08. Convert Sorted Array to Binary Search Tree

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

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.

Example:

```
Given the sorted array: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

      0
     / \
   -3   9
   /   /
 -10  5
```

这道题是要将有序数组转为二叉搜索树，所谓二叉搜索树，是一种始终满足左<根<右的特性，如果将二叉搜索树按中序遍历的话，得到的就是一个有序数组了。那么反过来，我们可以得知，根节点应该是有序数组的中间点，从中间点分开为左右两个有序数组，在分别找出其中间点作为原中间点的左右两个子节点，这不就是是二分查找法的核心思想么。所以这道题考的就是二分查找法，代码如下：

解法一：

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(nums, 0 , (int)nums.size() - 1);
    }
    TreeNode* helper(vector<int>& nums, int left, int right) {
        if (left > right) return NULL;
        int mid = left + (right - left) / 2;
        TreeNode *cur = new TreeNode(nums[mid]);
        cur->left = helper(nums, left, mid - 1);
        cur->right = helper(nums, mid + 1, right);
        return cur;
    }
};
```

我们也可以不使用额外的递归函数，而是在原函数中完成递归，由于原函数的参数是一个数组，所以当把输入数组的中间数字取出来后，需要把所有两端的数组组成一个新的数组，并且分别调用递归函数，并且连到新创建的cur结点的左右子结点上面，参见代码如下：

解法二：

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        if (nums.empty()) return NULL;
        int mid = nums.size() / 2;
        TreeNode *cur = new TreeNode(nums[mid]);
        vector<int> left(nums.begin(), nums.begin() + mid), right(nums.begin() + mid + 1, nums.end());
        cur->left = sortedArrayToBST(left);
        cur->right = sortedArrayToBST(right);
        return cur;
    }
};
```

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.

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.

Example:

```
Given the sorted linked list: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

      0
     / \
   -3   9
   /   /
 -10  5
```

这道题是要求把有序链表转为二叉搜索树，和之前那道 Convert Sorted Array to Binary Search Tree 思路完全一样，只不过是操作的数据类型有所差别，一个是数组，一个是链表。数组方便就方便在可以通过index直接访问任意一个元素，而链表不行。由于二分查找法每次需要找到中点，而链表的查找中间点可以通过快慢指针来操作，可参见之前的两篇博客 Reorder List 和 Linked List Cycle II 有关快慢指针的应用。找到中点后，要以中点的值建立一个数的根节点，然后需要把原链表断开，分为前后两个链表，都不能包含原中节点，然后再分别对这两个链表递归调用原函数，分别连上左右子节点即可。代码如下：

解法一：

```
class Solution {
public:
```

```
    TreeNode *sortedListToBST(ListNode* head) {
        if (!head) return NULL;
        if (!head->next) return new TreeNode(head->val);
        ListNode *slow = head, *fast = head, *last = slow;
        while (fast->next && fast->next->next) {
            last = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        fast = slow->next;
        last->next = NULL;
        TreeNode *cur = new TreeNode(slow->val);
        if (head != slow) cur->left = sortedListToBST(head);
        cur->right = sortedListToBST(fast);
        return cur;
    }
};
```

我们也可以采用如下的递归方法，重写一个递归函数，有两个输入参数，子链表的起点和终点，因为知道了这两个点，链表的范围就可以确定了，而直接将中间部分转换为二叉搜索树即可，递归函数中的内容跟上面解法中的极其相似，参见代码如下：

解法二：

```
class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        if (!head) return NULL;
        return helper(head, NULL);
    }
    TreeNode* helper(ListNode* head, ListNode* tail) {
        if (head == tail) return NULL;
        ListNode *slow = head, *fast = head;
        while (fast != tail && fast->next != tail) {
            slow = slow->next;
            fast = fast->next->next;
        }
        TreeNode *cur = new TreeNode(slow->val);
        cur->left = helper(head, slow);
        cur->right = helper(slow->next, tail);
        return cur;
    }
};
```

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]
]
```

这道二叉树路径之和在之前那道题 Path Sum 的基础上又需要找出路径，但是基本思想都一样，还是需要用深度优先搜索 DFS，只不过数据结构相对复杂一点，需要用到二维的 vector，而且每当 DFS 搜索到新结点时，都要保存该结点。而且每当找出一条路径之后，都将这个保存为一维 vector 的路径保存到最终结果二维 vector 中。并且，每当 DFS 搜索到子结点，发现不是路径和时，返回上一个结点时，需要把该结点从一维 vector 中移除，参见代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        vector<vector<int>> res;
        vector<int> out;
        helper(root, sum, out, res);
        return res;
    }
    void helper(TreeNode* node, int sum, vector<int>& out,
vector<vector<int>>& res) {
        if (!node) return;
        out.push_back(node->val);
        if (sum == node->val && !node->left && !node->right) {
            res.push_back(out);
        }
        helper(node->left, sum - node->val, out, res);
        helper(node->right, sum - node->val, out, res);
        out.pop_back();
```

```
    }
};
```

下面这种方法是迭代的写法，用的是中序遍历的顺序，参考之前那道 Binary Tree Inorder Traversal，中序遍历本来是要用栈来辅助运算的，由于要取出路径上的结点值，所以用一个 vector 来代替 stack，首先利用 while 循环找到最左子结点，在找的过程中，把路径中的结点值都加起来，这时候取出 vector 中的尾元素，如果其左右子结点都不存在且当前累加值正好等于 sum 了，将这条路径取出来存入结果 res 中，下面的部分是和一般的迭代中序写法有所不同的地方，由于中序遍历的特点，遍历到当前结点的时候，是有两种情况的，有可能此时是从左子结点跳回来的，此时正要去右子结点，则当前的结点值还是算在路径中的；也有可能当前是从右子结点跳回来的，并且此时要跳回上一个结点去，此时就要减去当前结点值，因为其已经不属于路径中的结点了。为了区分这两种情况，这里使用一个额外指针 pre 来指向前一个结点，如果右子结点存在且不等于 pre，直接将指针移到右子结点，反之更新 pre 为 cur，cur 重置为空，val 减去当前结点，st 删掉最后一个结点，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        vector<vector<int>> res;
        vector<TreeNode*> st;
        TreeNode *cur = root, *pre = nullptr;
        int val = 0;
        while (cur || !st.empty()) {
            while (cur) {
                st.push_back(cur);
                val += cur->val;
                cur = cur->left;
            }
            cur = st.back();
            if (!cur->left && !cur->right && val == sum) {
                vector<int> v;
                for (auto &a : st) v.push_back(a->val);
                res.push_back(v);
            }
            if (cur->right && cur->right != pre) {
                cur = cur->right;
            } else {
                pre = cur;
                val -= cur->val;
                st.pop_back();
                cur = nullptr;
            }
        }
        return res;
    }
};
```

114. Flatten Binary Tree to Linked List

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
```

click to show hints.

Hints:

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order trave

这道题要求把二叉树展开成链表，根据展开后形成的链表的顺序分析出是使用先序遍历，那么只要是数的遍历就有递归和非递归的两种方法来求解，这里我们也用两种方法来求解。首先来看递归版本的，思路是先利用 DFS 的思路找到最左子节点，然后回到其父节点，把其父节点和右子节点断开，将原左子结点连上父节点的右子节点上，然后再把原右子节点连到新右子节点的右子节点上，然后再回到上一父节点做相同操作。代码如下：

解法一：

```
class Solution {
public:
    void flatten(TreeNode *root) {
        if (!root) return;
        if (root->left) flatten(root->left);
        if (root->right) flatten(root->right);
        TreeNode *tmp = root->right;
        root->right = root->left;
        root->left = NULL;
        while (root->right) root = root->right;
        root->right = tmp;
    }
};
```

例如，对于下面的二叉树，上述算法的变换的过程如下：

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

      1
     / \
    2   5
     \   \
      3   6
       \
        4

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

下面再来看非迭代版本的实现，这个方法是从根节点开始出发，先检测其左子结点是否存在，如存在则将根节点和其右子节点断开，将左子结点及其后面所有结构一起连到原右子节点的位置，把原右子节点连到元左子结点最后面的右子节点之后。代码如下：

解法二：

```cpp
class Solution {
public:
    void flatten(TreeNode *root) {
        TreeNode *cur = root;
        while (cur) {
            if (cur->left) {
                TreeNode *p = cur->left;
                while (p->right) p = p->right;
                p->right = cur->right;
                cur->right = cur->left;
                cur->left = NULL;
            }
            cur = cur->right;
        }
    }
};
```

例如，对于下面的二叉树，上述算法的变换的过程如下：

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

  1
   \
    2
   / \
  3   4
       \
        5
         \
          6

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

前序迭代解法如下：

解法三：

```
class Solution {
public:
    void flatten(TreeNode* root) {
        if (!root) return;
        stack<TreeNode*> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode *t = s.top(); s.pop();
            if (t->left) {
                TreeNode *r = t->left;
                while (r->right) r = r->right;
                r->right = t->right;
                t->right = t->left;
                t->left = NULL;
            }
            if (t->right) s.push(t->right);
        }
    }
};
```

115. Distinct Subsequences

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`.

看到有关字符串的子序列或者配准类的问题，首先应该考虑的就是用动态规划Dynamic Programming来求解，这个应成为条件反射。而所有DP问题的核心就是找出递推公式，想这道题就是递推一个二维的dp数组，下面我们从题目中给的例子来分析，这个二维dp数组应为：

```
  Ø r a b b b i t
Ø 1 1 1 1 1 1 1 1
r 0 1 1 1 1 1 1 1
a 0 0 1 1 1 1 1 1
b 0 0 0 1 2 3 3 3
b 0 0 0 0 1 3 3 3
i 0 0 0 0 0 0 3 3
t 0 0 0 0 0 0 0 3
```

首先，若原字符串和子序列都为空时，返回1，因为空串也是空串的一个子序列。若原字符串不为空，而子序列为空，也返回1，因为空串也是任意字符串的一个子序列。而当原字符串为空，子序列不为空时，返回0，因为非空字符串不能当空字符串的子序列。理清这些，二维数组dp的边缘便可以初始化了，下面只要找出递推式，就可以更新整个dp数组了。我们通过观察上面的二维数组可以发现，当更新到dp[i][j]时，dp[i][j] >= dp[i][j - 1] 总是成立，再进一步观察发现，当 T[i - 1] == S[j - 1] 时，dp[i][j] = dp[i][j - 1] + dp[i - 1][j - 1]，若不等， dp[i][j] = dp[i][j - 1]，所以，综合以上，递推式为：

dp[i][j] = dp[i][j - 1] + (T[i - 1] == S[j - 1] ? dp[i - 1][j - 1] : 0)

根据以上分析，可以写出代码如下：

```
class Solution {
public:
    int numDistinct(string S, string T) {
        int dp[T.size() + 1][S.size() + 1];
        for (int i = 0; i <= S.size(); ++i) dp[0][i] = 1;
        for (int i = 1; i <= T.size(); ++i) dp[i][0] = 0;
        for (int i = 1; i <= T.size(); ++i) {
            for (int j = 1; j <= S.size(); ++j) {
                dp[i][j] = dp[i][j - 1] + (T[i - 1] == S[j - 1] ? dp[i - 1][j - 1] : 0);
            }
        }
        return dp[T.size()][S.size()];
    }
};
```

116. Populating Next Right Pointers in Each Node

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *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`.

Example:

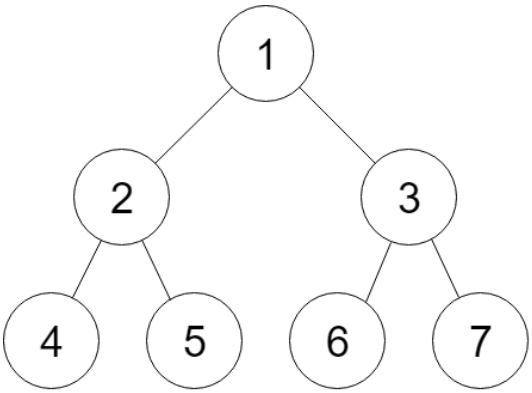

Figure A

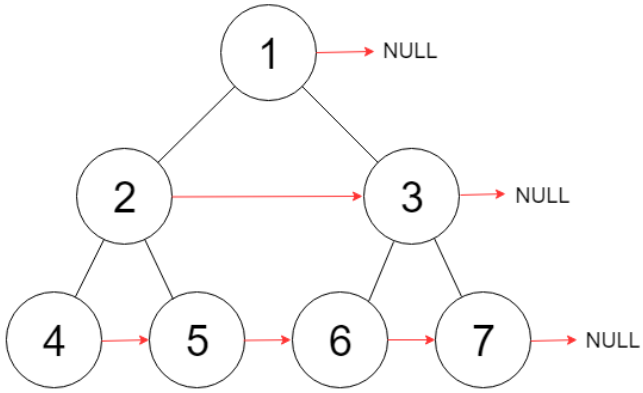

Figure B

```
Input: {"$id":"1","left":{"$id":"2","left":
{"$id":"3","left":null,"next":null,"right":null,"val":4},"next":null,"right
":
{"$id":"4","left":null,"next":null,"right":null,"val":5},"val":2},"next":nu
ll,"right":{"$id":"5","left":
{"$id":"6","left":null,"next":null,"right":null,"val":6},"next":null,"right
":
{"$id":"7","left":null,"next":null,"right":null,"val":7},"val":3},"val":1}

Output: {"$id":"1","left":{"$id":"2","left":{"$id":"3","left":null,"next":
{"$id":"4","left":null,"next":{"$id":"5","left":null,"next":
{"$id":"6","left":null,"next":null,"right":null,"val":7},"right":null,"val"
:6},"right":null,"val":5},"right":null,"val":4},"next":{"$id":"7","left":
{"$ref":"5"},"next":null,"right":{"$ref":"6"},"val":3},"right":
{"$ref":"4"},"val":2},"next":null,"right":{"$ref":"7"},"val":1}

Explanation: Given the above perfect binary tree (Figure A), your function
should populate each next pointer to point to its next right node, just
like in Figure B.
```

Note:

- You may only use constant extra space.
- Recursive approach is fine, implicit stack space does not count as extra space for this problem.

这道题实际上是树的层序遍历的应用，可以参考之前的博客 Binary Tree Level Order Traversal，既然是遍历，就有递归和非递归两种方法，最好两种方法都要掌握，都要会写。下面先来看递归的解法，由于是完全二叉树，所以若节点的左子结点存在的话，其右子节点必定存在，所以左子结点的 next 指针可以直接指向其右子节点，对于其右子节点的处理方法是，判断其父节点的 next 是否为空，若不为空，则指向其 next 指针指向的节点的左子结点，若为空则指向 NULL，代码如下：

解法一：

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        if (root->left) root->left->next = root->right;
        if (root->right) root->right->next = root->next? root->next->left
: NULL;
        connect(root->left);
        connect(root->right);
        return root;
    }
};
```

对于非递归的解法要稍微复杂一点，但也不算特别复杂，需要用到 queue 来辅助，由于是层序遍历，每层的节点都按顺序加入 queue 中，而每当从 queue 中取出一个元素时，将其 next 指针指向 queue 中下一个节点即可，对于每层的开头元素开始遍历之前，先统计一下该层的总个数，用个 for 循环，这样当 for 循环结束的时候，该层就已经被遍历完了，参见代码如下：

解法二：

```
// Non-recursion, more than constant space
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; ++i) {
                Node *t = q.front(); q.pop();
                if (i < size - 1) {
                    t->next = q.front();
                }
                if (t->left) q.push(t->left);
                if (t->right) q.push(t->right);
            }
        }
        return root;
    }
};
```

我们再来看下面这种碉堡了的方法，用两个指针 start 和 cur，其中 start 标记每一层的起始节点，cur 用来遍历该层的节点，设计思路之巧妙，不得不服啊：

解法三：

```
// Non-recursion, constant space
```

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        Node *start = root, *cur = NULL;
        while (start->left) {
            cur = start;
            while (cur) {
                cur->left->next = cur->right;
                if (cur->next) cur->right->next = cur->next->left;
                cur = cur->next;
            }
            start = start->left;
        }
        return root;
    }
};
```

117. Populating Next Right Pointers in Each Node II

Given a binary tree

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *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`.

Example:

```
Input: {"$id":"1","left":{"$id":"2","left":
{"$id":"3","left":null,"next":null,"right":null,"val":4},"next":null,"right
":
{"$id":"4","left":null,"next":null,"right":null,"val":5},"val":2},"next":nu
ll,"right":{"$id":"5","left":null,"next":null,"right":
{"$id":"6","left":null,"next":null,"right":null,"val":7},"val":3},"val":1}

Output: {"$id":"1","left":{"$id":"2","left":{"$id":"3","left":null,"next":
{"$id":"4","left":null,"next":
{"$id":"5","left":null,"next":null,"right":null,"val":7},"right":null,"val"
:5},"right":null,"val":4},"next":
{"$id":"6","left":null,"next":null,"right":{"$ref":"5"},"val":3},"right":
{"$ref":"4"},"val":2},"next":null,"right":{"$ref":"6"},"val":1}

Explanation: Given the above binary tree (Figure A), your function should
populate each next pointer to point to its next right node, just like in
Figure B.
```

Note:

- You may only use constant extra space.
- Recursive approach is fine, implicit stack space does not count as extra space for this problem.

这道是之前那道 Populating Next Right Pointers in Each Node 的延续，原本的完全二叉树的条件不再满足，但是整体的思路还是很相似，仍然有递归和非递归的解法。我们先来看递归的解法，这里由于子树有可能残缺，故需要平行扫描父节点同层的节点，找到他们的左右子节点。代码如下：

解法一：

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        Node *p = root->next;
        while (p) {
            if (p->left) {
                p = p->left;
                break;
            }
            if (p->right) {
                p = p->right;
                break;
            }
            p = p->next;
        }
        if (root->right) root->right->next = p;
        if (root->left) root->left->next = root->right ? root->right : p;
        connect(root->right);
```

```
        connect(root->left);
        return root;
    }
};
```

对于非递归的方法，我惊喜的发现之前的方法直接就能用，完全不需要做任何修改，算法思路可参见之前的博客 Populating Next Right Pointers in Each Node，代码如下：

解法二：

```
// Non-recursion, more than constant space
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int len = q.size();
            for (int i = 0; i < len; ++i) {
                Node *t = q.front(); q.pop();
                if (i < len - 1) t->next = q.front();
                if (t->left) q.push(t->left);
                if (t->right) q.push(t->right);
            }
        }
        return root;
    }
};
```

虽然以上的两种方法都能通过OJ，但其实它们都不符合题目的要求，题目说只能使用constant space，可是OJ却没有写专门检测space使用情况的test，那么下面贴上constant space的解法，这个解法也是用的层序遍历，只不过没有使用queue了，我们建立一个dummy结点来指向每层的首结点的前一个结点，然后指针cur用来遍历这一层，我们实际上是遍历一层，然后连下一层的next，首先从根结点开始，如果左子结点存在，那么cur的next连上左子结点，然后cur指向其next指针；如果root的右子结点存在，那么cur的next连上右子结点，然后cur指向其next指针。此时root的左右子结点都连上了，此时root向右平移一位，指向其next指针，如果此时root不存在了，说明当前层已经遍历完了，我们重置cur为dummy结点，root此时为dummy->next，即下一层的首结点，然后dummy的next指针清空，或者也可以将cur的next指针清空，因为前面已经将cur赋值为dummy了。那么现在想一想，为什么要清空？因为我们用dummy的目的就是要指到下一行的首结点的位置即dummy->next，而一旦将root赋值为dummy->next了之后，这个dummy的使命就已经完成了，必须要断开，如果不断开的话，那么假设现在root是叶结点了，那么while循环还会执行，不会进入前两个if，然后root右移赋空之后，会进入最后一个if，之前没有断开dummy->next的话，那么root又指向之前的叶结点了，死循环诞生了，跪了。所以一定要记得清空哦，呵呵哒～

这里再来说下dummy结点是怎样指向每层的首结点的前一个结点的，过程是这样的，dummy是创建出来的一个新的结点，其目的是为了指向root结点的下一层的首结点的前一个，具体是这么做到的呢，主要是靠cur指针，首先cur指向dummy，然后cur再连上root下一层的首结点，这样dummy也就连上了。然后当root层遍历完了之后，root需要往下移动一层，这样dummy结点之后连接的位置就正好赋值给root，然后cur再指向dummy，dummy之后断开，这样又回到了初始状态，以此往复就可以都连上了，代码如下：

解法三：

```
// Non-recursion, constant space
class Solution {
public:
    Node* connect(Node* root) {
        Node *dummy = new Node(0, NULL, NULL, NULL), *cur = dummy, *head = root;
        while (root) {
            if (root->left) {
                cur->next = root->left;
                cur = cur->next;
            }
            if (root->right) {
                cur->next = root->right;
                cur = cur->next;
            }
            root = root->next;
            if (!root) {
                cur = dummy;
                root = dummy->next;
                dummy->next = NULL;
            }
        }
        return head;
    }
};
```

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.

这道题给了我们一个二维数组组成的三角形，让我们寻找一条自上而下的路径，使得路径和最短。那么那道题后还是先考虑下暴力破解，我们可以发现如果要遍历所有的路径的话，那可以是阶乘级的时间复杂度啊，OJ必灭之，趁早断了念想比较好。必须要优化时间复杂度啊，题目中给的例子很容易把人带偏，让人误以为贪婪算法可以解题，因为看题例子中的红色数组，在根数字2的下方选小的数字3，在3的下方选小的数字5，在5的下方选小的数字1，每次只要选下一层相邻的两个数字中较小的一个，似乎就能得到答案了。其实是不对的，贪婪算法可以带到了局部最小，但不能保证每次都带到全局最小，很有可能在其他的分支的底层的数字突然变的超级小，但是贪婪算法已经将其他所有分支剪掉了。所以为了保证我们能得到全局最小，动态规划Dynamic Programming还是不二之选啊。其实这道题和 Dungeon Game 非常的类似，都是用DP来求解的问题。那么其实我们可以不新建dp数组，而是直接复用triangle数组，我们希望一层一层的累加下来，从而使得 triangle[i][j] 是从最顶层到 (i, j) 位置的最小路径和，那么我们如何得到状态转移方程呢？其实也不难，因为每个结点能往下走的只有跟它相邻的两个数字，那么每个位置 (i, j) 也就只能从上层跟它相邻的两个位置过来，也就是 (i-1, j-1) 和 (i-1, j) 这两个位置，那么状态转移方程为：

triangle[i][j] = min(triangle[i - 1][j - 1], triangle[i - 1][j])

我们从第二行开始更新，注意两边的数字直接赋值上一行的边界值，那么最终我们只要在最底层找出值最小的数字，就是全局最小的路径和啦，代码如下：

解法一：

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        for (int i = 1; i < triangle.size(); ++i) {
            for (int j = 0; j < triangle[i].size(); ++j) {
                if (j == 0) {
                    triangle[i][j] += triangle[i - 1][j];
                } else if (j == triangle[i].size() - 1) {
                    triangle[i][j] += triangle[i - 1][j - 1];
                } else {
                    triangle[i][j] += min(triangle[i - 1][j - 1],
triangle[i - 1][j]);
                }
            }
        }
```

```
        return *min_element(triangle.back().begin(),
triangle.back().end());
    }
};
```

这种方法可以通过OJ，但是毕竟修改了原始数组triangle，并不是很理想的方法。在网上搜到一种更好的DP方法，这种方法复制了三角形最后一行，作为用来更新的一位数组。然后逐个遍历这个DP数组，对于每个数字，和它之后的元素比较选择较小的再加上面一行相邻位置的元素做为新的元素，然后一层一层的向上扫描，整个过程和冒泡排序的原理差不多，最后最小的元素都冒到前面，第一个元素即为所求。代码如下：

解法二：

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        vector<int> dp(triangle.back());
        for (int i = (int)triangle.size() - 2; i >= 0; --i) {
            for (int j = 0; j <= i; ++j) {
                dp[j] = min(dp[j], dp[j + 1]) + triangle[i][j];
            }
        }
        return dp[0];
    }
};
```

下面我们来看一个例子，对于输入数组：

-1

2 3

1 -1 -3

5 3 -1 2

下面我们来看DP数组的变换过程（红色数字为每次dp数组中值改变的位置）：

DP：5 3 -1 2

DP：4 3 -1 2

DP：4 -2 -1 2

DP：4 -2 -4 2

DP：0 -2 -4 2

DP：0 -1 -4 2

DP：-2 -1 -4 2

121. Best Time to Buy and Sell Stock

Say you have an array for which the *i* th element is the price of a given stock on day *i*.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

这道题相当简单，感觉达不到Medium的难度，只需要遍历一次数组，用一个变量记录遍历过数中的最小值，然后每次计算当前值和这个最小值之间的差值最为利润，然后每次选较大的利润来更新。当遍历完成后当前利润即为所求，代码如下：

C++ 解法：

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0, buy = INT_MAX;
        for (int price : prices) {
            buy = min(buy, price);
            res = max(res, price - buy);
        }
        return res;
    }
};
```

Java 解法：

```
public class Solution {
    public int maxProfit(int[] prices) {
        int res = 0, buy = Integer.MAX_VALUE;
        for (int price : prices) {
            buy = Math.min(buy, price);
            res = Math.max(res, price - buy);
        }
        return res;
    }
}
```

122. Best Time to Buy and Sell Stock II

Say you have an array for which the *i* th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这道跟之前那道Best Time to Buy and Sell Stock 买卖股票的最佳时间很类似，但都比较容易解答。这道题由于可以无限次买入和卖出。我们都知道炒股想挣钱当然是低价买入高价抛出，那么这里我们只需要从第二天开始，如果当前价格比之前价格高，则把差值加入利润中，因为我们可以昨天买入，今日卖出，若明日价更高的话，还可以今日买入，明日再抛出。以此类推，遍历完整个数组后即可求得最大利润。代码如下：

C++ 解法：

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0, n = prices.size();
        for (int i = 0; i < n - 1; ++i) {
            if (prices[i] < prices[i + 1]) {
                res += prices[i + 1] - prices[i];
            }
        }
        return res;
    }
};
```

Java 解法：

```
public class Solution {
    public int maxProfit(int[] prices) {
        int res = 0;
        for (int i = 0; i < prices.length - 1; ++i) {
            if (prices[i] < prices[i + 1]) {
                res += prices[i + 1] - prices[i];
            }
        }
        return res;
    }
}
```

123. Best Time to Buy and Sell Stock III

Say you have an array for which the *i* th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most *two* transactions.

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
             Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
```

Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
             Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are
             engaging multiple transactions at the same time. You must sell before buying again.
```

Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

这道是买股票的最佳时间系列问题中最难最复杂的一道，前面两道 Best Time to Buy and Sell Stock 和 Best Time to Buy and Sell Stock II 的思路都非常的简洁明了，算法也很简单。而这道是要求最多交易两次，找到最大利润，还是需要用动态规划Dynamic Programming来解，而这里我们需要两个递推公式来分别更新两个变量local和global，参见网友Code Ganker的博客，我们其实可以求至少k次交易的最大利润，找到通解后可以设定 k = 2，即为本题的解答。我们定义local[i][j]为在到达第i天时最多可进行j次交易并且最后一次交易在最后一天卖出的最大利润，此为局部最优。然后我们定义global[i][j]为在到达第i天时最多可进行j次交易的最大利润，此为全局最优。它们的递推式为：

local[i][j] = max(global[i - 1][j - 1] + max(diff, 0), local[i - 1][j] + diff)

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

其中局部最优值是比较前一天并少交易一次的全局最优加上大于0的差值，和前一天的局部最优加上差值中取较大值，而全局最优比较局部最优和前一天的全局最优，代码如下：

解法一：

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if (prices.empty()) return 0;
        int n = prices.size(), g[n][3] = {0}, l[n][3] = {0};
        for (int i = 1; i < prices.size(); ++i) {
            int diff = prices[i] - prices[i - 1];
            for (int j = 1; j <= 2; ++j) {
```

```
                l[i][j] = max(g[i - 1][j - 1] + max(diff, 0), l[i - 1][j] + diff);
                g[i][j] = max(l[i][j], g[i - 1][j]);
            }
        }
        return g[n - 1][2];
    }
};
```

下面这种解法用一维数组来代替二维数组，可以极大的节省了空间，由于覆盖的顺序关系，我们需要j从2到1，这样可以取到正确的g[j-1]值，而非已经被覆盖过的值，参见代码如下：

解法二：

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if (prices.empty()) return 0;
        int g[3] = {0};
        int l[3] = {0};
        for (int i = 0; i < prices.size() - 1; ++i) {
            int diff = prices[i + 1] - prices[i];
            for (int j = 2; j >= 1; --j) {
                l[j] = max(g[j - 1] + max(diff, 0), l[j] + diff);
                g[j] = max(l[j], g[j]);
            }
        }
        return g[2];
    }
};
```

我们如果假设prices数组为1, 3, 2, 9, 那么我们来看每次更新时local 和 global 的值：

第一天两次交易：　　第一天一次交易：

local:　　0 0 0　　local:　　0 0 0

global:　　0 0 0　　global:　　0 0 0

第二天两次交易：　　第二天一次交易：

local:　　0 0 2　　local:　　0 2 2

global:　　0 0 2　　global:　　0 2 2

第三天两次交易：　　第三天一次交易：

local:　　0 2 2　　local:　　0 1 2

global:　　0 2 2　　global:　　0 2 2

第四天两次交易：　　第四天一次交易：

local:　　0 1 9　　local:　　0 8 9

global:　　0 2 9　　global:　　0 8 9

在网友@loveahnee的提醒下，发现了其实上述的递推公式关于local[i][j]的可以稍稍化简一下，我们之前定义的local[i][j]为在到达第i天时最多可进行j次交易并且最后一次交易在最后一天卖出的最大利润，然后网友@fgvlty解释了一下第 i 天卖第 j 支股票的话，一定是下面的一种：

\1. 今天刚买的
那么 Local(i, j) = Global(i-1, j-1)
相当于啥都没干

\2. 昨天买的
那么 Local(i, j) = Global(i-1, j-1) + diff
等于Global(i-1, j-1) 中的交易，加上今天干的那一票

\3. 更早之前买的
那么 Local(i, j) = Local(i-1, j) + diff
昨天别卖了，留到今天卖

但其实第一种情况是不需要考虑的，因为当天买当天卖不会增加利润，完全是重复操作，这种情况可以归纳在global[i-1][j-1]中，所以我们就不需要max(0, diff)了，那么由于两项都加上了diff，所以我们可以把diff抽到max的外面，所以更新后的递推公式为：

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

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

124. Binary Tree Maximum Path Sum

Given a non-empty 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 must contain at least one node and does not need to go through the root.

Example 1:

```
Input: [1,2,3]

       1
      / \
     2   3

Output: 6
```

Example 2:

```
Input: [-10,9,20,null,null,15,7]

   -10
   / \
  9  20
    /  \
   15   7

Output: 42
```

这道求二叉树的最大路径和是一道蛮有难度的题，难就难在起始位置和结束位置可以为任意位置，博主当然是又不会了，于是上网看看大神们的解法，像这种类似树的遍历的题，一般来说都需要用 DFS 来求解，先来看一个简单的例子：

```
    4
   / \
  11 13
 / \
7   2
```

由于这是一个很简单的例子，很容易就能找到最长路径为 7->11->4->13，那么怎么用递归来找出正确的路径和呢？根据以往的经验，树的递归解法一般都是递归到叶节点，然后开始边处理边回溯到根节点。这里就假设此时已经递归到结点7了，其没有左右子节点，如果以结点7为根结点的子树最大路径和就是7。然后回溯到结点 11，如果以结点 11 为根结点的子树，最大路径和为 7+11+2=20。但是当回溯到结点4的时候，对于结点 11 来说，就不能同时取两条路径了，只能取左路径，或者是右路径，所以当根结点是4的时候，那么结点 11 只能取其左子结点7，因为7大于2。所以，对于每个结点来说，要知道经过其左子结点的 path 之和大还是经过右子节点的 path 之和大。递归函数返回值就可以定义为以当前结点为根结点，到叶节点的最大路径之和，然后全局路径最大值放在参数中，用结果 res 来表示。

在递归函数中，如果当前结点不存在，直接返回0。否则就分别对其左右子节点调用递归函数，由于路径和有可能为负数，这里当然不希望加上负的路径和，所以和0相比，取较大的那个，就是要么不加，加就要加正数。然后来更新全局最大值结果 res，就是以左子结点为终点的最大 path 之和加上以右子结点为终点的最大 path 之和，还要加上当前结点值，这样就组成了一个条完整的路径。而返回值是取 left 和 right 中的较大值加上当前结点值，因为返回值的定义是以当前结点为终点的 path 之和，所以只能取 left 和 right 中较大的那个值，而不是两个都要，参见代码如下：

```
class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int res = INT_MIN;
        helper(root, res);
        return res;
    }
    int helper(TreeNode* node, int& res) {
        if (!node) return 0;
        int left = max(helper(node->left, res), 0);
        int right = max(helper(node->right, res), 0);
```

```
        res = max(res, left + right + node->val);
        return max(left, right) + node->val;
    }
};
```

讨论：这道题有一个很好的 Follow up，就是返回这个最大路径，那么就复杂很多，因为这样递归函数就不能返回路径和了，而是返回该路径上所有的结点组成的数组，递归的参数还要保留最大路径之和，同时还需要最大路径结点的数组，然后对左右子节点调用递归函数后得到的是数组，要统计出数组之和，并且跟0比较，如果小于0，和清零，数组清空。然后就是更新最大路径之和跟数组啦，还要拼出来返回值数组，代码长了很多，有兴趣的童鞋可以在评论区贴上你的代码～

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:

1. Only one letter can be changed at a time
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

Note:

- Return an empty list if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

Example 1:

```
Input:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

Output:
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
```

Example 2:

```
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Output: []

Explanation: The endWord "cog" is not in wordList, therefore no possible
transformation.
```

个人感觉这道题是相当有难度的一道题，它比之前那道 Word Ladder 要复杂很多，全场第四低的通过率 12.9% 正说明了这道题的难度，博主也是研究了网上别人的解法很久才看懂，然后照葫芦画瓢的写了出来，下面这种解法的核心思想是 BFS，大概思路如下：目的是找出所有的路径，这里建立一个路径集 paths，用以保存所有路径，然后是起始路径p，在p中先把起始单词放进去。然后定义两个整型变量 level，和 minLevel，其中 level 是记录循环中当前路径的长度，minLevel 是记录最短路径的长度，这样的好处是，如果某条路径的长度超过了已有的最短路径的长度，那么舍弃，这样会提高运行速度，相当于一种剪枝。还要定义一个 HashSet 变量 words，用来记录已经循环过的路径中的词，然后就是 BFS 的核心了，循环路径集 paths 里的内容，取出队首路径，如果该路径长度大于 level，说明字典中的有些词已经存入路径了，如果在路径中重复出现，则肯定不是最短路径，所以需要在字典中将这些词删去，然后将 words 清空，对循环对剪枝处理。然后取出当前路径的最后一个词，对每个字母进行替换并在字典中查找是否存在替换后的新词，这个过程在之前那道 Word Ladder 里面也有。如果替换后的新词在字典中存在，将其加入 words 中，并在原有路径的基础上加上这个新词生成一条新路径，如果这个新词就是结束词，则此新路径为一条完整的路径，加入结果中，并更新 minLevel，若不是结束词，则将新路径加入路径集中继续循环。写了这么多，不知道你看晕了没有，还是看代码吧，这个最有效：

```
class Solution {
public:
    vector<vector<string>> findLadders(string beginWord, string endWord,
vector<string>& wordList) {
        vector<vector<string>> res;
        unordered_set<string> dict(wordList.begin(), wordList.end());
        vector<string> p{beginWord};
        queue<vector<string>> paths;
        paths.push(p);
        int level = 1, minLevel = INT_MAX;
        unordered_set<string> words;
        while (!paths.empty()) {
            auto t = paths.front(); paths.pop();
            if (t.size() > level) {
                for (string w : words) dict.erase(w);
                words.clear();
                level = t.size();
                if (level > minLevel) break;
            }
            string last = t.back();
            for (int i = 0; i < last.size(); ++i) {
                string newLast = last;
                for (char ch = 'a'; ch <= 'z'; ++ch) {
```

```
                    newLast[i] = ch;
                    if (!dict.count(newLast)) continue;
                    words.insert(newLast);
                    vector<string> nextPath = t;
                    nextPath.push_back(newLast);
                    if (newLast == endWord) {
                        res.push_back(nextPath);
                        minLevel = level;
                    } else paths.push(nextPath);
                }
            }
        }
        return res;
    }
};
```

27. 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:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

Example 1:

```
Input:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" ->
"dog" -> "cog",
return its length 5.
```

Example 2:

```
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore no possible
transformation.
```

这道词句阶梯的问题给了我们一个单词字典，里面有一系列很相似的单词，然后给了一个起始单词和一个结束单词，每次变换只能改变一个单词，并且中间过程的单词都必须是单词字典中的单词，让我们求出最短的变化序列的长度。这道题还是挺有难度的，我当然是看了别人的解法才写出来的，这没啥的，从不会到完全掌握才是成长嘛～

当拿到题就懵逼的我们如何才能找到一个科学的探索解题的路径呢，那就是先别去管代码实现，如果让我们肉身解题该怎么做呢？让你将 'hit' 变为 'cog'，那么我们发现这两个单词没有一个相同的字母，所以我们就尝试呗，博主会先将第一个 'h' 换成 'c'，看看 'cit' 在不在字典中，发现不在，那么把第二个 'i' 换成 'o'，看看 'hot' 在不在，发现在，完美！然后尝试 'cot' 或者 'hog'，发现都不在，那么就比较麻烦了，我们没法快速的达到目标单词，需要一些中间状态，但我们怎么知道中间状态是什么。简单粗暴的方法就是brute force，遍历所有的情况，我们将起始单词的每一个字母都用26个字母来替换，比如起始单词 'hit' 就要替换为 'ait', 'bit', 'cit', ... 'yit', 'zit'，将每个替换成的单词都在字典中查找一下，如果有的话，那么说明可能是潜在的路径，要保存下来。那么现在就有个问题，比如我们换到了 'hot' 的时候，此时发现在字典中存在，那么下一步我们是继续试接下来的 'hpt', 'hqt', 'hrt'... 还是直接从 'hot' 的首字母开始换 'aot', 'bot', 'cot' ... 这实际上就是BFS和DFS的区别，到底是广度优先，还是深度优先。讲到这里，不知道你有没有觉得这个跟什么很像？对了，跟迷宫遍历很像啊，你想啊，迷宫中每个点有上下左右四个方向可以走，而这里有26个字母，就是二十六个方向可以走，本质上没有啥区别啊！如果熟悉迷宫遍历的童鞋们应该知道，应该用BFS来求最短路径的长度，这也不难理解啊，DFS相当于一条路走到黑啊，你走的那条道不一定是最短的啊。而BFS相当于一个小圈慢慢的一层一层扩大，相当于往湖里扔个石头，一圈一圈扩大的水波纹那种感觉，当水波纹碰到湖上的树叶时，那么此时水圈的半径就是圆心到树叶的最短距离。脑海中有没有浮现出这个生动的场景呢？

明确了要用BFS，我们可以开始解题了，为了提到字典的查找效率，我们使用HashSet保存所有的单词。然后我们需要一个HashMap，来建立某条路径结尾单词和该路径长度之间的映射，并把起始单词映射为1。既然是BFS，我们需要一个队列queue，把起始单词排入队列中，开始队列的循环，取出队首词，然后对其每个位置上的字符，用26个字母进行替换，如果此时和结尾单词相同了，就可以返回取出词在哈希表中的值加一。如果替换词在字典中存在但在哈希表中不存在，则将替换词排入队列中，并在哈希表中的值映射为之前取出词加一。如果循环完成则返回0，参见代码如下：

解法一：

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>&
wordList) {
        unordered_set<string> wordSet(wordList.begin(), wordList.end());
        if (!wordSet.count(endWord)) return 0;
        unordered_map<string, int> pathCnt{{{beginWord, 1}}};
```

```
        queue<string> q{{beginWord}};
        while (!q.empty()) {
            string word = q.front(); q.pop();
            for (int i = 0; i < word.size(); ++i) {
                string newWord = word;
                for (char ch = 'a'; ch <= 'z'; ++ch) {
                    newWord[i] = ch;
                    if (wordSet.count(newWord) && newWord == endWord) return pathCnt[word] + 1;
                    if (wordSet.count(newWord) && !pathCnt.count(newWord)) {
                        q.push(newWord);
                        pathCnt[newWord] = pathCnt[word] + 1;
                    }
                }
            }
        }
        return 0;
    }
};
```

其实我们并不需要上面解法中的HashMap，由于BFS的遍历机制就是一层一层的扩大的，那么我们只要记住层数就行，然后在while循环中使用一个小trick，加一个for循环，表示遍历完当前队列中的个数后，层数就自增1，这样的话我们就省去了HashMap，而仅仅用一个变量res来记录层数即可，参见代码如下：

解法二：

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        unordered_set<string> wordSet(wordList.begin(), wordList.end());
        if (!wordSet.count(endWord)) return 0;
        queue<string> q{{beginWord}};
        int res = 0;
        while (!q.empty()) {
            for (int k = q.size(); k > 0; --k) {
                string word = q.front(); q.pop();
                if (word == endWord) return res + 1;
                for (int i = 0; i < word.size(); ++i) {
                    string newWord = word;
                    for (char ch = 'a'; ch <= 'z'; ++ch) {
                        newWord[i] = ch;
                        if (wordSet.count(newWord) && newWord != word) {
                            q.push(newWord);
                            wordSet.erase(newWord);
                        }
                    }
```

```
                }
            }
            ++res;
        }
        return 0;
    }
};
```

128. Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

Your algorithm should run in O(*n*) complexity.

Example:

```
Input: [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4].
Therefore its length is 4.
```

这道题要求求最长连续序列，并给定了O(n)复杂度限制，我们的思路是，使用一个集合HashSet存入所有的数字，然后遍历数组中的每个数字，如果其在集合中存在，那么将其移除，然后分别用两个变量pre和next算出其前一个数跟后一个数，然后在集合中循环查找，如果pre在集合中，那么将pre移除集合，然后pre再自减1，直至pre不在集合之中，对next采用同样的方法，那么next-pre-1就是当前数字的最长连续序列，更新res即可。这里再说下，为啥当检测某数字在集合中存在当时候，都要移除数字。这是为了避免大量的重复计算，就拿题目中的例子来说吧，我们在遍历到4的时候，会向下遍历3，2，1，如果都不移除数字的话，遍历到1的时候，还会遍历2，3，4。同样，遍历到3的时候，向上遍历4，向下遍历2，1，等等等。如果数组中有大量的连续数字的话，那么就有大量的重复计算，十分的不高效，所以我们要从HashSet中移除数字，代码如下：

C++ 解法一：

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        int res = 0;
        unordered_set<int> s(nums.begin(), nums.end());
        for (int val : nums) {
            if (!s.count(val)) continue;
            s.erase(val);
            int pre = val - 1, next = val + 1;
            while (s.count(pre)) s.erase(pre--);
            while (s.count(next)) s.erase(next++);
            res = max(res, next - pre - 1);
        }
```

```
        return res;
    }
};
```

Java 解法一：

```
public class Solution {
    public int longestConsecutive(int[] nums) {
        int res = 0;
        Set<Integer> s = new HashSet<Integer>();
        for (int num : nums) s.add(num);
        for (int num : nums) {
            if (s.remove(num)) {
                int pre = num - 1, next = num + 1;
                while (s.remove(pre)) --pre;
                while (s.remove(next)) ++next;
                res = Math.max(res, next - pre - 1);
            }
        }
        return res;
    }
}
```

我们也可以采用哈希表来做，刚开始HashMap为空，然后遍历所有数字，如果该数字不在HashMap中，那么我们分别看其左右两个数字是否在HashMap中，如果在，则返回其哈希表中映射值，若不在，则返回0，虽然我们直接从HashMap中取不存在的映射值，也能取到0，但是一旦去取了，就会自动生成一个为0的映射，那么我们这里再for循环的开头判断如果存在映射就跳过的话，就会出错。然后我们将left+right+1作为当前数字的映射，并更新res结果，同时更新num-left和num-right的映射值。

下面来解释一下为啥要判断如何存在映射的时候要跳过，这是因为一旦某个数字创建映射了，说明该数字已经被处理过了，那么其周围的数字很可能也已经建立好了映射了，如果再遇到之前处理过的数字，再取相邻数字的映射值累加的话，会出错。举个例子，比如数组 [1, 2, 0, 1]，当0执行完以后，HashMap中的映射为 {1->2, 2->3, 0->3}，可以看出此时0和2的映射值都已经为3了，那么如果最后一个1还按照原来的方法处理，随后得到结果就是7，明显不合题意。还有就是，之前说的，为了避免访问不存在的映射值时，自动创建映射，我们使用m.count() 先来检测一下，只有存在映射，我们才从中取值，否则就直接赋值为0，参见代码如下：

C++ 解法二：

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        int res = 0;
        unordered_map<int, int> m;
        for (int num : nums) {
            if (m.count(num)) continue;
            int left = m.count(num - 1) ? m[num - 1] : 0;
            int right = m.count(num + 1) ? m[num + 1] : 0;
            int sum = left + right + 1;
```

```
            m[num] = sum;
            res = max(res, sum);
            m[num - left] = sum;
            m[num + right] = sum;
        }
        return res;
    }
};
```

Java 解法二：

```
public class Solution {
    public int longestConsecutive(int[] nums) {
        int res = 0;
        Map<Integer, Integer> m = new HashMap<Integer, Integer>();
        for (int num : nums) {
            if (m.containsKey(num)) continue;
            int left = m.containsKey(num - 1) ? m.get(num - 1) : 0;
            int right = m.containsKey(num + 1) ? m.get(num + 1) : 0;
            int sum = left + right + 1;
            m.put(num, sum);
            res = Math.max(res, sum);
            m.put(num - left, sum);
            m.put(num + right, sum);
        }
        return res;
    }
}
```

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.

Note: A leaf is a node with no children.

Example:

```
Input: [1,2,3]
    1
   / \
  2   3
Output: 25
Explanation:
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Therefore, sum = 12 + 13 = 25.
```

Example 2:

```
Input: [4,9,0,5,1]
    4
   / \
  9   0
 / \
5   1
Output: 1026
Explanation:
The root-to-leaf path 4->9->5 represents the number 495.
The root-to-leaf path 4->9->1 represents the number 491.
The root-to-leaf path 4->0 represents the number 40.
Therefore, sum = 495 + 491 + 40 = 1026.
```

这道求根到叶节点数字之和的题跟之前的求 Path Sum 很类似，都是利用DFS递归来解，这道题由于不是单纯的把各个节点的数字相加，而是每遇到一个新的子结点的数字，要把父结点的数字扩大10倍之后再相加。如果遍历到叶结点了，就将当前的累加结果sum返回。如果不是，则对其左右子结点分别调用递归函数，将两个结果相加返回即可，参见代码如下:

解法一:

```
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        return sumNumbersDFS(root, 0);
    }
    int sumNumbersDFS(TreeNode* root, int sum) {
        if (!root) return 0;
        sum = sum * 10 + root->val;
        if (!root->left && !root->right) return sum;
        return sumNumbersDFS(root->left, sum) + sumNumbersDFS(root->right, sum);
    }
};
```

我们也可以采用迭代的写法，这里用的是先序遍历的迭代写法，使用栈来辅助遍历，首先将根结点压入栈，然后进行while循环，取出栈顶元素，如果是叶结点，那么将其值加入结果res。如果其右子结点存在，那么其结点值加上当前结点值的10倍，再将右子结点压入栈。同理，若左子结点存在，那么其结点值加上当前结点值的10倍，再将左子结点压入栈，是不是跟之前的 Path Sum 极其类似呢，参见代码如下：

解法二：

```
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        if (!root) return 0;
        int res = 0;
        stack<TreeNode*> st{{root}};
        while (!st.empty()) {
            TreeNode *t = st.top(); st.pop();
            if (!t->left && !t->right) {
                res += t->val;
            }
            if (t->right) {
                t->right->val += t->val * 10;
                st.push(t->right);
            }
            if (t->left) {
                t->left->val += t->val * 10;
                st.push(t->left);
            }
        }
        return res;
    }
};
```

130. Surrounded Regions

Given a 2D board containing `'X'` and `'O'`(the letter O), capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.

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
```

Explanation:

Surrounded regions shouldn't be on the border, which means that any `'O'` on the border of the board are not flipped to `'X'`. Any `'O'` that is not on the border and it is not connected to an `'O'` on the border will be flipped to `'X'`. Two cells are connected if they are adjacent cells connected horizontally or vertically.

这是道关于 XXOO 的题，有点像围棋，将包住的O都变成X，但不同的是边缘的O不算被包围，跟之前那道 Number of Islands 很类似，都可以用 DFS 来解。刚开始我的思路是 DFS 遍历中间的O，如果没有到达边缘，都变成X，如果到达了边缘，将之前变成X的再变回来。但是这样做非常的不方便，在网上看到大家普遍的做法是扫矩阵的四条边，如果有O，则用 DFS 遍历，将所有连着的O都变成另一个字符，比如 $，这样剩下的O都是被包围的，然后将这些O变成X，把$变回O就行了。代码如下：

解法一：

```
class Solution {
public:
    void solve(vector<vector<char> >& board) {
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j) {
                if ((i == 0 || i == board.size() - 1 || j == 0 || j == board[i].size() - 1) && board[i][j] == 'O')
                    solveDFS(board, i, j);
            }
        }
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j) {
                if (board[i][j] == 'O') board[i][j] = 'X';
                if (board[i][j] == '$') board[i][j] = 'O';
            }
        }
    }
    void solveDFS(vector<vector<char> > &board, int i, int j) {
        if (board[i][j] == 'O') {
            board[i][j] = '$';
            if (i > 0 && board[i - 1][j] == 'O')
                solveDFS(board, i - 1, j);
            if (j < board[i].size() - 1 && board[i][j + 1] == 'O')
                solveDFS(board, i, j + 1);
            if (i < board.size() - 1 && board[i + 1][j] == 'O')
                solveDFS(board, i + 1, j);
            if (j > 0 && board[i][j - 1] == 'O')
                solveDFS(board, i, j - 1);
        }
```

```
    }
};
```

很久以前，上面的代码中最后一个 if 中必须是 j > 1 而不是 j > 0，为啥 j > 0 无法通过 OJ 的最后一个大数据集合，博主开始也不知道其中奥秘，直到被另一个网友提醒在本地机子上可以通过最后一个大数据集合，于是博主也写了一个程序来验证，请参见验证 LeetCode Surrounded Regions 包围区域的DFS方法，发现 j > 0 是正确的，可以得到相同的结果。神奇的是，现在用 j > 0 也可以通过 OJ 了。

下面这种解法还是 DFS 解法，只是递归函数的写法稍有不同，但是本质上并没有太大的区别，参见代码如下：

解法二：

```
class Solution {
public:
    void solve(vector<vector<char>>& board) {
        if (board.empty() || board[0].empty()) return;
        int m = board.size(), n = board[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i == 0 || i == m - 1 || j == 0 || j == n - 1) {
                    if (board[i][j] == 'O') dfs(board, i , j);
                }
            }
        }
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == 'O') board[i][j] = 'X';
                if (board[i][j] == '$') board[i][j] = 'O';
            }
        }
    }
    void dfs(vector<vector<char>> &board, int x, int y) {
        int m = board.size(), n = board[0].size();
        vector<vector<int>> dir{{0,-1},{-1,0},{0,1},{1,0}};
        board[x][y] = '$';
        for (int i = 0; i < dir.size(); ++i) {
            int dx = x + dir[i][0], dy = y + dir[i][1];
            if (dx >= 0 && dx < m && dy > 0 && dy < n && board[dx][dy] ==
'O') {
                dfs(board, dx, dy);
            }
        }
    }
};
```

我们也可以使用迭代的解法，但是整体的思路还是一样的，在找到边界上的O后，然后利用队列 queue 进行 BFS 查找和其相连的所有O，然后都标记上美元号。最后的处理还是先把所有的O变成X，然后再把美元号变回O即可，参见代码如下：

解法三：

```
class Solution {
public:
    void solve(vector<vector<char>>& board) {
        if (board.empty() || board[0].empty()) return;
        int m = board.size(), n = board[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i != 0 && i != m - 1 && j != 0 && j != n - 1)
continue;
                if (board[i][j] != 'O') continue;
                board[i][j] = '$';
                queue<int> q{{i * n + j}};
                while (!q.empty()) {
                    int t = q.front(), x = t / n, y = t % n; q.pop();
                    if (x >= 1 && board[x - 1][y] == 'O') {board[x - 1][y]
= '$'; q.push(t - n);}
                    if (x < m - 1 && board[x + 1][y] == 'O') {board[x + 1]
[y] = '$'; q.push(t + n);}
                    if (y >= 1 && board[x][y - 1] == 'O') {board[x][y - 1]
= '$'; q.push(t - 1);}
                    if (y < n - 1 && board[x][y + 1] == 'O') {board[x][y +
1] = '$'; q.push(t + 1);}
                }
            }
        }
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == 'O') board[i][j] = 'X';
                if (board[i][j] == '$') board[i][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*.

Example:

```
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

这又是一道需要用DFS来解的题目，既然题目要求找到所有可能拆分成回文数的情况，那么肯定是所有的情况都要遍历到，对于每一个子字符串都要分别判断一次是不是回文数，那么肯定有一个判断回文数的子函数，还需要一个DFS函数用来递归，再加上原本的这个函数，总共需要三个函数来求解。我们将已经检测好的回文子串放到字符串数组out中，当s遍历完了之后，将out加入结果res中。那么在递归函数中我们必须要知道当前遍历到的位置，用变量start来表示，所以在递归函数中，如果start等于字符串s的长度，说明已经遍历完成，将out加入结果res中，并返回。否则就从start处开始遍历，由于不知道该如何切割，所以我们要遍历所有的切割情况，即一个字符，两个字符，三个字符，等等。。首先判断取出的子串是否是回文串，调用一个判定回文串的子函数即可，这个子函数传入了子串的起始和终止的范围，若子串是回文串，那么我们将其加入out，并且调用递归函数，此时start传入 i+1，之后还要恢复out的状态。

那么，对原字符串的所有子字符串的访问顺序是什么呢，如果原字符串是 abcd, 那么访问顺序为: a -> b -> c -> d -> cd -> bc -> bcd-> ab -> abc -> abcd, 这是对于没有两个或两个以上子回文串的情况。那么假如原字符串是 aabc，那么访问顺序为：a -> a -> b -> c -> bc -> ab -> abc -> aa -> b -> c -> bc -> aab -> aabc，中间当检测到aa时候，发现是回文串，那么对于剩下的bc当做一个新串来检测，于是有 b -> c -> bc，这样扫描了所有情况，即可得出最终答案，代码如下：

解法一：

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> res;
        vector<string> out;
        helper(s, 0, out, res);
        return res;
    }
    void helper(string s, int start, vector<string>& out,
vector<vector<string>>& res) {
        if (start == s.size()) { res.push_back(out); return; }
        for (int i = start; i < s.size(); ++i) {
            if (!isPalindrome(s, start, i)) continue;
            out.push_back(s.substr(start, i - start + 1));
            helper(s, i + 1, out, res);
            out.pop_back();
        }
    }
    bool isPalindrome(string s, int start, int end) {
        while (start < end) {
            if (s[start] != s[end]) return false;
            ++start; --end;
```

```
        }
        return true;
    }
};
```

我们也可以不单独写递归函数，而是使用原函数本身来递归。首先判空，若字符串s为空，则返回一个包有空字符串数组的数组，注意这里不能直接返回一个空数组，后面会解释原因。然后我们从0开始遍历字符串s，因为是使用原函数当递归，所以无法传入起始位置start，所以只能从默认位置0开始，但是我们的输入字符串s是可以用子串来代替的，这样就相当于起始位置start的作用。首先我们还是判断子串是否为回文串，这里的判断子串还是得用一个子函数，由于起点一直是0，所以只需要传一个终点位置即可。如果子串是回文串，则对后面的整个部分调用递归函数，这样我们会得到一个二维数组，是当前子串之后的整个部分拆分为的回文串的所有情况，那么我们只需将当前的回文子串加入到返回的这些所有情况的集合中。现在解释下之前说的为啥当字符串s为空的时候，要返回一个带有空数组的数组，这是因为当子串就是原字符串s的时候，而是还是个回文串，那么后面部分就为空了，若我们对空串调用递归返回的是一个空数组，那么就无法对其进行遍历，则当前的回文串就无法加入到结果res之中，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> res;
        if (s.empty()) return {{}};
        for (int i = 0; i < s.size(); ++i) {
            if (!isPalindrome(s, i + 1)) continue;
            for (auto list : partition(s.substr(i + 1))) {
                list.insert(list.begin(), s.substr(0, i + 1));
                res.push_back(list);
            }
        }
        return res;
    }
    bool isPalindrome(string s, int n) {
        for (int i = 0; i < n / 2; ++i) {
            if (s[i] != s[n - 1 - i]) return false;
        }
        return true;
    }
};
```

下面这种解法是基于解法一的优化，我们可以先建立好字符串s的子串回文的dp数组，光这一部分就可以另出一个道题了 Palindromic Substrings，当我们建立好这样一个二维数组dp，其中 dp[i][j] 表示 [i, j] 范围内的子串是否为回文串，这样就不需要另外的子函数去判断子串是否为回文串了，大大的提高了计算的效率，岂不美哉？！ 递归函数的写法跟解法一中的没啥区别，可以看之前的讲解，参见代码如下：

解法三：

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        int n = s.size();
        vector<vector<string>> res;
        vector<string> out;
        vector<vector<bool>> dp(n, vector<bool>(n));
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j <= i; ++j) {
                if (s[i] == s[j] && (i - j <= 2 || dp[j + 1][i - 1])) {
                    dp[j][i] = true;
                }
            }
        }
        helper(s, 0, dp, out, res);
        return res;
    }
    void helper(string s, int start, vector<vector<bool>>& dp,
vector<string>& out, vector<vector<string>>& res) {
        if (start == s.size()) { res.push_back(out); return; }
        for (int i = start; i < s.size(); ++i) {
            if (!dp[start][i]) continue;
            out.push_back(s.substr(start, i - start + 1));
            helper(s, i + 1, dp, out, res);
            out.pop_back();
        }
    }
};
```

再来看一种迭代的解法，这里还是像上个解法一样建立判断字符串s的子串是否为回文串的dp数组，但建立了一个三维数组的res，这里的res数组其实也可以看作是一个dp数组，其中 res[i] 表示前i个字符组成的子串，即范围 [0, i+1] 内的子串的所有拆分方法，那么最终只要返回 res[n] 极为所求。然后进行for循环，i 从 0 到 n，j 从 0 到 i，这里我们同时更新了两个dp数组，一个是回文串的dp数组，另一个就是结果res数组了，对于区间 [j, i] 的子串，若其是回文串，则 dp[j][i] 更新为 true，并且遍历 res[j] 中的每一种组合，将当前子串加入，并且存入到 res[i+1] 中，参见代码如下：

解法四：

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        int n = s.size();
        vector<vector<vector<string>>> res(n + 1);
        res[0].push_back({});
        vector<vector<bool>> dp(n, vector<bool>(n));
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j <= i; ++j) {
                if (s[i] == s[j] && (i - j <= 2 || dp[j + 1][i - 1])) {
                    dp[j][i] = true;
```

```
                    string cur = s.substr(j, i - j + 1);
                    for (auto list : res[j]) {
                        list.push_back(cur);
                        res[i + 1].push_back(list);
                    }
                }
            }
        }
        return res[n];
    }
};
```

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*.

Example:

```
Input: "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using
1 cut.
```

这道题是让找到把原字符串拆分成回文串的最小切割数，如果我们首先考虑用brute force来做的话就会十分的复杂，因为我们不但要判断子串是否是回文串，而且还要找出最小切割数，情况会非常的多，不好做。所以对于这种玩字符串且是求极值的题，就要祭出旷古神器动态规划Dynamic Programming了，秒天秒地秒空气，DP在手天下我有。好，吹完一波后，开始做题。DP解法的两个步骤，定义dp数组和找状态转移方程。首先来定义dp数组，这里使用最直接的定义方法，一维的dp数组，其中dp[i]表示子串 [0, i] 范围内的最小分割数，那么我们最终要返回的就是 dp[n-1] 了，这里先加个corner case的判断，若s串为空，直接返回0，OJ的test case中并没有空串的检测，但博主认为还是加上比较好，毕竟空串也算是回文串的一种，所以最小分割数为0也说得过去。接下来就是大难点了，如何找出状态转移方程。

如何更新dp[i]呢，前面说过了其表示子串 [0, i] 范围内的最小分割数。那么这个区间的每个位置都可以尝试分割开来，所以就用一个变量j来从0遍历到i，这样就可以把区间 [0, i] 分为两部分，[0, j-1] 和 [j, i]，那么suppose我们已经知道区间 [0, j-1] 的最小分割数 dp[j-1]，因为我们是从前往后更新的，而 j 小于等于 i，所以 dp[j-1] 肯定在 dp[i] 之前就已经算出来了。这样我们就只需要判断区间 [j, i] 内的子串是否为回文串了，是的话，dp[i] 就可以用 1 + dp[j-1] 来更新了。判断子串的方法用的是之前那道 Palindromic Substrings 一样的方法，使用一个二维的dp数组p，其中 p[i][j] 表示区间 [i, j] 内的子串是否为回文串，其状态转移方程为 p[i][j] = (s[i] == s[j]) && p[i+1][j-1]，其中 p[i][j] = true if [i, j]为回文。这样的话，这道题实际相当于同时用了两个DP的方法，确实难度不小呢。

第一个for循环遍历的是i，此时我们现将 dp[i] 初始化为 i，因为对于区间 [0, i]，就算我们每个字母割一刀（怎么听起来像凌迟？！），最多能只用分割 i 次，不需要再多于这个数字。但是可能会变小，所以第二个for循环用 j 遍历区间 [0, j]，根据上面的解释，我们需要验证的是区间 [j, i] 内的子串是否为回文串，那么只要 s[j] == s[i]，并且 i-j < 2 或者 p[j+1][i-1] 为true的话，先更新 p[j][i] 为true，然后在更新

dp[i]，这里需要注意一下corner case，当 j=0 时，我们直接给 dp[i] 赋值为0，因为此时能运行到这，说明 [j, i] 区间是回文串，而 j=0， 则说明 [0, i] 区间内是回文串，这样根本不用分割啊。若 j 大于0，则用 dp[j-1] + 1 来更新 dp[i]，最终返回 dp[n-1] 即可，参见代码如下：

解法一：

```
class Solution {
public:
    int minCut(string s) {
        if (s.empty()) return 0;
        int n = s.size();
        vector<vector<bool>> p(n, vector<bool>(n));
        vector<int> dp(n);
        for (int i = 0; i < n; ++i) {
            dp[i] = i;
            for (int j = 0; j <= i; ++j) {
                if (s[i] == s[j] && (i - j < 2 || p[j + 1][i - 1])) {
                    p[j][i] = true;
                    dp[i] = (j == 0) ? 0 : min(dp[i], dp[j - 1] + 1);
                }
            }
        }
        return dp[n - 1];
    }
};
```

我们也可以反向推，这里的dp数组的定义就刚好跟前面反过来了，dp[i] 表示区间 [i, n-1] 内的最小分割数，所以最终只需要返回 dp[0] 就是区间 [0, n-1] 内的最喜哦啊分割数了，极为所求。然后每次初始化 dp[i] 为 n-1-i 即可，j 的更新范围是 [i, n)，此时我们就只需要用 1 + dp[j+1] 来更新 dp[i] 了，为了防止越界，需要对 j == n-1 的情况单独处理一下，整个思想跟上面的解法一模一样，请参见之前的讲解。

解法二：

```
class Solution {
public:
    int minCut(string s) {
        if (s.empty()) return 0;
        int n = s.size();
        vector<vector<bool>> p(n, vector<bool>(n));
        vector<int> dp(n);
        for (int i = n - 1; i >= 0; --i) {
            dp[i] = n - i - 1;
            for (int j = i; j < n; ++j) {
                if (s[i] == s[j] && (j - i <= 1 || p[i + 1][j - 1])) {
                    p[i][j] = true;
                    dp[i] = (j == n - 1) ? 0 : min(dp[i], dp[j + 1] + 1);
                }
            }
        }
```

```
        return dp[0];
    }
};
```

下面这种解法是论坛上的高分解法，没用使用判断区间 [i, j] 内是否为回文串的二维dp数组，节省了空间。但写法上比之前的解法稍微有些凌乱，也算是个 trade-off 吧。这里还是用的一维dp数组，不过大小初始化为了 n+1，这样其定义就稍稍发生了些变化，dp[i] 表示由s串中前 i 个字母组成的子串的最小分割数，这样 dp[n] 极为最终所求。接下来就要找状态转移方程了。这道题的更新方式比较特别，跟之前的都不一样，之前遍历 i 的时候，都是更新的 dp[i]，这道题更新的却是 dp[i+len+1] 和 dp[i+len+2]，其中 len 是以i为中心，总长度为 2*len + 1 的回文串，比如 bob，此时 i=1，len=1，或者是i为中心之一，总长度为 2*len + 2 的回文串，比如 noon，此时 i=1，len=1。中间两个for循环就是分别更新以 i 为中心且长度为 2*len + 1 的奇数回文串，和以 i 为中心之一且长度为 2*len + 2 的偶数回文串的。i-len 正好是奇数或者偶数回文串的起始位置，由于我们定义的 dp[i] 是区间 [0, i-1] 的最小分割数，所以 dp[i-len] 就是区间 [0, i-len-1] 范围内的最小分割数，那么加上奇数回文串长度 2*len + 1，此时整个区间为 [0, i+len]，即需要更新 dp[i+len+1]。如果是加上偶数回文串的长度 2*len + 2，那么整个区间为 [0, i+len+1]，即需要更新 dp[i+len+2]。这就是分奇偶的状态转移方程，参见代码如下：

解法三：

```
class Solution {
public:
    int minCut(string s) {
        if (s.empty()) return 0;
        int n = s.size();
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = -1;
        for (int i = 0; i < n; ++i) {
            for (int len = 0; i - len >= 0 && i + len < n && s[i - len] == s[i + len]; ++len) {
                dp[i + len + 1] = min(dp[i + len + 1], 1 + dp[i - len]);
            }
            for (int len = 0; i - len >= 0 && i + len + 1 < n && s[i - len] == s[i + len + 1]; ++len) {
                dp[i + len + 2] = min(dp[i + len + 2], 1 + dp[i - len]);
            }
        }
        return dp[n];
    }
};
```

133. Clone Graph

Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node in the graph contains a val (`int`) and a list (`List[Node]`) of its neighbors.

Example:

```
Input:
{"$id":"1","neighbors":[{"$id":"2","neighbors":[{"$ref":"1"},
{"$id":"3","neighbors":[{"$ref":"2"},{"$id":"4","neighbors":[{"$ref":"3"},
{"$ref":"1"}],"val":4}],"val":3}],"val":2},{"$ref":"4"}],"val":1}

Explanation:
Node 1's value is 1, and it has two neighbors: Node 2 and 4.
Node 2's value is 2, and it has two neighbors: Node 1 and 3.
Node 3's value is 3, and it has two neighbors: Node 2 and 4.
Node 4's value is 4, and it has two neighbors: Node 1 and 3.
```

Note:

1. The number of nodes will be between 1 and 100.
2. The undirected graph is a simple graph, which means no repeated edges and no self-loops in the graph.
3. Since the graph is undirected, if node *p* has node *q* as neighbor, then node *q* must have node *p* as neighbor too.
4. You must return the copy of the given node as a reference to the cloned graph.

这道无向图的复制问题和之前的 Copy List with Random Pointer 有些类似，那道题的难点是如何处理每个结点的随机指针，这道题目的难点在于如何处理每个结点的 neighbors，由于在深度拷贝每一个结点后，还要将其所有 neighbors 放到一个 vector 中，而如何避免重复拷贝呢？这道题好就好在所有结点值不同，所以我们可以使用 HashMap 来对应原图中的结点和新生成的克隆图中的结点。对于图的遍历的两大基本方法是深度优先搜索 DFS 和广度优先搜索 BFS，这里我们先使用深度优先搜索DFS来解答此题，在递归函数中，首先判空，然后再看当前的结点是否已经被克隆过了，若在 HashMap 中存在，则直接返回其映射结点。否则就克隆当前结点，并在 HashMap 中建立映射，然后遍历当前结点的所有 neihbor 结点，调用递归函数并且加到克隆结点的 neighbors 数组中即可，代码如下：

解法一：

```
class Solution {
public:
    Node* cloneGraph(Node* node) {
        unordered_map<Node*, Node*> m;
        return helper(node, m);
    }
    Node* helper(Node* node, unordered_map<Node*, Node*>& m) {
        if (!node) return NULL;
        if (m.count(node)) return m[node];
        Node *clone = new Node(node->val);
        m[node] = clone;
        for (Node *neighbor : node->neighbors) {
            clone->neighbors.push_back(helper(neighbor, m));
        }
        return clone;
    }
```

```
};
```

我们也可以使用 BFS 来遍历图，使用队列 queue 进行辅助，还是需要一个 HashMap 来建立原图结点和克隆结点之间的映射。先克隆当前结点，然后建立映射，并加入 queue 中，进行 while 循环。在循环中，取出队首结点，遍历其所有 neighbor 结点，若不在 HashMap 中，我们根据 neigbor 结点值克隆一个新 neighbor 结点，建立映射，并且排入 queue 中。然后将 neighbor 结点在 HashMap 中的映射结点加入到克隆结点的 neighbors 数组中即可，参见代码如下：

解法二：

```
class Solution {
public:
    Node* cloneGraph(Node* node) {
        if (!node) return NULL;
        unordered_map<Node*, Node*> m;
        queue<Node*> q{{node}};
        Node *clone = new Node(node->val);
        m[node] = clone;
        while (!q.empty()) {
            Node *t = q.front(); q.pop();
            for (Node *neighbor : t->neighbors) {
                if (!m.count(neighbor)) {
                    m[neighbor] = new Node(neighbor->val);
                    q.push(neighbor);
                }
                m[t]->neighbors.push_back(m[neighbor]);
            }
        }
        return clone;
    }
};
```

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.

这道转圈加油问题不算很难，只要想通其中的原理就很简单。我们首先要知道能走完整个环的前提是gas的总量要大于cost的总量，这样才会有起点的存在。假设开始设置起点start = 0, 并从这里出发，如果当前的gas值大于cost值，就可以继续前进，此时到下一个站点，剩余的gas加上当前的gas再减去cost，看是否大于0，若大于0，则继续前进。当到达某一站点时，若这个值小于0了，则说明从起点到这个点中间的任何一个点都不能作为起点，则把起点设为下一个点，继续遍历。当遍历完整个环时，当前保存的起点即为所求。代码如下：

解法一：

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int total = 0, sum = 0, start = 0;
        for (int i = 0; i < gas.size(); ++i) {
            total += gas[i] - cost[i];
            sum += gas[i] - cost[i];
            if (sum < 0) {
                start = i + 1;
                sum = 0;
            }
        }
        return (total < 0) ? -1 : start;
    }
};
```

我们也可以从后往前遍历，用一个变量mx来记录出现过的剩余油量的最大值，total记录当前剩余油量的值，start还是记录起点的位置。当total大于mx的时候，说明当前位置可以作为起点，更新start，并且更新mx。为啥呢？因为我们每次total加上的都是当前位置的油量减去消耗，如果这个差值大于0的话，说明当前位置可以当作起点，因为从当前位置到末尾都不会出现油量不够的情况，而一旦差值小于0的话，说明当前位置如果是起点的话，油量就不够，无法走完全程，所以我们不更新起点位置start。最后结束后我们还是看totoa是否大于等于0，如果其小于0的话，说明没有任何一个起点能走完全程，因为总油量都不够，参见代码如下：

解法二：

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int total = 0, mx = -1, start = 0;
        for (int i = gas.size() - 1; i >= 0; --i) {
            total += gas[i] - cost[i];
            if (total > mx) {
                start = i;
                mx = total;
            }
        }
        return (total < 0) ? -1 : start;
    }
};
```

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?

这道题看起来很难，其实解法并没有那么复杂，当然我也是看了别人的解法才做出来的，先来看看两遍遍历的解法，首先初始化每个人一个糖果，然后这个算法需要遍历两遍，第一遍从左向右遍历，如果右边的小盆友的等级高，等加一个糖果，这样保证了一个方向上高等级的糖果多。然后再从右向左遍历一遍，如果相邻两个左边的等级高，而左边的糖果又少的话，则左边糖果数为右边糖果数加一。最后再把所有小盆友的糖果数都加起来返回即可。代码如下：

解法一：

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int res = 0, n = ratings.size();
        vector<int> nums(n, 1);
        for (int i = 0; i < n - 1; ++i) {
            if (ratings[i + 1] > ratings[i]) nums[i + 1] = nums[i] + 1;
        }
        for (int i = n - 1; i > 0; --i) {
            if (ratings[i - 1] > ratings[i]) nums[i - 1] = max(nums[i - 1], nums[i] + 1);
        }
        for (int num : nums) res += num;
        return res;
    }
};
```

下面来看一次遍历的方法，相比于遍历两次的思路简单明了，这种只遍历一次的解法就稍有些复杂了。首先我们给第一个同学一个糖果，那么对于接下来的一个同学就有三种情况：

1. 接下来的同学的rating等于前一个同学，那么给接下来的同学一个糖果就行。
2. 接下来的同学的rating大于前一个同学，那么给接下来的同学的糖果数要比前一个同学糖果数加1。
3. 接下来的同学的rating小于前一个同学，那么我们此时不知道应该给这个同学多少个糖果，需要看后面的情况。

对于第三种情况，我们不确定要给几个，因为要是只给1个的话，那么有可能接下来还有rating更小的同学，总不能一个都不给吧。也不能直接给前一个同学的糖果数减1，有可能给多了，因为如果后面再没人了的话，其实只要给一个就行了。还有就是，如果后面好几个rating越来越小的同学，那么前一个同学的糖果数可能还得追加，以保证最后面的同学至少能有1个糖果。来一个例子吧，四个同学，他们的rating如下：

1 3 2 1

先给第一个rating为1的同学一个糖果，然后从第二个同学开始遍历，第二个同学rating为3，比1大，所以多给一个糖果，第二个同学得到两个糖果。下面第三个同学，他的rating为2，比前一个同学的rating小，如果我们此时给1个糖果的话，那么rating更小的第四个同学就得不到糖果了，所以我们要给第四个同学1个糖果，而给第三个同学2个糖果，此时要给第二个同学追加1个糖果，使其能够比第三个同学的糖果数多至少一个。那么我们就需要统计出多有个连着的同学的rating变小，用变量cnt来记录，找出了最后一个减小的同学，那么就可以往前推，每往前一个加一个糖果，这就是个等差数列，我们可以直接利用求和公式算出这些rating减小的同学的糖果之和。然后我们还要看第一个开始减小的同学的前一个同学需不需要追加糖果，只要比较cnt和pre的大小，pre是之前同学得到的最大糖果数，二者做差加1就是需要追加的糖果数，加到结果res中即可，参见代码如下：

解法二：

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        if (ratings.empty()) return 0;
        int res = 1, pre = 1, cnt = 0;
        for (int i = 1; i < ratings.size(); ++i) {
            if (ratings[i] >= ratings[i - 1]) {
                if (cnt > 0) {
                    res += cnt * (cnt + 1) / 2;
                    if (cnt >= pre) res += cnt - pre + 1;
                    cnt = 0;
                    pre = 1;
                }
                pre = (ratings[i] == ratings[i - 1]) ? 1 : pre + 1;
                res += pre;
            } else {
                ++cnt;
            }
        }
        if (cnt > 0) {
            res += cnt * (cnt + 1) / 2;
            if (cnt >= pre) res += cnt - pre + 1;
        }
        return res;
    }
};
```

136. Single Number

Given a non-empty 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?

Example 1:

```
Input: [2,2,1]
Output: 1
```

Example 2:

```
Input: [4,1,2,1,2]
Output: 4
```

这道题给了我们一个非空的整数数组，说是除了一个数字之外所有的数字都正好出现了两次，让我们找出这个只出现一次的数字。题目中让我们在线性的时间复杂度内求解，那么一个非常直接的思路就是使用 HashSet，利用其常数级的查找速度。遍历数组中的每个数字，若当前数字已经在 HashSet 中了，则将 HashSet 中的该数字移除，否则就加入 HashSet。这相当于两两抵消了，最终凡事出现两次的数字都被移除了 HashSet，唯一剩下的那个就是单独数字了，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        unordered_set<int> st;
        for (int num : nums) {
            if (st.count(num)) st.erase(num);
            else st.insert(num);
        }
        return *st.begin();
    }
};
```

Java 解法一：

```java
class Solution {
    public int singleNumber(int[] nums) {
        Set<Integer> st = new HashSet<>();
        for (int num : nums) {
            if (!st.add(num)) st.remove(num);
        }
        return st.iterator().next();
    }
}
```

题目中让我们不使用额外空间来做，本来是一道非常简单的题，但是由于加上了时间复杂度必须是 O(n)，并且空间复杂度为 O(1)，使得不能用排序方法，也不能使用 HashSet 数据结构。那么只能另辟蹊径，需要用位操作 Bit Operation 来解此题，这个解法如果让我想，肯定想不出来，因为谁会想到用逻辑异或来解题呢。逻辑异或的真值表为：

异或运算 ? 的真值表如下：

A	B	⊕
F	F	F
F	T	T
T	F	T
T	T	F

由于数字在计算机是以二进制存储的，每位上都是0或1，如果我们把两个相同的数字异或，0与0 '异或' 是0，1与1 '异或' 也是0，那么我们会得到0。根据这个特点，我们把数组中所有的数字都 '异或' 起来，则每对相同的数字都会得0，然后最后剩下来的数字就是那个只有1次的数字。这个方法确实很赞，但是感觉一般人不会往 '异或' 上想，绝对是为CS专业的同学设计的好题呀，赞一个~~

C++ 解法二：

```cpp
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int res = 0;
        for (auto num : nums) res ^= num;
        return res;
    }
};
```

Java 解法二：

```java
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for (int num : nums) res ^= num;
        return res;
    }
}
```

137. Single Number II

Given a non-empty array of integers, every element appears *three* times except for one, which appears exactly once. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Example 1:

```
Input: [2,2,3,2]
Output: 3
```

Example 2:

```
Input: [0,1,0,1,0,1,99]
Output: 99
```

这道题是之前那道 Single Number 的延伸，那道题的解法就比较独特，是利用计算机按位储存数字的特性来做的，这道题就是除了一个单独的数字之外，数组中其他的数字都出现了三次，还是要利用位操作 Bit Manipulation 来解。可以建立一个 32 位的数字，来统计每一位上1出现的个数，如果某一位上为1的话，那么如果该整数出现了三次，对3取余为0，这样把每个数的对应位都加起来对3取余，最终剩下来的那个数就是单独的数字。代码如下：

解法一：

```java
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int res = 0;
        for (int i = 0; i < 32; ++i) {
            int sum = 0;
            for (int j = 0; j < nums.size(); ++j) {
                sum += (nums[j] >> i) & 1;
            }
            res |= (sum % 3) << i;
        }
```

```
        return res;
    }
};
```

还有一种解法，思路很相似，用3个整数来表示 INT 的各位的出现次数情况，one 表示出现了1次，two 表示出现了2次。当出现3次的时候该位清零。最后答案就是one的值。

1. `ones` 代表第 ith 位只出现一次的掩码变量
2. `twos` 代表第 ith 位只出现两次次的掩码变量
3. `threes` 代表第 ith 位只出现三次的掩码变量

假设现在有一个数字1，更新 one 的方法就是 '亦或' 这个1，则 one 就变成了1，而 two 的更新方法是用上一个状态下的 one 去 '与' 上数字1，然后 '或' 上这个结果，这样假如之前 one 是1，那么此时 two 也会变成1，这 make sense，因为说明是当前位遇到两个1了；反之如果之前 one 是0，那么现在 two 也就是0。注意更新的顺序是先更新 two，再更新 one，不理解的话只要带个只有一个数字1的输入数组看一下就不难理解了。然后更新 three，如果此时 one 和 two 都是1了，由于先更新的 two，再更新的 one，two 为1，说明此时至少有两个数字1了，而此时 one 为1，说明了此时已经有了三个数字1，这块要仔细想清楚，因为 one 是要 '亦或' 一个1的，值能为1，说明之前 one 为0，实际情况是，当第二个1来的时候，two 先更新为1，此时 one 再更新为0，下面 three 就是0了，那么 '与' 上t hree 的相反数1不会改变 one 和 two 的值；那么当第三个1来的时候，two 还是1，此时 one 就更新为1了，那么 three 就更新为1了，此时就要清空 one 和 two 了，让它们 '与' 上 three 的相反数0即可，最终结果将会保存在 one 中，参见代码如下：

解法二：

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int one = 0, two = 0, three = 0;
        for (int i = 0; i < nums.size(); ++i) {
            two |= one & nums[i];
            one ^= nums[i];
            three = one & two;
            one &= ~three;
            two &= ~three;
        }
        return one;
    }
};
```

下面这种解法思路也十分巧妙，根据上面解法的思路，我们把数组中数字的每一位累加起来对3取余，剩下的结果就是那个单独数组该位上的数字，由于累加的过程都要对3取余，那么每一位上累加的过程就是 0->1->2->0，换成二进制的表示为 00->01->10->00，可以写出对应关系：

00 (+) 1 = 01

01 (+) 1 = 10

10 (+) 1 = 00 (mod 3)

用 ab 来表示开始的状态，对于加1操作后，得到的新状态的 ab 的算法如下：

b = b xor r & ~a;

a = a xor r & ~b;

这里的 ab 就是上面的三种状态 00，01，10 的十位和各位，刚开始的时候，a和b都是0，当此时遇到数字1的时候，b更新为1，a更新为0，就是 01 的状态；再次遇到1的时候，b更新为0，a更新为1，就是 10 的状态；再次遇到1的时候，b更新为0，a更新为0，就是 00 的状态，相当于重置了；最后的结果保存在b中，明白了上面的分析过程，就能写出代码如下；

解法三：

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int a = 0, b = 0;
        for (int i = 0; i < nums.size(); ++i) {
            b = (b ^ nums[i]) & ~a;
            a = (a ^ nums[i]) & ~b;
        }
        return 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.

Example 1:

```
Input:
{"$id":"1","next":{"$id":"2","next":null,"random":
{"$ref":"2"},"val":2},"random":{"$ref":"2"},"val":1}

Explanation:
Node 1's value is 1, both of its next and random pointer points to Node 2.
Node 2's value is 2, its next pointer points to null and its random pointer
points to itself.
```

Note:

1. You must return the copy of the given head as a reference to the cloned list.

这道链表的深度拷贝题的难点就在于如何处理随机指针的问题，由于每一个节点都有一个随机指针，这个指针可以为空，也可以指向链表的任意一个节点，如果在每生成一个新节点给其随机指针赋值时，都要去遍历原链表的话，OJ 上肯定会超时，所以可以考虑用 HashMap 来缩短查找时间，第一遍遍历生成所有新节点时同时建立一个原节点和新节点的 HashMap，第二遍给随机指针赋值时，查找时间是常数级。代码如下：

解法一：

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;
        Node *res = new Node(head->val, nullptr, nullptr);
        Node *node = res, *cur = head->next;
        unordered_map<Node*, Node*> m;
        m[head] = res;
        while (cur) {
            Node *t = new Node(cur->val, nullptr, nullptr);
            node->next = t;
            m[cur] = t;
            node = node->next;
            cur = cur->next;
        }
        node = res; cur = head;
        while (cur) {
            node->random = m[cur->random];
            node = node->next;
            cur = cur->next;
        }
        return res;
    }
};
```

我们可以使用递归的解法，写起来相当的简洁，还是需要一个 HashMap 来建立原链表结点和拷贝链表结点之间的映射。在递归函数中，首先判空，若为空，则返回空指针。然后就是去 HashMap 中查找是否已经在拷贝链表中存在了该结点，是的话直接返回。否则新建一个拷贝结点 res，然后建立原结点和该拷贝结点之间的映射，然后就是要给拷贝结点的 next 和 random 指针赋值了，直接分别调用递归函数即可，参见代码如下：

解法二：

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        unordered_map<Node*, Node*> m;
        return helper(head, m);
    }
    Node* helper(Node* node, unordered_map<Node*, Node*>& m) {
        if (!node) return nullptr;
```

```
        if (m.count(node)) return m[node];
        Node *res = new Node(node->val, nullptr, nullptr);
        m[node] = res;
        res->next = helper(node->next, m);
        res->random = helper(node->random, m);
        return res;
    }
};
```

当然，如果使用 HashMap 占用额外的空间，如果这道题限制了空间的话，就要考虑别的方法。下面这个方法很巧妙，可以分为以下三个步骤：

1. 在原链表的每个节点后面拷贝出一个新的节点。
2. 依次给新的节点的随机指针赋值，而且这个赋值非常容易 cur->next->random = cur->random->next。
3. 断开链表可得到深度拷贝后的新链表。

举个例子来说吧，比如原链表是 1(2) -> 2(3) -> 3(1)，括号中是其 random 指针指向的结点，那么这个解法是首先比遍历一遍原链表，在每个结点后拷贝一个同样的结点，但是拷贝结点的 random 指针仍为空，则原链表变为 1(2) -> 1(null) -> 2(3) -> 2(null) -> 3(1) -> 3(null)。然后第二次遍历，是将拷贝结点的 random 指针赋上正确的值，则原链表变为 1(2) -> 1(2) -> 2(3) -> 2(3) -> 3(1) -> 3(1)，注意赋值语句为：

cur->next->random = cur->random->next;

这里的 cur 是原链表中结点，cur->next 则为拷贝链表的结点，cur->next->random 则为拷贝链表的 random 指针。cur->random 为原链表结点的 random 指针指向的结点，因为其指向的还是原链表的结点，所以我们要再加个 next，才能指向拷贝链表的结点。最后再遍历一次，就是要把原链表和拷贝链表断开即可，参见代码如下：

解法二：

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;
        Node *cur = head;
        while (cur) {
            Node *t = new Node(cur->val, nullptr, nullptr);
            t->next = cur->next;
            cur->next = t;
            cur = t->next;
        }
        cur = head;
        while (cur) {
            if (cur->random) cur->next->random = cur->random->next;
            cur = cur->next->next;
        }
        cur = head;
```

```
        Node *res = head->next;
        while (cur) {
            Node *t = cur->next;
            cur->next = t->next;
            if (t->next) t->next = t->next->next;
            cur = cur->next;
        }
        return res;
    }
};
```

139. Word Break

Given a non-empty string *s* and a dictionary *wordDict* containing a list of non-empty words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

Note:

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet
code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple
pen apple".
             Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
Output: false
```

这道拆分词句问题是看给定的词句能分被拆分成字典里面的内容，这是一道很经典的题目，解法不止一种，考察的范围很广，属于我们必须要熟练掌握的题目。那么先来想 brute force 的解法，就拿例子1来分析，如果字典中只有两个单词，我们怎么去判断，是不是可以将原字符串s分成任意两段，然后再看分成的单词是否在字典中。注意这道题说是单词可以重复使用，所以可以分成任意段，而且字典中的单词可以有很多个，这就增加了题目的难度，很多童鞋就在这里迷失了，毫无头绪。那么，就由博主来给

各位指点迷津吧（此处应有掌声👏）。

既然要分段，看子字符串是否在字典中，由于给定的字典是数组（之前还是 HashSet 呢），那么我们肯定不希望每次查找都需要遍历一遍数组，费劲！还是把字典中的所有单词都存入 HashSet 中吧，这样我们就有了常数时间级的查找速度，perfect！好，我们得开始给字符串分段了，怎么分，只能一个一个分了，先看第一个字母是否在字典中，如果不在的话，好办，说明这种分法肯定是错的。问题是在的话，后面的那部分怎么处理，难道还用 for 循环？咱也不知道还要分多少段，怎么用 for 循环。对于这种不知道怎么处理的情况，一个万能的做法是丢给递归函数，让其去递归求解，这里我们 suppose 递归函数会返回我们一个正确的值，如果返回的是 true 的话，表明我们现在分成的两段都在字典中，我们直接返回 true 即可，因为只要找出一种情况就行了。这种调用递归函数的方法就是 brute force 的解法，我们遍历了所有的情况，优点是写法简洁，思路清晰，缺点是存在大量的重复计算，被 OJ 啪啪打脸。所以我们需要进行优化，使用记忆数组 memo 来保存所有已经计算过的结果，再下次遇到的时候，直接从 cache 中取，而不是再次计算一遍。这种使用记忆数组 memo 的递归写法，和使用 dp 数组的迭代写法，乃解题的两大神器，凡事能用 dp 解的题，一般也有用记忆数组的递归解法，好似一对形影不离的好基友～关于 dp 解法，博主会在下文中讲解。这里我们的记忆数组 memo[i] 定义为范围为 [i, n] 的子字符串是否可以拆分，初始化为 -1，表示没有计算过，如果可以拆分，则赋值为1，反之为0。在之前讲 brute force 解法时，博主提到的是讲分成两段的后半段的调用递归函数，我们也可以不取出子字符串，而是用一个 start 变量，来标记分段的位置，这样递归函数中只需要从 start 的位置往后遍历即可，在递归函数更新记忆数组 memo 即可，参见代码如下：

解法一：

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<int> memo(s.size(), -1);
        return check(s, wordSet, 0, memo);
    }
    bool check(string s, unordered_set<string>& wordSet, int start, vector<int>& memo) {
        if (start >= s.size()) return true;
        if (memo[start] != -1) return memo[start];
        for (int i = start + 1; i <= s.size(); ++i) {
            if (wordSet.count(s.substr(start, i - start)) && check(s, wordSet, i, memo)) {
                return memo[start] = 1;
            }
        }
        return memo[start] = 0;
    }
};
```

这道题其实还是一道经典的 DP 题目，也就是动态规划 Dynamic Programming。博主曾经说玩子数组或者子字符串且求极值的题，基本就是 DP 没差了，虽然这道题没有求极值，但是玩子字符串也符合 DP 的状态转移的特点。把一个人的温暖转移到另一个人的胸膛... 咳咳，跑错片场了，那是爱情转移～强行拉回，DP 解法的两大难点，定义 dp 数组跟找出状态转移方程，先来看 dp 数组的定义，这里我们就用一个一维的 dp 数组，其中 dp[i] 表示范围 [0, i) 内的子串是否可以拆分，注意这里 dp 数组的长度

比s串的长度大1，是因为我们要 handle 空串的情况，我们初始化 dp[0] 为 true，然后开始遍历。注意这里我们需要两个 for 循环来遍历，因为此时已经没有递归函数了，所以我们必须要遍历所有的子串，我们用j把 [0, i) 范围内的子串分为了两部分，[0, j) 和 [j, i)，其中范围 [0, j) 就是 dp[j]，范围 [j, i) 就是 s.substr(j, i-j)，其中 dp[j] 是之前的状态，我们已经算出来了，可以直接取，只需要在字典中查找 s.substr(j, i-j) 是否存在了，如果二者均为 true，将 dp[i] 赋为 true，并且 break 掉，此时就不需要再用 j去分 [0, i) 范围了，因为 [0, i) 范围已经可以拆分了。最终我们返回 dp 数组的最后一个值，就是整个数组是否可以拆分的布尔值了，代码如下：

解法二：

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size() + 1);
        dp[0] = true;
        for (int i = 0; i < dp.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (dp[j] && wordSet.count(s.substr(j, i - j))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp.back();
    }
};
```

下面我们从题目中给的例子来分析：

l

le e

lee ee e

leet

leetc eetc etc tc c

leetco eetco etco tco co o

leetcod eetcod etcod tcod cod od d

leetcode eetcode etcode tcode **code**

T F F F T F F F T

我们知道算法的核心思想是逐行扫描，每一行再逐个字符扫描，每次都在组合出一个新的字符串都要到字典里去找，如果有的话，则跳过此行，继续扫描下一行。

既然 DFS 都可以解题，那么 BFS 也就坐不住了，也要出来蹦跶一下。其实本质跟递归的解法没有太大的区别，递归解法在调用递归的时候，原先的状态被存入了栈中，这里 BFS 是存入了队列中，使用 visited 数组来标记已经算过的位置，作用跟 memo 数组一样，从队列中取出一个位置进行遍历，把可以拆分的新位置存入队列中，遍历完成后标记当前位置，然后再到队列中去取即可，参见代码如下：

解法三：

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<bool> visited(s.size());
        queue<int> q{{0}};
        while (!q.empty()) {
            int start = q.front(); q.pop();
            if (!visited[start]) {
                for (int i = start + 1; i <= s.size(); ++i) {
                    if (wordSet.count(s.substr(start, i - start))) {
                        q.push(i);
                        if (i == s.size()) return true;
                    }
                }
                visited[start] = true;
            }
        }
        return false;
    }
};
```

140. Word Break II

Given a non-empty string *s* and a dictionary *wordDict* containing a list of non-empty words, add spaces in *s* to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

Note:

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

Example 1:

```
Input:
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
Output:
[
  "cats and dog",
  "cat sand dog"
]
```

Example 2:

```
Input:
s = "pineapplepenapple"
wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]
Output:
[
  "pine apple pen apple",
  "pineapple pen apple",
  "pine applepen apple"
]
Explanation: Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input:
s = "catsandog"
wordDict = ["cats", "dog", "sand", "and", "cat"]
Output:
[]
```

这道题是之前那道Word Break 拆分词句的拓展，那道题只让我们判断给定的字符串能否被拆分成字典中的词，而这道题加大了难度，让我们求出所有可以拆分成的情况，就像题目中给的例子所示。之前的版本中字典wordDict的数据类型是HashSet，现在的不知为何改成了数组vector，而且博主看到第二个例子就笑了，PPAP么，哈哈～

根据老夫行走江湖多年的经验，像这种返回结果要列举所有情况的题，十有八九都是要用递归来做的。当我们一时半会没有啥思路的时候，先不要考虑代码如何实现，如果就给你一个s和wordDict，不看Output的内容，你会怎么找出结果。比如对于例子1，博主可能会先扫一遍wordDict数组，看有没有单词可以当s的开头，那么我们可以发现cat和cats都可以，比如我们先选了cat，那么此时s就变成了"sanddog"，我们再在数组里找单词，发现了sand可以，最后剩一个dog，也在数组中，于是一个结果就出来了。然后回到开头选cats的话，那么此时s就变成了 "anddog"，我们再在数组里找单词，发现了and可以，最后剩一个dog，也在数组中，于是另一个结果也就出来了。那么这个查询的方法很适合用递归来实现，因为s改变后，查询的机制并不变，很适合调用递归函数。再者，我们要明确的是，如果不用记忆数组做减少重复计算的优化，那么递归方法跟brute force没什么区别，大概率无法通过OJ。所以我们要避免重复计算，如何避免呢，还是看上面的分析，如果当s变成 "sanddog"的时候，那么此时我们知道其可以拆分成sand和dog，当某个时候如果我们又遇到了这个 "sanddog"的时候，我们难道还需要再调用递归算一遍吗，当然不希望啦，所以我们要将这个中间结果保存起来，由于我们必须要同时

保存s和其所有的拆分的字符串，那么可以使用一个HashMap，来建立二者之间的映射，那么在递归函数中，我们首先检测当前s是否已经有映射，有的话直接返回即可，如果s为空了，我们如何处理呢，题目中说了给定的s不会为空，但是我们递归函数处理时s是会变空的，这时候我们是直接返回空集吗，这里有个小trick，我们其实放一个空字符串返回，为啥要这么做呢？我们观察题目中的Output，发现单词之间是有空格，而最后一个单词后面没有空格，所以这个空字符串就起到了标记当前单词是最后一个，那么我们就不要再加空格了。接着往下看，我们遍历wordDict数组，如果某个单词是s字符串中的开头单词的话，我们对后面部分调用递归函数，将结果保存到rem中，然后遍历里面的所有字符串，和当前的单词拼接起来，这里就用到了我们前面说的trick。for循环结束后，记得返回结果res之前建立其和s之间的映射，方便下次使用，参见代码如下：

解法一：

```
class Solution {
public:
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        unordered_map<string, vector<string>> m;
        return helper(s, wordDict, m);
    }
    vector<string> helper(string s, vector<string>& wordDict, unordered_map<string, vector<string>>& m) {
        if (m.count(s)) return m[s];
        if (s.empty()) return {""};
        vector<string> res;
        for (string word : wordDict) {
            if (s.substr(0, word.size()) != word) continue;
            vector<string> rem = helper(s.substr(word.size()), wordDict, m);
            for (string str : rem) {
                res.push_back(word + (str.empty() ? "" : " ") + str);
            }
        }
        return m[s] = res;
    }
};
```

我们也可以将将主函数本身当作递归函数，这样就不用单独的使用一个递归函数了，不过我们的HashMap必须是全局了，写在外部就好了，参见代码如下：

解法二：

```
class Solution {
public:
    unordered_map<string, vector<string>> m;
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        if (m.count(s)) return m[s];
        if (s.empty()) return {""};
        vector<string> res;
        for (string word : wordDict) {
            if (s.substr(0, word.size()) != word) continue;
```

```
            vector<string> rem = wordBreak(s.substr(word.size()), wordDict);
            for (string str : rem) {
                res.push_back(word + (str.empty() ? "" : " ") + str);
            }
        }
        return m[s] = res;
    }
};
```

143. Reorder List

Given a singly linked list *L* : *L* 0→ *L* 1→…→ *L n* -1→ *L* n,
reorder it to: *L* 0→ *L n* → *L* 1→ *L n* -1→ *L* 2→ *L n* -2→…

You may not modify the values in the list's nodes, only nodes itself may be changed.

Example 1:

```
Given 1->2->3->4, reorder it to 1->4->2->3.
```

Example 2:

```
Given 1->2->3->4->5, reorder it to 1->5->2->4->3.
```

这道链表重排序问题可以拆分为以下三个小问题：

1. 使用快慢指针来找到链表的中点，并将链表从中点处断开，形成两个独立的链表。
2. 将第二个链翻转。
3. 将第二个链表的元素间隔地插入第一个链表中。

解法一：

```
class Solution {
public:
    void reorderList(ListNode *head) {
        if (!head || !head->next || !head->next->next) return;
        ListNode *fast = head, *slow = head;
        while (fast->next && fast->next->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        ListNode *mid = slow->next;
        slow->next = NULL;
        ListNode *last = mid, *pre = NULL;
        while (last) {
            ListNode *next = last->next;
```

```
            last->next = pre;
            pre = last;
            last = next;
        }
        while (head && pre) {
            ListNode *next = head->next;
            head->next = pre;
            pre = pre->next;
            head->next->next = next;
            head = next;
        }
    }
};
```

我们尝试着看能否写法上简洁一些，上面的第二步是将后半段链表翻转，那么我们其实可以借助栈的后进先出的特性来做，如果我们按顺序将所有的结点压入栈，那么出栈的时候就可以倒序了，实际上就相当于翻转了链表。由于只需将后半段链表翻转，那么我们就要控制出栈结点的个数，还好栈可以直接得到结点的个数，我们减1除以2，就是要出栈结点的个数了。然后我们要做的就是将每次出栈的结点隔一个插入到正确的位置，从而满足题目中要求的顺序，链表插入结点的操作就比较常见了，这里就不多解释了，最后记得断开栈顶元素后面的结点，比如对于 1->2->3->4，栈顶只需出一个结点4，然后加入原链表之后为 1->4->2->3->(4)，因为在原链表中结点3之后是连着结点4的，虽然我们将结点4取出插入到结点1和2之间，但是结点3后面的指针还是连着结点4的，所以我们要断开这个连接，这样才不会出现环，由于此时结点3在栈顶，所以我们直接断开栈顶结点即可，参见代码如下：

解法二：

```
class Solution {
public:
    void reorderList(ListNode *head) {
        if (!head || !head->next || !head->next->next) return;
        stack<ListNode*> st;
        ListNode *cur = head;
        while (cur) {
            st.push(cur);
            cur = cur->next;
        }
        int cnt = ((int)st.size() - 1) / 2;
        cur = head;
        while (cnt-- > 0) {
            auto t = st.top(); st.pop();
            ListNode *next = cur->next;
            cur->next = t;
            t->next = next;
            cur = next;
        }
        st.top()->next = NULL;
    }
};
```

146. LRU Cache

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

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

Follow up:
Could you do both operations in O(1) time complexity?

Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

这道题让我们实现一个 LRU 缓存器，LRU 是 Least Recently Used 的简写，就是最近最少使用的意思。那么这个缓存器主要有两个成员函数，get 和 put，其中 get 函数是通过输入 key 来获得 value，如果成功获得后，这对 (key, value) 升至缓存器中最常用的位置（顶部），如果 key 不存在，则返回 -1。而 put 函数是插入一对新的 (key, value)，如果原缓存器中有该 key，则需要先删除掉原有的，将新的插入到缓存器的顶部。如果不存在，则直接插入到顶部。若加入新的值后缓存器超过了容量，则需要删掉一个最不常用的值，也就是底部的值。具体实现时我们需要三个私有变量，cap, l和m，其中 cap 是缓存器的容量大小，l是保存缓存器内容的列表，m是 HashMap，保存关键值 key 和缓存器各项的迭代器之间映射，方便我们以 O(1) 的时间内找到目标项。

然后我们再来看 get 和 put 如何实现，get 相对简单些，我们在 HashMap 中查找给定的 key，若不存在直接返回 -1。如果存在则将此项移到顶部，这里我们使用 C++ STL 中的函数 splice，专门移动链表中的一个或若干个结点到某个特定的位置，这里我们就只移动 key 对应的迭代器到列表的开头，然后返回 value。这里再解释一下为啥 HashMap 不用更新，因为 HashMap 的建立的是关键值 key 和缓存列表中的迭代器之间的映射，get 函数是查询函数，如果关键值 key 不在 HashMap，那么不需要更新。如果在，我们需要更新的是该 key-value 键值对儿对在缓存列表中的位置，而 HashMap 中还是这个 key 跟键值对儿的迭代器之间的映射，并不需要更新什么。

对于 put，我们也是现在 HashMap 中查找给定的 key，如果存在就删掉原有项，并在顶部插入新来项，然后判断是否溢出，若溢出则删掉底部项(最不常用项)。代码如下：

```
class LRUCache{
```

```
public:
    LRUCache(int capacity) {
        cap = capacity;
    }

    int get(int key) {
        auto it = m.find(key);
        if (it == m.end()) return -1;
        l.splice(l.begin(), l, it->second);
        return it->second->second;
    }

    void put(int key, int value) {
        auto it = m.find(key);
        if (it != m.end()) l.erase(it->second);
        l.push_front(make_pair(key, value));
        m[key] = l.begin();
        if (m.size() > cap) {
            int k = l.rbegin()->first;
            l.pop_back();
            m.erase(k);
        }
    }

private:
    int cap;
    list<pair<int, int>> l;
    unordered_map<int, list<pair<int, int>>::iterator> m;
};
```

147. Insertion Sort List

Sort a linked list using insertion sort.

A graphical example of insertion sort. The partial sorted list (black) initially contains only the first element in the list.
With each iteration one element (red) is removed from the input data and inserted in-place into the sorted list

Algorithm of Insertion Sort:

1. Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.
2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.
3. It repeats until no input elements remain.

Example 1:

```
Input: 4->2->1->3
Output: 1->2->3->4
```

Example 2:

```
Input: -1->5->3->4->0
Output: -1->0->3->4->5
```

链表的插入排序实现原理很简单，就是一个元素一个元素的从原链表中取出来，然后按顺序插入到新链表中，时间复杂度为 O(n2)，是一种效率并不是很高的算法，但是空间复杂度为 O(1)，以高时间复杂度换取了低空间复杂度，参见代码如下：

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode *dummy = new ListNode(-1), *cur = dummy;
        while (head) {
            ListNode *t = head->next;
            cur = dummy;
            while (cur->next && cur->next->val <= head->val) {
                cur = cur->next;
            }
            head->next = cur->next;
            cur->next = head;
            head = t;
        }
        return dummy->next;
    }
};
```

148. Sort List

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

Example 1:

```
Input: 4->2->1->3
Output: 1->2->3->4
```

Example 2:

```
Input: -1->5->3->4->0
Output: -1->0->3->4->5
```

常见排序方法有很多，插入排序，选择排序，堆排序，快速排序，冒泡排序，归并排序，桶排序等等。。它们的时间复杂度不尽相同，而这里题目限定了时间必须为O(nlgn)，符合要求只有快速排序，归并排序，堆排序，而根据单链表的特点，最适于用归并排序。为啥呢？这是由于链表自身的特点决定的，由于不能通过坐标来直接访问元素，所以快排什么的可能不太容易实现（但是被评论区的大神们打脸，还是可以实现的），堆排序的话，如果让新建结点的话，还是可以考虑的，若只能交换结点，最好还是不要用。而归并排序（又称混合排序）因其可以利用递归来交换数字，天然适合链表这种结构。归并排序的核心是一个 merge() 函数，其主要是合并两个有序链表，这个在 LeetCode 中也有单独的题目 Merge Two Sorted Lists。由于两个链表是要有序的才能比较容易 merge，那么对于一个无序的链表，如何才能拆分成有序的两个链表呢？我们从简单来想，什么时候两个链表一定都是有序的？就是当两个链表各只有一个结点的时候，一定是有序的。而归并排序的核心其实是分治法 Divide and Conquer，就是将链表从中间断开，分成两部分，左右两边再分别调用排序的递归函数 sortList()，得到各自有序的链表后，再进行 merge()，这样整体就是有序的了。因为子链表的递归函数中还是会再次拆成两半，当拆到链表只有一个结点时，无法继续拆分了，而这正好满足了前面所说的“一个结点的时候一定是有序的”，这样就可以进行 merge 了。然后再回溯回去，每次得到的都是有序的链表，然后进行 merge，直到还原整个长度。这里将链表从中间断开的方法，采用的就是快慢指针，大家可能对快慢指针找链表中的环比较熟悉，其实找链表中的中点同样好使，因为快指针每次走两步，慢指针每次走一步，当快指针到达链表末尾时，慢指针正好走到中间位置，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *slow = head, *fast = head, *pre = head;
        while (fast && fast->next) {
            pre = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        pre->next = NULL;
        return merge(sortList(head), sortList(slow));
    }
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1);
        ListNode *cur = dummy;
        while (l1 && l2) {
            if (l1->val < l2->val) {
                cur->next = l1;
                l1 = l1->next;
            } else {
                cur->next = l2;
                l2 = l2->next;
            }
            cur = cur->next;
        }
        if (l1) cur->next = l1;
        if (l2) cur->next = l2;
```

```
        return dummy->next;
    }
};
```

Java 解法一：

```
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode slow = head, fast = head, pre = head;
        while (fast != null && fast.next != null) {
            pre = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        pre.next = null;
        return merge(sortList(head), sortList(slow));
    }
    public ListNode merge(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode cur = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                l1 = l1.next;
            } else {
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }
        if (l1 != null) cur.next = l1;
        if (l2 != null) cur.next = l2;
        return dummy.next;
    }
}
```

下面这种方法也是归并排序，而且在merge函数中也使用了递归，这样使代码更加简洁啦～

C++ 解法二：

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *slow = head, *fast = head, *pre = head;
        while (fast && fast->next) {
            pre = slow;
            slow = slow->next;
```

```
            fast = fast->next->next;
        }
        pre->next = NULL;
        return merge(sortList(head), sortList(slow));
    }
    ListNode* merge(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;
        if (l1->val < l2->val) {
            l1->next = merge(l1->next, l2);
            return l1;
        } else {
            l2->next = merge(l1, l2->next);
            return l2;
        }
    }
};
```

Java 解法二：

```
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode slow = head, fast = head, pre = head;
        while (fast != null && fast.next != null) {
            pre = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        pre.next = null;
        return merge(sortList(head), sortList(slow));
    }
    public ListNode merge(ListNode l1, ListNode l2) {
        if (l1 == null) return l2;
        if (l2 == null) return l1;
        if (l1.val < l2.val) {
            l1.next = merge(l1.next, l2);
            return l1;
        } else {
            l2.next = merge(l1, l2.next);
            return l2;
        }
    }
}
```

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.

Example 1:

```
Input: [[1,1],[2,2],[3,3]]
Output: 3
Explanation:
^
|
|        o
|     o
|  o
+------------->
0  1  2  3  4
```

Example 2:

```
Input: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
Output: 4
Explanation:
^
|
|  o
|     o        o
|        o
|  o        o
+------------------->
0  1  2  3  4  5  6
```

这道题给了我们一堆二维点，然后让求最大的共线点的个数，根据初中数学可以知道，两点确定一条直线，而且可以写成 y = ax + b 的形式，所有共线的点都满足这个公式。所以这些给定点两两之间都可以算一个斜率，每个斜率代表一条直线，对每一条直线，带入所有的点看是否共线并计算个数，这是整体的思路。但是还有两点特殊情况需要考虑，一是当两个点重合时，无法确定一条直线，但这也是共线的情况，需要特殊处理。二是斜率不存在的情况，由于两个点 (x1, y1) 和 (x2, y2) 的斜率k表示为 (y2 - y1) / (x2 - x1)，那么当 x1 = x2 时斜率不存在，这种共线情况需要特殊处理。这里需要用到 TreeMap 来记录斜率和共线点个数之间的映射，其中第一种重合点的情况假定其斜率为 INT_MIN，第二种情况假定其斜率为 INT_MAX，这样都可以用 TreeMap 映射了。还需要顶一个变量 duplicate 来记录重合点的个数，最后只需和 TreeMap 中的数字相加即为共线点的总数，但这种方法现在已经无法通过 OJ 了，代码可以参见评论区八楼。

由于通过斜率来判断共线需要用到除法，而用 double 表示的双精度小数在有的系统里不一定准确，为了更加精确无误的计算共线，应当避免除法，从而避免无线不循环小数的出现，那么怎么办呢，这里把除数和被除数都保存下来，不做除法，但是要让这两数分别除以它们的最大公约数，这样例如8和4，4和2，2和1，这三组商相同的数就都会存到一个映射里面，同样也能实现目标，而求 GCD 的函数如果用递归来写那么一行就搞定了，叼不叼，这个方法能很好的避免除法的出现，算是牺牲了空间来保证精度吧，参见代码如下：

C++ 解法一：

```
class Solution {
public:
    int maxPoints(vector<vector<int>>& points) {
        int res = 0;
        for (int i = 0; i < points.size(); ++i) {
            map<pair<int, int>, int> m;
            int duplicate = 1;
            for (int j = i + 1; j < points.size(); ++j) {
                if (points[i][0] == points[j][0] && points[i][1] == points[j][1]) {
                    ++duplicate; continue;
                }
                int dx = points[j][0] - points[i][0];
                int dy = points[j][1] - points[i][1];
                int d = gcd(dx, dy);
                ++m[{dx / d, dy / d}];
            }
            res = max(res, duplicate);
            for (auto it = m.begin(); it != m.end(); ++it) {
                res = max(res, it->second + duplicate);
            }
        }
        return res;
    }
    int gcd(int a, int b) {
        return (b == 0) ? a : gcd(b, a % b);
    }
};
```

Java 解法一：

```
class Solution {
    public int maxPoints(int[][] points) {
        int res = 0;
        for (int i = 0; i < points.length; ++i) {
            Map<Map<Integer, Integer>, Integer> m = new HashMap<>();
            int duplicate = 1;
            for (int j = i + 1; j < points.length; ++j) {
                if (points[i][0] == points[j][0] && points[i][1] == points[j][1]) {
                    ++duplicate; continue;
                }
                int dx = points[j][0] - points[i][0];
                int dy = points[j][1] - points[i][1];
                int d = gcd(dx, dy);
                Map<Integer, Integer> t = new HashMap<>();
                t.put(dx / d, dy / d);
                m.put(t, m.getOrDefault(t, 0) + 1);
```

```
            }
            res = Math.max(res, duplicate);
            for (Map.Entry<Map<Integer, Integer>, Integer> e :
m.entrySet()) {
                res = Math.max(res, e.getValue() + duplicate);
            }
        }
        return res;
    }
    public int gcd(int a, int b) {
        return (b == 0) ? a : gcd(b, a % b);
    }
}
```

令博主惊奇的是，这道题的 OJ 居然容忍 brute force 的方法通过，博主认为下面这种 O(n3) 的解法之所以能通过 OJ，可能还有一个原因就是用了比较高效的判断三点共线的方法。一般来说判断三点共线有三种方法，斜率法，周长法，面积法 (请参见这个帖子)。而其中通过判断叉积为零的面积法是坠好的。比如说有三个点 A(x1, y1)、B(x2, y2)、C(x3, y3)，那么判断三点共线就是判断下面这个等式是否成立：

行列式的求法不用多说吧，不会的话回去翻线性代数，当初少打点刀塔不就好啦~

C++ 解法二：

```
class Solution {
public:
    int maxPoints(vector<vector<int>>& points) {
        int res = 0;
        for (int i = 0; i < points.size(); ++i) {
            int duplicate = 1;
            for (int j = i + 1; j < points.size(); ++j) {
                int cnt = 0;
                long long x1 = points[i][0], y1 = points[i][1];
                long long x2 = points[j][0], y2 = points[j][1];
                if (x1 == x2 && y1 == y2) {++duplicate; continue;}
                for (int k = 0; k < points.size(); ++k) {
                    int x3 = points[k][0], y3 = points[k][1];
                    if (x1 * y2 + x2 * y3 + x3 * y1 - x3 * y2 - x2 * y1 -
x1 * y3 == 0) {
                        ++cnt;
                    }
                }
                res = max(res, cnt);
            }
            res = max(res, duplicate);
        }
        return res;
    }
};
```

Java 解法二：

```java
class Solution {
    public int maxPoints(int[][] points) {
        int res = 0, n = points.length;
        for (int i = 0; i < n; ++i) {
            int duplicate = 1;
            for (int j = i + 1; j < n; ++j) {
                int cnt = 0;
                long x1 = points[i][0], y1 = points[i][1];
                long x2 = points[j][0], y2 = points[j][1];
                if (x1 == x2 && y1 == y2) {++duplicate;continue;}
                for (int k = 0; k < n; ++k) {
                    int x3 = points[k][0], y3 = points[k][1];
                    if (x1*y2 + x2*y3 + x3*y1 - x3*y2 - x2*y1 - x1 * y3 ==
0) {
                        ++cnt;
                    }
                }
                res = Math.max(res, cnt);
            }
            res = Math.max(res, duplicate);
        }
        return res;
    }
}
```