Reverse Words in a String

Given an input string, reverse the string word by word.

For example,

Given s = "`the sky is blue`",

return "`blue is sky the`".

```
public class Solution {
    public String reverseWords(String s) {

        StringBuffer revStr = new StringBuffer();

        if (s == null || s.length() == 0)
        {
            return "";
        }


            String[] str = s.split(" ");
            int n = str.length;

            for (int i = n - 1; i >= 0; i--)
            {
                if (!str[i].equals(""))
                {
                    revStr.append(str[i]).append(" ");
                }

            }

        return revStr.length() == 0 ? "" : revStr.substring(0, revStr.length() - 1);


    }
}
```

Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

Some examples:

```
  ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

```
public class Solution {
    public int evalRPN(String[] tokens) {
        String operator = "+-*/";
        Stack<String> bucket = new Stack<String>();

        for (String p : tokens)
        {
            if (!operator.contains(p))
            {
                bucket.push(p);
            }
            else
            {
                int n1 = Integer.parseInt(bucket.pop());
                int n2 = Integer.parseInt(bucket.pop());
                int out = operation(p, n2, n1);
                bucket.push(Integer.toString(out));
            }
        }
        return Integer.parseInt(bucket.pop());
    }

    private int operation(String operator, int a, int b)
    {
        int c = 0;

        if (operator.equals("+")) c = a + b;
        else if (operator.equals("-")) c = a - b;
        else if (operator.equals("*")) c = a * b;
        else if (operator.equals("/") && b != 0) c = a / b;

        return c;
    }
}
```

Sort List

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

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
        {
            return head;
        }

        ListNode fast = head;
        ListNode slow = head;
        while (fast.next != null && fast.next.next != null)
        {
            fast = fast.next.next;
            slow = slow.next;
        }
        fast = slow.next;
        slow.next = null;
        fast = sortList(fast);
        slow = sortList(head);
        return merge(slow, fast);
    }

    private ListNode merge(ListNode lo, ListNode hi)
    {
        ListNode head = new ListNode(0);
        ListNode cur = head;
        if (lo == null)
        {
            return hi;
        }
        else if (hi == null)
        {
            return lo;
        }
        while (lo != null && hi != null)
        {
            if (lo.val <= hi.val)
            {
                cur.next = lo;
                lo = lo.next;
            }
            else
```

```
            {
                cur.next = hi;
                hi = hi.next;
            }
            cur = cur.next;
        }

        if (lo != null)
        {
            cur.next = lo;
        }
        else if (hi != null)
        {
            cur.next = hi;
        }
        return head.next;

    }
}
```

Reorder List

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

reorder it to: $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \cdots$

You must do this in-place without altering the nodes' values.

For example,

Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

1. break the linked list in the middle into two list;
2. reverse the second linked list
3. merge the two list by inserting the second one after the first one.

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public void reorderList(ListNode head) {
        if (head != null && head.next != null && head.next.next != null)
        {

        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null && fast.next.next != null)
        {
            fast = fast.next.next;
            slow = slow.next;
        }
        ListNode second = slow.next;
        slow.next = null;

        second = reverseOrder(second);
        ListNode curr = head;
        ListNode curr2 = second;
        while (curr2 != null)
        {
            ListNode temp1 = curr.next;
            ListNode temp2 = curr2.next;

            curr.next = curr2;
            curr = temp1;
            curr2.next = curr;

            curr2 = temp2;
        }
        }
    }

    public ListNode reverseOrder(ListNode list)
    {
        if (list == null || list.next == null)
        {
            return list;
        }

        ListNode currNode, nextNode;
        currNode = list;
        nextNode = list.next;
        list.next = null;
        while(nextNode != null)
        {
            ListNode loopNode = nextNode.next;
            nextNode.next = currNode;
            currNode = nextNode;
            nextNode = loopNode;
        }
        list = currNode;

        return list;
    }

}
```

Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

Follow up:

Can you solve it without using extra space?

循环中奇数和偶数个结点；
1. 使用两个指针slow,fast。两个指针都从表头开始走，slow每次走一步，fast每次走两步，如果fast遇到null，则说明没有环，返回false；如果slow==fast，说明有环
2.第一次相遇后，让slow,fast继续走. slow从head开始走，slow，fast各走一步，再次相遇的就是循环起始结点

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head == null || head.next == null)
        {
            return null;
        }

        ListNode slow = head;
        ListNode fast = head;
        while (true)
        {
            slow = slow.next;
            fast = fast.next.next;

            if (fast == null || fast.next == null)
            {
                return null;
            }
```

```
            if (fast == slow)
            {
                break;
            }
        }
        slow = head;
        while (slow != fast)
        {
            slow = slow.next;
            fast = fast.next;
        }
        return slow;
    }
}
```

Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:

Can you solve it without using extra space?

同上

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null)
        {
            return false;
        }
        ListNode slow = head;
        ListNode fast = head;
        while (true)
        {
            if (fast == null || fast.next == null)
            {
                return false;
            }
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast)
            {
                return true;
            }
        }
    }
}
```

Binary Tree Preorder Traversal

Given a binary tree, return the *preorder* traversal of its nodes' values.

For example:

Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[1,2,3]`.

Recursive solution

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<Integer>();
        if (root == null)
        {
            return Collections.EMPTY_LIST;
        }
        list.add(root.val);
        list.addAll(preorderTraversal(root.left));
        list.addAll(preorderTraversal(root.right));

        return list;
    }
}
```

Binary Tree Postorder Traversal

Given a binary tree, return the *postorder* traversal of its nodes' values.

For example:

Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[3,2,1]`.

Note: Recursive solution is trivial, could you do it iteratively?

recursive solution

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
```

```
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<Integer>();
        if (root == null)
        {
            return Collections.EMPTY_LIST;
        }
        list.addAll(postorderTraversal(root.left));
        list.addAll(postorderTraversal(root.right));
        list.add(root.val);
        return list;
    }
}
```

Word Break

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

For example, given

s = `"leetcode"`,

dict = `["leet", "code"]`.

Return true because `"leetcode"` can be segmented as `"leet code"`.

"leetcode": result = TFFFTFFFT

```
public class Solution {
    public boolean wordBreak(String s, Set<String> dict) {
        if (s == null || dict.size() <= 0)
        {
            return false;
        }

        boolean[] result = new boolean[s.length() + 1];
        result[0] = true;
        for (int i = 0; i < s.length(); i++)
        {
            if (result[i] == false)
            {
                continue;
            }
            for (String d : dict)
            {
                int end = i + d.length();
                if (end > s.length())
                {
                    continue;
                }

                if (s.substring(i, end).equals(d))
                {
                    result[end] = true;
                }
            }
        }
        return result[s.length()];
    }
}
```

Word Break II

Given a string *s* and a dictionary of words *dict*, add spaces in *s* to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

s = `"catsanddog"`,

dict = `["cat", "cats", "and", "sand", "dog"]`.

A solution is `["cats and dog", "cat sand dog"]`.

Submission Result: Time Limit Exceeded

Last executed input:"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab", ["a","aa","aaa","aaaa","aaaaa","aaaaaa","aaaaaaa","aaaaaaaa","aaaaaaaaa","aaaaaaaaaa"]

```
public class Solution {
    public List<String> wordBreak(String s, Set<String> dict) {
        List<String> list = new ArrayList<String>();

        if (s == null || dict.size() == 0)
        {
            return list;
        }
        helpBuilder(s, dict, 0, "", list);
        return list;
    }
    private void helpBuilder(String s, Set<String> dict, int start, String item, List<String> list){
        if (start > s.length()){
            list.add(item);
            return;
        }
        StringBuffer currStr = new StringBuffer();

        for (int i = start; i < s.length(); i++)
        {
            currStr.append(s.charAt(i));
            if (dict.contains(currStr.toString())){
                String newItem = item.length() > 0 ? (item + " " + currStr.toString()) : currStr.toString();
                helpBuilder(s, dict, i + 1, newItem, list);
            }
        }
    }

}
```

Single Number

Given an array of integers, every element appears *twice* except for one. Find that single one.

Note:

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

```
int cnt = 0;
int x = cnt++;
cnt = 0;
int y = ++cnt;
System.out.println(x + " " + y); //0 1//

     int cnt = 0;
```

```
            int x = cnt++;
            int y = ++cnt;
            System.out.println(x + " " + y); // 0 2//
//cnt++ increments cnt by 1, but return the old value of cnt; ++cnt increments cnt by 1 and return the new value;//


public class Solution {
    public int singleNumber(int[] A) {
        if (A.length <= 1 ) {
            return A[0];
        }

        int len = A.length;
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

        for (int i = 0; i < len; i++) {
            if (!map.containsKey(A[i])) {
                int count = 0;
                map.put(A[i], ++count);
            }
            else {
                map.put(A[i], map.get(A[i]) + 1);
            }
        }

        int results = 0;
        for (Integer j : map.keySet()) {
            if (map.get(j) == 1) {
                results = j;
            }
        }

        return results;
    }
}
```

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?

通常是要求的变量跟左右元素有关系的题目: 两边扫描的方法

一些例子:

1 2 3 3 3 =》 8 因为candy数可以是

1 2 3 1 1

1 2 3 2 3 =》 9 因为candy数可以是

1 2 3 1 2

思路:

1

```
    d[i] 是给第i个小孩最少几块糖
        rank[i] > rank[i - 1]，必须比前一个多给一块，d[i] = d[i - 1] + 1
        rank[i] <= rank[i - 1]，两个排名一样，第二个就给一块就行了，d[i] = 1
```

基本思路就是进行两次扫描，一次从左往右，一次从右往左。第一次扫描的时候维护对于每一个小孩左边所需要最少的糖果数量，存入数组对应元素中，第二次扫描的时候维护右边所需的最少糖果数，并且比较将左边和右边大的糖果数量存入结果数组对应元素中。这样两遍扫描之后就可以得到每一个所需要的最最少糖果量，从而累加得出结果。方法只需要两次扫描，所以时间复杂度是O(2*n)=O(n)。空间上需要一个长度为n的数组，复杂度是O(n)。

```
public class Solution {
    public int candy(int[] ratings) {
        if (ratings.length == 0) {
            return 0;
        }

        int[] num = new int[ratings.length];
        num[0] = 1;

        for (int i = 1; i < ratings.length; i++) {
            if (ratings[i] > ratings[i - 1]) {
                num[i] = num[i - 1] + 1;
            }
            else if (ratings[i] == ratings[i - 1]) {
                num[i] = 1;
            }
            else if (ratings[i] < ratings[i - 1]) {
                num[i] = 1;
            }
        }

        int res = num[ratings.length - 1];
        for (int i = ratings.length - 2; i >= 0; i--) {
            int curr = 1;
            if (ratings[i] > ratings[i + 1]) {
                curr = num[i + 1] + 1;
            }
            res += Math.max(num[i], curr);
            num[i] = curr;
        }

        return res;
    }
}
```

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.

```
public class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        if (gas.length == 0 || cost.length == 0 || gas.length != cost.length) {
            return -1;
        }

        int sum = 0;
        int total = 0;
        int start = 0;

        for (int i = 0; i < gas.length; i++) {
            sum += gas[i] - cost[i];
            total += sum;

            if (sum < 0) {
```

```
            start = i + 1;
            sum = 0;
        }
      }

      if (total < 0) {
        return -1;
      }

      return start;
    }
}
```

Palindrome Partitioning

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of *s*.

For example, given *s* = `"aab"`,

Return

```
  [
    ["aa","b"],
    ["a","a","b"]
  ]
recursive solution, two pointers; a start pointer that increases recursively, when the end index pointer is fixed(a,ab -> a,a,b);
an end index pointer increases iteratively, when the start pointer is fixed (a->aa).
dfs

public class Solution {
    public List<List<String>> partition(String s) {
        int n = s.length();
        List tol = new ArrayList();
        List<String> head = new ArrayList<String>();
        if (s == null || n == 0) {
            return tol;
        }

        getPal(s, tol, head, 0);
        return tol;
    }

    private void getPal(String s, List tol, List<String> head, int start) {
        int n = s.length();
        if (n == start) {
                tol.add(new ArrayList<String>(head));
         } else {

          for (int i = start; i < n; i++) {
            String sub = s.substring(start, i + 1);
            String reverseSub = new StringBuffer(sub).reverse().toString();

            if (sub.equals(reverseSub) && !sub.equals("")) {
                head.add(sub);
                getPal(s, tol, head, i + 1);
                head.remove(head.size() - 1);
            }

        }
         }
    }
}
```

Palindrome Partitioning II

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of *s*.

For example, given *s* = `"aab"`,

Return `1` since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

Submission Result: Time Limit Exceeded

```
  Last executed input:      "abababababababababababcbabababababababababababa"
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space


public class Solution {
    public int minCut(String s) {
        int n = s.length();
        if (s == null || n == 0) {
            return 0;
        }

        List tol = new ArrayList();
        List<String> head = new ArrayList<String>();
        getPal(s, tol, head, 0);

        int size = Integer.MAX_VALUE;
        Iterator<List> it = tol.iterator();
        while (it.hasNext()) {
            List root = it.next();
            int temp = root.size();
            if (temp < size) {
                size = temp;
            }
        }
        return size;
    }

    private void getPal(String s, List tol, List<String> head, int start) {
        int n = s.length();
        if (n == start) {
                tol.add(new ArrayList<String>(head));
         } else {

          for (int i = start; i < n; i++) {
            String sub = s.substring(start, i + 1);
            String reverseSub = new StringBuffer(sub).reverse().toString();

            if (sub.equals(reverseSub) && !sub.equals("")) {
                head.add(sub);
                getPal(s, tol, head, i + 1);
                head.remove(head.size() - 1);
            }

        }
         }
    }
}
```

Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

```
Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

DFS: add(ad), add(ae), add(af), add(bd)…..

```
public class Solution {
    public List<String> letterCombinations(String digits) {

        String[] map = new String[10];
        map[0] = "";
        map[1] = "";
        map[2] = "abc";
        map[3] = "def";
        map[4] = "ghi";
        map[5] = "jkl";
        map[6] = "mno";
        map[7] = "pqrs";
        map[8] = "tuv";
        map[9] = "wxyz";

        int n = digits.length();
        List<String> comb = new ArrayList<String>();
        char[] trace = new char[n];

        if (digits == null || n == 0) {
            comb.add(new String(trace));
            return comb;
        }

        LC(digits, 0, 0, comb, trace, map);
        return comb;

    }

    private void LC(String digits, int l, int d, List<String> comb, char[] trace, String[] map) {
        int n = digits.length();
        if (l == n) {
            comb.add(new String(trace));
        } else {
            int ind = Integer.parseInt(digits.substring(l, l + 1));
            if (ind > 1) {
                String A = map[ind];
                for (int j = 0; j < A.length(); j++) {
                    trace[d] = A.charAt(j);
                    LC(digits, l + 1, d + 1, comb, trace, map);
                }
            } else {
                LC(digits, l + 1, d, comb, trace, map);
            }

        }
    }
}
```