4.带环链表问题（快慢指针）
Hint: 通过判断快慢指针遍历是否相遇
```
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: True if it has a cycle, or false
     */
    public boolean hasCycle(ListNode head) {
        // write your code here
        if(head == null || head.next == null){
            return false;
        }
        ListNode slow = head;
        ListNode fast = head.next;
        while(fast.next != null && fast.next.next != null){
            if(slow == fast){
                return true;
            }
            fast = fast.next.next;
            slow = slow.next;
        }
        return false;
    }
}
```
Follow up: 1. 求环入口
Hint：两指针相遇后其中一个从head走起，每次走一步，若head = slow.next/fast.next返回head
2. 求两个链表交点
Hint：HashMap/将linkedlist头尾相接变成环，求环的入口

5.两大经典排序算法

Quick Sort（Divide and Conquer)
整体有序 -> 局部有序

Key： Partition时实现均分，将<=和>=分布在template的两边，否则在特殊情况中会导致时间复杂度退化至O(n^2)

Code:
```
public void sortIntegers2(int[] A) {
        // write your code here
        if(A == null || A.length == 0){
            return;
        }
        quickSort(A, 0, A.length - 1);
    }
    private void quickSort(int[] A, int start, int end){
        if(start >= end){
            return;
        }
        int left = start;
        int right = end;
        int mid = (left + right) / 2;
        int pivot = A[mid];
//判断条件要为left <= right, 否则造成stack overflow，例如（1，2）排序
        while(left <= right){
            while(left <= right && A[left] < pivot){
//为了实现两可能的均分
                left++;
            }
            while(left <= right && A[right] > pivot){
                right--;
            }
            if(left <= right){
                int temp = A[left];
                A[left] = A[right];
                A[right] = temp;
                left++;
                right--;
            }
        }
        quickSort(A, start, right);
        quickSort(A, left, end);
    }
```
Quick Sort 三大要点：
1.pivot， start， end
2.left <= right not left < right
3.A[left] < pivot not A[left] <= pivot

Merge Sort
局部有序 -> 整体有序
由于需要开辟额外空间，实际使用中稳定性不如Quick Sort

```
public void sortIntegers2(int[] A) {
        // write your code here
        if(A == null || A.length == 0){
            return;
        }

        int[] temp = new int[A.length];
        mergeSort(A, 0, A.length - 1, temp);
    }

    private void mergeSort(int[] A, int start, int end, int[] temp){
        if(start >= end){
            return;
        }
        mergeSort(A, start, (start + end) / 2, temp);
        mergeSort(A, (start + end) / 2 + 1, end, temp);
        merge(A, start, end, temp);
    }

    private void merge(int[] A, int start, int end, int[] temp){
        int mid = (start + end) / 2;
        int leftindex = start;
        int rightindex = mid + 1;
        int index = leftindex;
        while(leftindex <= mid && rightindex <= end){
            if(A[leftindex] < A[rightindex]){
                temp[index++] = A[leftindex++];
            } else{
                temp[index++] = A[rightindex++];
            }
        }
        while(leftindex <= mid){
            temp[index++] = A[leftindex++];
        }
        while(rightindex <= end){
            temp[index++] = A[rightindex++];
        }
        for(int i = start; i <= end; i++){
            A[i] = temp[i];
        }
    }
```

Quick Sort 与 Merge Sort 比较

	时间复杂度	空间复杂度	排序顺序	稳定性	
Quick Sort	平均O(nlgn) 最坏O(n^2)	O(1)原地排序	整体有序->局部有序	不稳定排序	
Merge Sort	平均O(nlgn)（最好最坏一样）	O(n)	局部有序->整体有序	稳定排序	

Quick Select
Hint:与Quick Sort类似，但是partition之后根据情况再进行partition，而Quick Sort则是partition以后无条件对两边进行partition递归下去

Practice：Kth Largest Element

```
public class Solution {
    /**
     * @param n: An integer
     * @param nums: An array
     * @return: the Kth largest element
     */
    public int kthLargestElement(int n, int[] nums) {
        // write your code here
        if(nums == null || nums.length == 0){
            return -1;
        }
        return quickSelect(nums, 0, nums.length - 1, n);
    }

    private int quickSelect(int[] nums, int start, int end, int n){
        if(start == end){
            return nums[start];
        }
        int left = start;
        int right = end;
        int pivot = nums[(left + right) / 2];
        while(left <= right){
            while(left <= right && nums[left] > pivot){
                left++;
            }
            while(left <= right && nums[right] < pivot){
                right--;
            }
            if(left <= right){
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
                left++;
                right--;
            }
        }
        if(start + n - 1 <= right ){
            return quickSelect(nums, start, right, n);
        }
        if(start + n - 1 >= left ){
            return quickSelect(nums, left, end, n - (left - start));
        }
        return nums[right + 1];
    }
}
```