TreeNode\* root, int &max\_path) {  if (!root->left && !root->right) {  max\_path = max(max\_path, root->val);  return root->val;  }    int lp = root->left==NULL? 0: maxPathSum\_recur(root->left, max\_path);  int rp = root->right==NULL? 0: maxPathSum\_recur(root->right, max\_path);  max\_path = max(max\_path, max(lp, 0) + max(rp, 0)+root->val);  return max(max(lp, 0)+root->val, max(rp, 0)+root->val);  } |

# 125. Valid Palindrome

|  |
| --- |
| Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.  For example,  "A man, a plan, a canal: Panama" is a palindrome.  "race a car" is not a palindrome. |

**Corner case:** 假设字符串中只有0-1个字母／数字字符，怎么处理？（这里假设仍然返回true）

**Sol:two pointer**

两个指针分别从左、右扫描对应的字符进行比较。跳过所有非字母／数字字符，并统一把字符转化为小写字母。

|  |
| --- |
| **bool isPalindrome(string s)** {  for (int l = 0, r = s.size()-1; l < r; ++l, --r) {  while (l < r && !isalpha(s[l]) &&!isdigit(s[l])) ++l;  while (l < r && !isalpha(s[r]) &&!isdigit(s[r])) --r;  if (tolower(s[l]) != tolower(s[r])) return false;  }  return true;  } |

# 126. Word Ladder II

|  |
| --- |
| Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:  - Only one letter can be changed at a time  - Each intermediate word must exist in the word list  Note:  - All words have the same length.  - All words contain only lowercase alphabetic characters. |
| For example, given:  beginWord = "hit"  endWord = "cog"  wordList = ["hot","dot","dog","lot","log"]  Return  [  ["hit","hot","dot","dog","cog"],  ["hit","hot","lot","log","cog"]  ] |

**Sol: BFS+backtracking**

见127 wordLadder的求ladder长度算法，本题同样使用（单向）BFS，然后回溯结果。

用一个unordered\_map<string, WordLabel>标记从beginWord到endWord的路径上，每个word可能的上一步。

|  |
| --- |
| struct WordLabel {  int level;  vector<string> prev;  WordLabel(): level(-1){};  };  **vector<vector<string>> findLadders(string beginWord, string endWord,**  **unordered\_set<string> &wordList)** {  unordered\_map<string, WordLabel> labels;  vector<vector<string>> result;    WordLabel lb;  for (string w : wordList) labels[w] = lb;    // forward BFS  auto it = labels.find(beginWord);  if (it == labels.end()) return vector<vector<string>>{};  it->second.level = 0;    it = labels.find(endWord);  if (it == labels.end()) return vector<vector<string>>{};  unordered\_set<string> pre; pre.insert(beginWord);    for (int k = 1; !pre.empty() && it->second.level < 0; ++k) {  unordered\_set<string> cur;  for (string w : pre) searchWord(w, labels, cur, k);  swap(pre, cur);  }  if (it->second.level < 0) return vector<vector<string>>{};  // backward BFS  result.push\_back(vector<string>{endWord});  for (int k = it->second.level; k > 0; --k) {  vector<vector<string>> res;  for (vector<string> v : result) {  for (string w : labels[v.back()].prev) {  vector<string> vec = v;  vec.push\_back(w);  res.push\_back(vec);  }  }  swap(result, res);  }    for (vector<string> &v: result) reverse(v.begin(), v.end());  return result;  }    void searchWord(string &word, unordered\_map<string, WordLabel>& labels,  unordered\_set<string> &cur, int level) {  string word0 = word;  for (int k = 0; k < word.size(); ++k) {  char c0 = word[k];  for (char c = 'a'; c <= 'z'; ++c) {  word[k] = c;  if (c == c0) continue;  auto it = labels.find(word);  if (it != labels.end() && (it->second.level == level || it->second.level < 0)) {  cur.insert(it->first);  it->second.level = level;  it->second.prev.push\_back(word0);  }  }  word[k] = c0;  }  } |

# 127. Word Ladder

|  |
| --- |
| Given two words (beginWord and endWord), and a dictionary's word list, find the length of shortest transformation sequence from beginWord to endWord, such that:  Only one letter can be changed at a time  Each intermediate word must exist in the word list  Note:  Return 0 if there is no such transformation sequence.  All words have the same length.  All words contain only lowercase alphabetic characters. |
| For example,  Given:  beginWord = "hit"  endWord = "cog"  wordList = ["hot","dot","dog","lot","log"]  As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",  return its length 5. |

**Sol: BFS**

把字典看成一张无向图，目标是找到beginWord->endWord的最短路。我们通过图上的BFS可以找到beginWord到所有单词的最短路长度。

|  |
| --- |
| **int ladderLength(string beginWord, string endWord, unordered\_set<string>& wordList)** {  auto it = wordList.find(beginWord);  if (it == wordList.end()) return 0;  wordList.erase(it);  vector<string> pre; pre.push\_back(beginWord);    for (int k = 1; !pre.empty(); ++k) {  vector<string> cur;  for (string word: pre) {  if (word.compare(endWord) == 0) return k;  searchWord(word, wordList, cur);  }  swap(pre, cur);  }  return 0;  }  void searchWord(string &word, unordered\_set<string>& wordList, vector<string> &result) {  for (int k = 0; k < word.size(); ++k) {  char c0 = word[k];  for (char c = 'a'; c <= 'z'; ++c) {  word[k] = c;  if (c == c0) continue;  auto it = wordList.find(word);  if (it != wordList.end()) {  result.push\_back(\*it);  wordList.erase(it);  }  }  word[k] = c0;  }  } |

**Sol2:foward-backward BFS**

从beginWord, endWord两端作BFS，最短路径出现在当一个单词已经在本层被visited的时候。例如上例中：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

Level 0: hit, cog

Level 1: [hot], [dog, log]

Level 2: 两棵树的这一层都是[dot, lot]，因此返回5

我们用一个unordered\_map<string, int>给word标记每个word是在第几步被访问的。

|  |
| --- |
| **int ladderLength(string beginWord, string endWord, unordered\_set<string>& wordList)** {  unordered\_map<string, int> step;  for (string w : wordList) step[w] = -1;    auto it = step.find(beginWord);  if (it == step.end()) return 0; //begin word not found  it->second = 0;  it = step.find(endWord);  if (it == step.end()) return 0; //end word not found  if (it -> second == 0) return 1; //begin word = end word & in dictionary  it->second = 1;    unordered\_set<string> pre; pre.insert(beginWord);  unordered\_set<string> post; post.insert(endWord);    for (int k = 1; !pre.empty(); ++k) {  unordered\_set<string> cur1, cur2;  for (string word: pre)  if (searchWord(word, step, cur1, k\*2)) return k\*2;  for (string word: post)  if (searchWord(word, step, cur2, k\*2+1)) return k\*2+1;  swap(pre, cur1);  swap(post, cur2);  }    return 0;  }  bool searchWord(string &word, unordered\_map<string, int>& step,  unordered\_set<string> &cur, int level) {  for (int k = 0; k < word.size(); ++k) {  char c0 = word[k];  for (char c = 'a'; c <= 'z'; ++c) {  word[k] = c;  if (c == c0) continue;  auto it = step.find(word);  if (it != step.end()) {  if (it->second >= 0 && it->second == level-1) return true;  cur.insert(it->first);  it->second = level;  }  }  word[k] = c0;  }  return false;  } |

# 128. Longest Consecutive Sequences

|  |
| --- |
| Given an unsorted array of integers, find the length of the longest consecutive elements sequence.  For example,  Given [100, 4, 200, 1, 3, 2],  The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.  Your algorithm should run in O(n) complexity. |

**Sol1: hashset**

用一个unordered\_set存储数组中的数，从而我们可以O(1)时间判断数是否在数组中。然后我们每次从set中取出一个数，并计算它所在的sequence的长度（以这个数为起点探测sequence的两段，并把找到的数从set中移除）。重复这个过程我们得到所有sequence的长度，因此可以知道最长sequence是多长。

|  |
| --- |
| **int longestConsecutive(vector<int>& nums)** {  int maxlen = 0;  unordered\_set<int> num\_set(nums.begin(), nums.end());  while (!num\_set.empty()) {  int v = \*(num\_set.begin()), vb = v-1, ve = v+1;  num\_set.erase(v);  for (; !num\_set.empty(); vb--) {  auto it = num\_set.find(vb);  if (it == num\_set.end()) break; else num\_set.erase(it);  }  for (; !num\_set.empty(); ve++) {  auto it = num\_set.find(ve);  if (it == num\_set.end()) break; else num\_set.erase(it);  }  maxlen = max(maxlen, ve-vb-1);  }  return maxlen;  } |

**Sol2: hashmap**

用一个unordered\_map存储包含某个数的sequence两端。我们只需要保证其中sequence两段为key对应的entry是up-to-date的即可。当加入一个新数时，它只需要retrieve它前后两个值的信息，以确定它所在的sequence的两端。

|  |
| --- |
| **int longestConsecutive(vector<int>& nums)** {  unordered\_map<int, pair<int, int>> boundary;    int maxlen = 0;  for (int v : nums) {  if (boundary.find(v) != boundary.end()) continue;    auto pre = boundary.find(v-1), post = boundary.find(v+1);  int pre\_v = pre == boundary.end()? v : pre->second.first;  int post\_v = post == boundary.end()? v : post->second.second;    maxlen = max(maxlen, post\_v - pre\_v + 1);  boundary[v] = make\_pair(pre\_v, post\_v);  boundary[pre\_v].second = post\_v;  boundary[post\_v].first = pre\_v;  }  return maxlen;  } |

# 129. Sum Root to Leaf Numbers

|  |
| --- |
| Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.  An example is the root-to-leaf path 1->2->3 which represents the number 123.  Find the total sum of all root-to-leaf numbers. |
| For example,  1  / \  2 3  The root-to-leaf path 1->2 represents the number 12.  The root-to-leaf path 1->3 represents the number 13.  Return the sum = 12 + 13 = 25. |

**Sol:preorder traversal**

前序遍历并maintain从根结点到每个结点对应的数。在到达叶结点时加入结果。

|  |
| --- |
| **int longestConsecutive(vector<int>& nums)** {  unordered\_map<int, pair<int, int>> boundary;    int maxlen = 0;  for (int v : nums) {  if (boundary.find(v) != boundary.end()) continue;    auto pre = boundary.find(v-1), post = boundary.find(v+1);  int pre\_v = pre == boundary.end()? v : pre->second.first;  int post\_v = post == boundary.end()? v : post->second.second;    maxlen = max(maxlen, post\_v - pre\_v + 1);    boundary[v] = make\_pair(pre\_v, post\_v);  if (pre\_v != v) boundary[pre\_v].second = post\_v;  if (post\_v != v) boundary[post\_v].first = pre\_v;  }  return maxlen;  } |

# 130. Surrounded Regions

|  |
| --- |
| Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.  A region is captured by flipping all 'O's into 'X's in that surrounded region. |
| For example,  X X X X  X O O X  X X O X  X O X X  After running your function, the board should be:  X X X X  X X X X  X X X X  X O X X |

**Sol1:flood fil (recursion)**

我们只需要从board的四个边界开始flood fill即可。为了和未访问过的'O'作区分，我们把访问过的'O'改变label为'N'。最后我们把所有仍为'O'的entry改为‘X'，'N'改为‘O'.

|  |
| --- |
| **void solve(vector<vector<char>>& board)** {  if (board.size() == 0 || board[0].size() == 0) return;  const int R = board.size(), C = board[0].size();  for (int k = 0; k < R; ++k) solve\_recur(board, k, 0, R, C);  for (int j = 0; j < C; ++j) solve\_recur(board, 0, j, R, C);  for (int k = 0; k < R; ++k) solve\_recur(board, k, C-1, R, C);  for (int j = 0; j < C; ++j) solve\_recur(board, R-1, j, R, C);  for (int k = 0; k < R; ++k) {  for (int j = 0; j < C; ++j) {  if (board[k][j] == 'O') board[k][j] = 'X';  if (board[k][j] == 'N') board[k][j] = 'O';  }  }  }  void solve\_recur(vector<vector<char>>& board, int k, int j, const int R, const int C) {  if (k < 0 || j < 0 || k >= R || j >= C || board[k][j] != 'O') return;  board[k][j] = 'N';    if (k < R-1) solve\_recur(board, k+1, j, R, C);  if (j < C-1) solve\_recur(board, k, j+1, R, C);  if (k > 1) solve\_recur(board, k-1, j, R, C);  if (j > 1) solve\_recur(board, k, j-1, R, C);  } |

**Sol2:flood fil (stack)**

对大的board递归实现很容易stack overflow。因此我们作一个非递归实现。

|  |
| --- |
| **void solve(vector<vector<char>>& board)** {  if (board.size() == 0 || board[0].size() == 0) return;  const int R = board.size(), C = board[0].size();  stack<pair<int, int>> pos;  for (int k = 0; k < R; ++k) {  pos.push(make\_pair(k, 0));  pos.push(make\_pair(k, C-1));  }  for (int j = 1; j < C-1; ++j) {  pos.push(make\_pair(0, j));  pos.push(make\_pair(R-1, j));  }  while (!pos.empty()) {  int k = pos.top().first, j = pos.top().second;  pos.pop();  if (board[k][j] != 'O') continue; else board[k][j] = 'N';    if (k < R-1) pos.push(make\_pair(k+1, j));  if (j < C-1) pos.push(make\_pair(k, j+1));  if (k > 0) pos.push(make\_pair(k-1, j));  if (j > 0) pos.push(make\_pair(k, j-1));  }    for (int k = 0; k < R; ++k) {  for (int j = 0; j < C; ++j) {  if (board[k][j] == 'O') board[k][j] = 'X';  if (board[k][j] == 'N') board[k][j] = 'O';  }  }  } |

# 

# 131. Palindrome Partitioning

|  |
| --- |
| Given a string s, partition s such that every substring of the partition is a palindrome.  Return all possible palindrome partitioning of s. |
| For example, given s = "aab", return  [  ["aa","b"],  ["a","a","b"]  ] |

**Sol: recursion**

递归结构: 取字符串的每个回文子串，并求剩余串的分割。

递归结束条件:字符串子串为空

预处理:判断字符串的子串是不是回文串可以通过遍历所有的子串中心。

|  |
| --- |
| **vector<vector<string>> partition(string s)** {  vector<vector<bool>> isp(s.size(), vector<bool>(s.size(), false));  for (int c = 0; c < s.size()\*2-1; ++c) {  int l = c/2, r = (c+1)/2;  for (; l >= 0 && s[l] == s[r]; --l, ++r) isp[l][r] = true;  }    vector<vector<string>> result;  vector<string> prefix;  partition\_recur(s, 0, prefix, result, isp);  return result;  }  void partition\_recur(string s, int l, vector<string> &prefix,  vector<vector<string>> &result, const vector<vector<bool>> &isp) {  if (l == s.size()) {  result.push\_back(prefix);  return;  }    prefix.push\_back("");  for (int r = l; r < s.size(); ++r) {  if (isp[l][r] == false) continue;  prefix.back() = s.substr(l, r-l+1);  partition\_recur(s, r+1, prefix, result, isp);  }  prefix.pop\_back();  } |

# 132. Palindrome Partitioning II

|  |
| --- |
| Given a string s, partition s such that every substring of the partition is a palindrome.  Return the minimum cuts needed for a palindrome partitioning of s.  For example, given s = "aab",  Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut. |

**Sol: DP**

保存前k个字符组成前缀的最小回文分割数，然后当检查一个更长的前缀时，遍历它可能的最后一个cut。

|  |
| --- |
| **int minCut(string s)** {  vector<vector<bool>> isp(s.size(), vector<bool>(s.size(), false));  for (int c = 0; c < s.size()\*2-1; ++c) {  int l = c/2, r = (c+1)/2;  for (; l >= 0 && s[l] == s[r]; --l, ++r) isp[l][r] = true;  }    vector<int> mc(s.size());  for (int k = 0; k < s.size(); ++k) {  mc[k] = isp[0][k]? 0 : k;  for (int j = 0; j < k; ++j) {  if (isp[j+1][k]) mc[k] = min(mc[k], mc[j]+1);  }  }  return mc.back();  } |

# 133. Clone Graph

|  |
| --- |
| Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors. |

**Corner case:**

需要问清graph的存储形式。像本例中，graph其实只是一个联通分支。

**Sol: recursion (DFS)**

在递归的过程中先复制结点再复制边。用一个unordered\_map记录结点到结点copy的映射。

1.Node如果没有被复制过，则

1. 复制并加入映射字典中。
2. 递归地复制node的所有邻居。
3. 把所有复制的邻居加入到复制node的邻居中。

2.返回复制的node

本题BFS也可以做

|  |
| --- |
| **UndirectedGraphNode \*cloneGraph(UndirectedGraphNode \*node)** {  unordered\_map<UndirectedGraphNode \*, UndirectedGraphNode \*> map;  return cloneGraph\_recur(node, map);  }  inline UndirectedGraphNode \*  cloneGraph\_recur(UndirectedGraphNode \*node,  unordered\_map<UndirectedGraphNode \*, UndirectedGraphNode \*> &map) {  if (node == NULL) return NULL;  auto it = map.find(node);  if (it == map.end()) {  UndirectedGraphNode \*node\_copy = new UndirectedGraphNode(node->label);  map[node] = node\_copy;  for (UndirectedGraphNode \*n : node->neighbors)  node\_copy->neighbors.push\_back(cloneGraph\_recur(n, map));  return node\_copy;  }  return it->second;  } |

# 134. Gas Station

|  |
| --- |
| There are N gas stations along a circular route, where the amount of gas at station i is gas[i].  You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.  Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.  Note:  The solution is guaranteed to be unique. |

**Sol: sliding window**

假设环形路上按照每天净油量的正负分段。则每段的负油量需由之前的正的净油量消除。如果正的净油量不够，则必须再累加前一段负的路程才能到更前一段正的路程加油。对称的，当净油量为正，则可以延伸扫描的尾段以消耗目前存余的油

|  |
| --- |
| **int canCompleteCircuit(vector<int>& gas, vector<int>& cost)** {  const int N = gas.size();  int first = N-1, last = 0, sum = 0;  for (int k = 0; k < N; ++k) {  if (sum + gas[last]-cost[last] >= 0) {  sum += gas[last]-cost[last];  last = last == N-1? 0:(last+1);  }  else {  sum += gas[first]-cost[first];  first = first == 0? N-1:(first-1);  }  }  return sum>=0? (first+1)%N: -1;  } |

# 135. Candy

|  |
| --- |
| There are N children standing in a line. Each child is assigned a rating value.  You are giving candies to these children subjected to the following requirements:  Each child must have at least one candy.  Children with a higher rating get more candies than their neighbors.  What is the minimum candies you must give? |

**Sol1: DP**

从左到右把小朋友分为升、降、持平三种类型的段落。升段的糖果从1...n，降段的糖果从n...1,在顶峰的糖果取两侧的较小值。持平的段落取1. 较简单的一种实现是先从左向右扫描，assign升段的糖果。然后从右向左扫描，assign降段的糖果，同时这也正确更新了peak位置的糖果。

|  |
| --- |
| **int candy(vector<int>& ratings)** {  vector<int> candies(ratings.size(), 1);  for (int k = 1; k < ratings.size(); ++k) {  if (ratings[k]>ratings[k-1]) candies[k] = candies[k-1]+1;  }    int sum = candies.back();  for (int k = ratings.size()-2; k >=0; --k) {  if (ratings[k]>ratings[k+1]) candies[k] = max(candies[k], candies[k+1]+1);  sum += candies[k];  }  return sum;  } |

**Sol2: DP (constant space)**

依次扫描由升、降、平段落形成的section，并加入总糖果数。但要注意minimal处会重复计数，需要扣除。

|  |
| --- |
| **int candy(vector<int>& ratings)** {  int sum = 0, N = ratings.size();  for (int k = 0; k < N; sum--) {  int upcount = 1, downcount = 1;  for (; k < N-1 && ratings[k+1] > ratings[k]; ++k, ++upcount) sum += upcount;  for (; k < N-1 && ratings[k+1] < ratings[k]; ++k, ++downcount) sum += downcount;  sum+= max(upcount, downcount);  for (; k < N-1 && ratings[k+1] == ratings[k]; ++k) ++sum;  if (k == N-1) break;  }  return sum;  } |

# 136. Single Number

|  |
| --- |
| Given an array of integers, every element appears twice except for one. Find that single one.  Note:  Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory? |

**Sol: bit ops**

考虑数字的某一位，如果single number该位为1,则数字中有奇数个1，否则有偶数个1. 通过xor运算可以达到该目的。

|  |
| --- |
| **int singleNumber(vector<int>& nums)** {  int result = 0;  for (int v : nums) result ^= v;  return result;  } |

# 137. Single Number II

|  |
| --- |
| Given an array of integers, every element appears three times except for one. Find that single one.  Note:  Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory? |

**Sol: bit opts**

考虑数字的某一位。若single number是1,则该位1出现3k+1次。我们用两个变量来实现一个0->3->2->0的计数器。

v(input) a(in) b(in) -> a(out) b(out)

**0 0 0 0** 0

**0 1 1**  1 **1**

0 1 0 1 0

**1 0 0** 1 **1**

1 1 1 1 0

**1 1 0 0** 0

因此

**a\_out** = ~((~v & ~a & ~b) | (v & a & ~b))

**b\_out** = (~v & a & b) | (v & ~a & ~b)

|  |
| --- |
| **int singleNumber(vector<int>& nums)** {  int a = 0, b = 0;  for(int v : nums) {  int ta = ~((~v & ~a & ~b) | (v & a & ~b));  int tb = (~v & a & b) | (v & ~a & ~b);  a = ta, b = tb;  }  return a & b;  } |

# 138. Copy List with Random Pointer

|  |
| --- |
| A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.  Return a deep copy of the list. |

**Sol1: hashing(with hashmap)**

第一轮复制nodes并建立一个从输入到复制的映射，第二轮把link复制下来。

|  |
| --- |
| **RandomListNode \*copyRandomList(RandomListNode \*head)** {  unordered\_map<RandomListNode \*, RandomListNode \*> map;  for (RandomListNode \*p = head; p; p = p->next) map[p] = new RandomListNode(p->label);  for (RandomListNode \*p = head; p; p = p->next) {  if (p->next) map[p]->next = map[p->next];  if (p->random) map[p]->random = map[p->random];  };  return map[head];  } |

**Sol2: hashing(with next pointer)**

通过两个链表的next指针把nodes连接起来（左二图），从而我们可以正确地复制random pointer（通过右二图中蓝色link)。最后我们可以通过修改next指针分离两个链表。

这个实现虽然省空间，但是因为修改了输入，线程安全性不好

|  |
| --- |
| **RandomListNode \*copyRandomList(RandomListNode \*head)** {  if (head == NULL) return NULL;    for (RandomListNode \*p = head; p;) {  RandomListNode \*q = new RandomListNode(p->label);  q->next = p->next;  p->next = q;  p = q->next;  }    RandomListNode \*head\_copy = head->next;    for (RandomListNode \*p = head; p; p = p->next->next) {  if (p->random != NULL) p->next->random = p->random->next;  }    for (RandomListNode \*p = head; p; p = p->next) {  RandomListNode \*q = p->next;  p->next = q->next;  if (q->next) q->next = q->next->next;  }    return head\_copy;  } |

# 139. Word Break

|  |
| --- |
| Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words. |
| For example, given  s = "leetcode",  dict = ["leet", "code"].  Return true because "leetcode" can be segmented as "leet code". |

**Sol: DP**

用has\_break[k]存储s[1...k]是否能被word break(s[0]表示空串)。

|  |
| --- |
| **bool wordBreak(string s, unordered\_set<string>& wordDict)** {  vector<bool> has\_break(s.size()+1, false);  has\_break[0] = true;  for (int k = 1; k <= s.size(); ++k) {  for (int j = 0; j < k; ++j) {  if (has\_break[j] && wordDict.find(s.substr(j, k-j)) != wordDict.end())  has\_break[k] = true;  }  }    return has\_break.back();  } |

# 140. Word Break II

|  |
| --- |
| Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.  Return all such possible sentences. |
| For example, given  s = "catsanddog",  dict = ["cat", "cats", "and", "sand", "dog"].  A solution is ["cats and dog", "cat sand dog"]. |

**Sol1: recursion**

剥离一个非空前缀为第一个单词，然后递归地分割剩下的部分。算法是指数级的，时间过长。

|  |
| --- |
| **vector<string> wordBreak(string s, unordered\_set<string>& wordDict)** {  vector<string> result;  string prefix("");  wordBreak\_recur(s, 0, wordDict, prefix, result);  return result;  }  void wordBreak\_recur(const string &s, int first, unordered\_set<string>& wordDict,  const string &prefix, vector<string> &result) {  if (first == s.size()) result.push\_back(prefix);  for (int k = first; k < s.size(); ++k) {  string word = s.substr(first, k-first+1);  if (wordDict.find(word) != wordDict.end()) {  //cout << word << endl;  string next\_prefix = prefix;  if (prefix.size()>0) next\_prefix += ' ';  next\_prefix += word;  wordBreak\_recur(s, k+1, wordDict, next\_prefix, result);  }  }  } |

**Sol2: memorized recursion**

用一个unordered\_map<int, vector<string>>存储s[1...k]的所有分割。

|  |
| --- |
| **vector<string> wordBreak(string s, unordered\_set<string>& wordDict)** {  unordered\_map<int, vector<string>> cache;  cache[s.size()] = vector<string>{""};  wordBreak\_recur(s, 0, wordDict, cache);  return cache[0];  }  void wordBreak\_recur(const string s, int first, const unordered\_set<string>& wordDict,  unordered\_map<int, vector<string>> &cache) {  if (first == s.size() || cache.find(first) != cache.end()) return;    vector<string> result;  for (int k = first; k < s.size(); ++k) {  string word = s.substr(first, k-first+1);  if (wordDict.find(word) == wordDict.end()) continue;  wordBreak\_recur(s, k+1, wordDict, cache);  for (string str : cache[k+1])  result.push\_back(k == s.size()-1? word: (word + ' ' + str));  }  cache[first] = result;  } |

# 

# 141. Linked List Cycle

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

**Sol1: hashing**

把已经出现的nodes压进unordered\_set，直到出现一个已经出现过的node。

|  |
| --- |
| **ListNode \*detectCycle(ListNode \*head)** {  unordered\_set<ListNode \*> occ;  for (ListNode \*p = head; p; p = p->next) {  if (occ.find(p) == occ.end()) occ.insert(p);  else return true;  }  return false;  } |

**Sol2: fast-slow pointer**

用两个不同速度的pointer扫描链表，这样如果是cycled linked list，fast pointer会赶上slow pointer.

|  |
| --- |
| **bool hasCycle(ListNode \*head)** {  for (ListNode \*p = head, \*q = head; q; p = p->next) {  q = q->next;  if (!q) return false;  q = q->next;  if (p == q) return true;  }  return false;  } |

# 

# 142. Linked List Cycle II

|  |
| --- |
| Given a linked list, return the node where the cycle begins. If there is no cycle, return null.  Note: Do not modify the linked list.  Follow up: Can you solve it without using extra space? |

**Sol1: hashing**

用unordered\_map存储已经出现的结点。

|  |
| --- |
| **ListNode \*detectCycle(ListNode \*head)** {  unordered\_set<ListNode \*> occ;  for (ListNode \*p = head; p; p = p->next) {  if (occ.find(p) == occ.end()) occ.insert(p);  else return p;  }  return NULL;  } |

**Sol2: two pointer (fast/slow)**

假设环外有m个node，环内有n个node。把环内结点标为0, 1, … n-1.两个node第一次相遇时走了kn/2n步 ，在node kn-m。因此再走m步可以到起点位置。所以我们把一个指针放回链表头，以同样的速度前进，则两个指针在m步后相遇在cycle开始位置。

|  |
| --- |
| **ListNode \*detectCycle(ListNode \*head)** {  if (head == NULL) return NULL;    ListNode \*fast, \*slow;  for (fast = slow = head; fast;) {  fast = fast->next;  if (!fast) return NULL;  slow = slow->next;  fast = fast->next;  if (fast == slow) {  for (ListNode \*p = head; p != fast; p = p->next, fast = fast->next);  return fast;  }  }  return NULL;  } |

# 

# 143. Reorder List

|  |
| --- |
| Given a singly linked list L: L0→L1→…→Ln-1→Ln,  reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…  You must do this in-place without altering the nodes' values.  For example,  Given {1,2,3,4}, reorder it to {1,4,2,3}. |

**Sol: linkedlist ops**

首先把链表分成均匀的两段，然后把第二段反转，最后合并两个链表。

|  |
| --- |
| **void reorderList(ListNode\* head)** {  head = head == NULL? NULL : merge(head, reverse(splitHalf(head)));  }  inline ListNode \*splitHalf(ListNode \*head) {  ListNode \*p, \*q;  for (p = q = head; q; q = q->next, p = p->next) {  q = q->next;  if (q == NULL) break;  }  q = p->next;  p->next = NULL;  return q;  }  inline ListNode \*reverse(ListNode \*head) {  ListNode \*tail = NULL;  for (ListNode \*p = head; p;) {  ListNode \*next = p->next;  p->next = tail;  tail = p;  p = next;  }  return tail;  }  inline ListNode \*merge(ListNode \*l1, ListNode \*l2) {  ListNode sentinel(0);  for (ListNode \*tail= &sentinel; l1 != NULL || l2 != NULL; ) {  if (l1) {  tail->next = l1;  tail = l1;  l1 = l1->next;  }  if (l2) {  tail->next = l2;  tail = l2;  l2 = l2->next;  }  }  return sentinel.next;  } |

# 144. Binary Tree Preorder Traversal

|  |
| --- |
| Given a binary tree, return the preorder traversal of its nodes' values.  Note: Recursive solution is trivial, could you do it iteratively? |
| For example:  Given binary tree {1,#,2,3},  1  \  2  /  3  return [1,2,3]. |

**Sol1: recursion**

递归实现按定义来就可以。

|  |
| --- |
| **vector<int> preorderTraversal(TreeNode\* root)** {  vector<int> result;  preorderTraversal\_recur(root, result);  return result;  }  void preorderTraversal\_recur(TreeNode \*root, vector<int> &result) {  if (!root) return;  result.push\_back(root->val);  preorderTraversal\_recur(root->left, result);  preorderTraversal\_recur(root->right, result);  } |

**Sol2: iteration with stack**

非递归实现用一个stack即可，注意把右子树先进栈。

|  |
| --- |
| **vector<int> preorderTraversal(TreeNode\* root)** {  vector<int> result;  stack<TreeNode \*> stk;  stk.push(root);    while (!stk.empty()) {  TreeNode \*t = stk.top();  stk.pop();  if (t) {  result.push\_back(t->val);  stk.push(t->right);  stk.push(t->left);  }  }    return result;  } |

# 

# 145. Binary Tree Postorder Traversal

|  |
| --- |
| Given a binary tree, return the postorder traversal of its nodes' values.  Note: Recursive solution is trivial, could you do it iteratively? |
| For example:  Given binary tree {1,#,2,3},  1  \  2  /  3  return [3,2,1]. |

**Sol1: recursion**

递归按照定义来就可以。

|  |
| --- |
| **vector<int> postorderTraversal(TreeNode\* root)** {  vector<int> result;  postorderTraversal\_recur(root, result);  return result;  }  void postorderTraversal\_recur(TreeNode\* root, vector<int> &result) {  if (root == NULL) return;  postorderTraversal\_recur(root->left, result);  postorderTraversal\_recur(root->right, result);  result.push\_back(root->val);  } |

**Sol2: iteration with a stack**

用一个stack实现。每个node进出栈两次。第一次出栈时把自己和左右结点进栈，第二次出栈时print结点。为了区分两次出栈，用一个bool来标记每个入栈结点。

|  |
| --- |
| **vector<int> postorderTraversal(TreeNode\* root)** {  vector<int> result;  stack<pair<TreeNode\*, bool>> stk;  stk.push(make\_pair(root, true));    while (!stk.empty()) {  TreeNode \*t = stk.top().first;  bool flag = stk.top().second;    stk.pop();  if (t == NULL) continue;  if (flag) {  stk.push(make\_pair(t, false));  stk.push(make\_pair(t->right, true));  stk.push(make\_pair(t->left, true));  }  else {  result.push\_back(t->val);  }  }  return result;  } |

**Sol3: iteration with stack**

如果允许更改树结构，我们可以不用bool标记。而是在第一次入栈时分离父结点和子结点，并在结点为叶子时输出。

|  |
| --- |
| **vector<int> postorderTraversal(TreeNode\* root)** {  vector<int> result;  stack<TreeNode\*> stk;  if (root) stk.push(root);    while (!stk.empty()) {  TreeNode \*t = stk.top();  stk.pop();  TreeNode \*l = t->left, \*r = t->right;  t->left = t->right = NULL;  if (l != NULL || r != NULL) stk.push(t);  else result.push\_back(t->val);    if (r) stk.push(r);  if (l) stk.push(l);  }  return result;  } |

# 

# 146. LRU Cache

|  |
| --- |
| Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.  get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.  set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item. |

**Sol: doubly linked list+hashing**

我们需要一个双向链表记录历史，hash表记录从key到value的mapping。

考虑set的几种情况：

1. LRU未满且key不存在：只需要创键(key, value)对，插入历史头部
2. LRU满了且key不存在：删除历史尾部(key, value)对，并插入历史头部
3. key存在／get：移动(key, value)到历史头部

因此我们用hash表存储key->链表元素的映射，链表存储(key, value) pair。

|  |
| --- |
| **class LRUCache{**  **public:**  **LRUCache(int capacity)**: count(capacity) {}    **int get(int key)** {  auto it = map.find(key);  if (it == map.end()) return -1;  history.splice(history.begin(), history, it->second);  return it->second->second;  }    **void set(int key, int value)** {  auto it = map.find(key);  if (it != map.end()) {//move key to begining of history  history.splice(history.begin(), history, it->second);  it->second->second = value;  }  else {  if (count == 0) {//remove front key from history  map.erase(history.back().first);  history.pop\_back();  count++;  }  auto it = history.insert(history.begin(), make\_pair(key, value));  map[key] = it;  count--;  }  }    private:  list<pair<int, int>> history;  unordered\_map<int, list<pair<int, int>>::iterator> map;  int count;  **};** |

# 147. Insertion Sorted List

|  |
| --- |
| Sort a linked list using insertion sort. |

**Sol: linkedlist ops**

maintain一个输出list存储已经排好序的前k个元素，为方便初始时放置一个值为INT\_MIN的链表头。

|  |
| --- |
| **ListNode\* insertionSortList(ListNode\* head)** {  ListNode sentinel(INT\_MIN);    for (ListNode \*p = head; p; ) {  ListNode \*q, \*t, \*next = p->next;  p->next = NULL;  for (q = &sentinel; q != NULL && q->val <= p->val; q = q->next) t = q;  p->next = q;  t->next = p;  p = next;  }  return sentinel.next;  } |

# 

# 148. Sort List

|  |
| --- |
| Sort a linked list in *O*(*n* log *n*) time using constant space complexity. |

**Sol：merge sort.**

每个循环依次取出长度为2k的链表两两合并。

|  |
| --- |
| **ListNode\* sortList(ListNode\* head)** {  for (int k = 1; ; k \*= 2) {  ListNode sentinel(0);  ListNode \*tail = &sentinel;  bool merged = false;  while (head) {  ListNode \*first = head;  ListNode \*second = getPrefix(head, k);  if (second == NULL) {  tail->next = head;  break;  }  merged = true;  head = getPrefix(second, k);    ListNode \*new\_tail;  tail->next = merge(first, second, new\_tail);  tail = new\_tail;  }  head = sentinel.next;  if (merged == false) break;  }  return head;  }  ListNode\* getPrefix(ListNode\* head, int count) {  ListNode \*p = head;  for (int k = 0; p && k < count-1; ++k) p = p->next;  if (p == NULL) return NULL;  ListNode \*q = p->next;  p->next = NULL;  return q;  }  ListNode\* merge(ListNode \*l1, ListNode \*l2, ListNode \*&tail) {  ListNode sentinel(0); tail = &sentinel;  while (l1 || l2) {  ListNode \*&h = (l1 == NULL || (l2 != NULL && l2->val < l1->val))? l2 : l1;  tail->next = h;  h = h->next;  tail = tail->next;  }  return sentinel.next;  } |

# 

# 149. Max Points on a Line

|  |
| --- |
| Given n points on a 2D plane, find the maximum number of points that lie on the same straight line. |

**Sol1: Hough transform**

我们用ax + by + c = 0来表示每根直线，但约束a, b互质或有一个为0。我们维护一个从直线参数(a, b, c)到直线上点数的映射（unordered\_map)。

**Sol2: hashing + enumeration**

我们遍历直线上出现的第一个点。我们用hashmap存储从slope到count的映射。当遇到和起点重合的点，则累加到所有的直线。

|  |
| --- |
| **int maxPoints(vector<Point>& points)** {  int max\_count = 0;  for (int k = 0; k < points.size(); ++k) {  struct hashfunc { size\_t operator() (const pair<int,int>& l) const {  return l.first ^ l.second; }  };  unordered\_map<pair<int, int>, int, hashfunc> hist;    Point p0 = points[k];  int mc = 0, np0 = 1;  for (int j = k+1; j < points.size(); ++j) {  if (points[j].x == p0.x && points[j].y == p0.y) {  ++np0;  continue;  }  pair<int, int> slope = offset\_to\_slope(points[j].x - p0.x, points[j].y - p0.y);  auto it = hist.find(slope);  if (it == hist.end()) {  hist[slope] = 1;  mc = max(mc, 1);  }  else {  it->second++;  mc = max(mc, it->second);  }  }  max\_count = max(max\_count, mc+np0);  }  return max\_count;  }  // compute unique slope representation  pair<int, int> offset\_to\_slope(int vx, int vy) {  int factor = gcd(abs(vx), abs(vy));  if (vx < 0 || (vx == 0 && vy < 0)) factor = -factor;  return make\_pair(vx/factor, vy/factor);  }  // greatest common divisor  int gcd(int a, int b) {  if ( a==0 ) return b;  return gcd ( b%a, a );  } |

# 150. Evaluate Reverse Polish Notation

|  |
| --- |
| Evaluate the value of an arithmetic expression in Reverse Polish Notation.  Valid operators are +, -, \*, /. Each operand may be an integer or another expression. |
| Some examples:  ["2", "1", "+", "3", "\*"] -> ((2 + 1) \* 3) -> 9  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6 |

**Sol: stack**

用一个栈保存操作数。每次看到一个操作符，则计算以栈顶两元素为操作数的结果压入栈。

|  |
| --- |
| **int evalRPN(vector<string>& tokens)** {  stack<int> stk;  for (string t : tokens) {  if ((t.size()> 1) || (t[0]>='0' && t[0]<='9')) stk.push(stoi(t));  else {  int op2 = stk.top(); stk.pop();  int op1 = stk.top(); stk.pop();    switch (t[0]) {  case '+': stk.push(op1 + op2); break;  case '-': stk.push(op1 - op2); break;  case '\*': stk.push(op1 \* op2); break;  case '/': stk.push(op1 / op2); break;  default: break;  }  }  }  return stk.top();  } |