常见算法题

算法题：

1. 复杂链表的复制
2. 链表倒数第k个节点
3. 链表中间节点
4. 链表反转
5. 链表两两反转
6. K个一组反转链表。时间复杂度，空间复杂度分析一下
7. int数组，实现偶数在前，奇数在后，不改变相对顺序
8. 二分法查找
9. 二分法找排序数组中绝对值最小的元素
10. 双重校验单例模式
11. 找出数组中第K大的数
12. 链表判断是否存在环
13. 按照二叉树先序、中序、后序打印所有的节点
14. 斐波拉切数列
15. 无序数组中位数
16. String替换字符串
17. 找出数组中唯一的出现奇数次的数
18. 判断二叉树是否为镜像二叉树
19. 合并两个有序链表

场景题：

1. 给定 a、b 两个文件，各存放 50 亿个 URL，每个 URL 各占 64B，内存限制是 4G。请找出 a、b 两个文件...
2. 64匹马，8个赛道，找最快的4匹马
3. 亿级别query其中有重复，如何找到重复率top(100)，单机场景
4. ip黑名单

算法题：

1. 复杂链表的复制

解法1：（时间复杂度是O(n^2)，不推荐）

第一步是复制原始链表上的每一个结点，并用Next节点链接起来；

第二步是设置每个结点的Sibling节点指针。

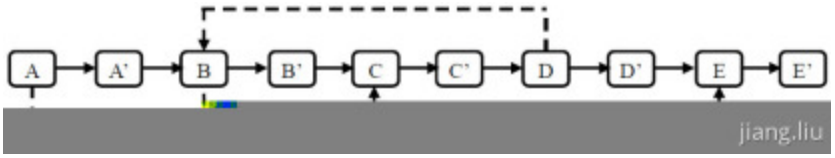
解法2：（使用空间换时间，以O(n)的空间消耗把时间复杂度由O(n^2)降低到O(n)）

第一步是复制原始链表上的每个结点N创建N'，然后把这些创建出来的结点用Next链接起来。同时我们把<N,N'>的配对信息放到一个哈希表中。

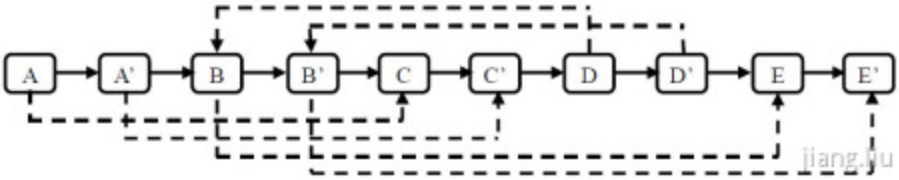
第二步还是设置复制链表上每个结点的m_pSibling。由于有了哈希表，我们可以用O(1)的时间根据S找到S' 。

解法3：（时间复杂度O(n)）

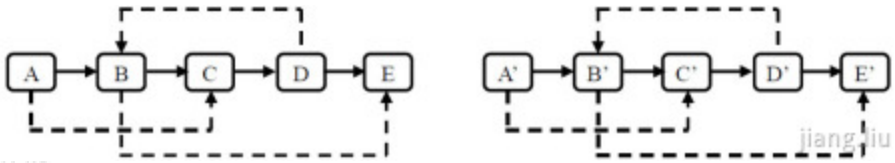
第一步仍然是根据原始链表的每个结点N创建对应的N'。（把N'链接在N的后面）。



第二步设置复制出来的结点的Sibling。（把N'的Sibling指向N的Sibling）。



第三步把这个长链表拆分成两个链表：把奇数位置的结点用Next链接起来就是原始链表，偶数数值的则是复制链表。



2. 链表倒数第k个节点

定义两个指针，第一个指针先走k-1步，第二个指针在走，当第一个指针走完，第二个指针的位置即为倒数第K个节点。

```java
public ListNode FindKthToTail (ListNode pHead, int k) {
        if(pHead == null){
            return pHead;
        }
        ListNode fast = pHead;
        while(k > 0){
           fast = fast.next;
           k--;
        }
        while(fast != null){
            fast = fast.next;
            pHead = pHead.next;
        }
        return pHead;
    }
```

3. 链表中间节点

可以定义两个指针，同时从链表的头出发，一个走一步，一个走两步，走的快的到表尾时，走得慢的正好是中间结点。

4. 链表反转

方法1：将单链表储存为数组，然后按照数组的索引逆序进行反转，效率不高需要遍历两次。

方法2：遍历原链表，用头插法重新生成一个新的链表。缺点是需要额外空间存储新链表。

方法3：**三个指针就地反转：**使用三个指针指向：当前节点A，下个节点B，以及下下个节点C。遍历时，首先记录下下个节点C，然后节点B的指针断开并指向A。然后移动进入下一组。

A -> B -> C ->D -> E

A <- B 、 C ->D -> E

整个过程只需遍历链表一次，效率提高不少，且需要的外部空间也较第一种方法要少很多。

方法4：递归实现反转：

```java
public ListNode ReverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode last = ReverseList(head.next);
        head.next.next = head;
        head.next = null;
        return last;
}

public ListNode reverseList(ListNode head) {
        ListNode temp = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = temp;
            temp = curr;
            curr = next;
        }
        return temp;
    }
```

5. 链表两两反转

我们需要四个指针： current，next， nextNext、prev（*上一组的尾指针，在下一组反转后需要改变*）。

分析：我们需要两个“指针”指着当前要反转的两个值current和next。两两反转后，我们还需要记录下一个的值，即反转A和B后， 需要记录 C 值，我们才能够不断向下走，直到到达链表末端。所以，需要另一个指向下一个值的“指针”，即nextNext。反转以后，A的下一个是C， 但是，实际上，A的下一个应该是D，所以，每次反转时，我们需要更新前一个值的下一个值，也就是说把 A -> C 改成 A -> D，所以需要prev指针。所以，要完成这个操作，我们总共需要4个“指针”。

6. K个一组反转链表。时间复杂度，空间复杂度分析一下

类似【4.链表反转】，只不过需要先确定k个长度的子链表，以及连接每次反转之后的小链表的next.

7. int数组，实现偶数在前，奇数在后，不改变相对顺序

采用两个指针，一个指向数组的开头，一个指向数组的结尾。当开头的指针遇到奇数，跳过，遇到偶数，停止，指向数组末尾的指针遇到偶数，跳过，遇到奇数，停止。然后两个指针交换元素。这种思

想类似于快速排序中的partition思想。算法原地调整，时间复杂度O(n),空间复杂度O(1).

8. 二分法查找

Java 复制代码

```java
public static int binarySearch(int[] num, int key){
        int left = 0;
        int right = num.length-1;
        while(left <= right){
            int mid = (right+left)/2;
            if(num[mid] < key){
                left = mid + 1;
            }else if(num[mid] > key){
                right  = mid - 1;
            }else{
                while(mid != 0 && num[mid] == num[mid-1]){
                    mid--;
                }
                return mid;
            }
        }
        return -1;
    }
```

9. 二分法找排序数组中绝对值最小的元素

分三种情况：

（1）如果全是正数，返回第一个元素值。if(A[0] >= 0)

（2）如果全是负数，返回最后一个元素值。if(A[n-1] <= 0)

（3）有正有负，利用二分查找找到0的插入位置，如果正好找到0的位置，0就是绝对值最小的元素, 如果没有找到0，插入位置的左右元素比较绝对值大小 ，返回较小者OK。

10. 双重校验单例模式

```java
/**
 * 双检锁机制，基于懒汉式
 */
public class DoubleCheck {

    private volatile static  DoubleCheck doubleCheck;
    private DoubleCheck(){}
    public static DoubleCheck getInstance(){
        if(doubleCheck == null){
            synchronized (DoubleCheck.class){
                if(doubleCheck == null){
                    doubleCheck = new DoubleCheck();
                }
            }
        }
        return doubleCheck;
    }
}
```

11. 找出数组中第K大的数

方法一：

利用冒泡排序，进行k趟排序，即可查找到第k大的数（每趟冒泡排序都寻找当前序列中的最大值）

方法二：

利用选择排序，进行k趟排序，即可查找到第k大的数（每趟选择排序都寻找当前序列中的最大值）

方法三：

利用快速排序的思想，只需找到第k大的数，不必把所有的数排好序。

思路分析：先任取一个数（找到基准值），把比它大的数移动到它的右边，比它小的数移动到它的左边。移动完成一轮后，看该数的下标（从0计数），如果刚好为length−k，则它就是第k大的数（因为移动后的基准值左边都是比它小的数，右边都是比他大的数，右边若没有元素，则说明当前基准值是第1大元素，若右边有1个元素，则说明当前基准值是第2大元素，若右边有2个元素，则说明当前基准值是第3大元素.....即当前基准值右边有k−1个元素，或当前基准值的下标为length−k，则说明当前基准值是第k大元素），如果小于length−k，说明第k大的数在它右边，如果大于length−k，则说明第k大的数在它左边，取左边或者右边继续进行移动，直到找到。

上述方法对应的数据量比较小，如果N很大，100亿？利用最小堆来实现。

12. 链表判断是否存在环

采用“快慢指针”，或者set集合判断是否重复。

Java 复制代码

```java
//双指针
public boolean hasCycle(ListNode head) {
        if(head == null) return false;

      ListNode k = head;
        ListNode m = head;
        while(k != null && k.next != null){
            k = k.next.next;
            m = m.next;
            if(k == m)  return true;
        }
        return false;
}

//利用set方式
public boolean hasCycle(ListNode head) {
        Set<ListNode> set =new HashSet<>();
        while(head != null){
            if(set.contains(head)){
                return true;
            }
            set.add(head);
            head = head.next;
        }
        return false;
}
```

13. 按照二叉树先序、中序、后序打印所有的节点

Java 复制代码

```java
public int[][] threeOrders (TreeNode root) {
    // write code here
    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    List<Integer> list3 = new ArrayList<>();
    pre(root,list1);
    in(root,list2);
    post(root,list3);

    int[][] res = new int[3][list1.size()];
    for(int i = 0; i< list1.size(); i++){
        res[0][i] = list1.get(i);
        res[1][i] = list2.get(i);
        res[2][i] = list3.get(i);
    }
    return res;
}

public void pre(TreeNode root, List<Integer> list){
    if(root == null){
        return;
    }
    list.add(root.val);
    pre(root.left,list);
    pre(root.right,list);
}

 public void in(TreeNode root, List<Integer> list){
    if(root == null){
        return;
    }
    in(root.left,list);
    list.add(root.val);
    in(root.right,list);
}

 public void post(TreeNode root, List<Integer> list){
    if(root == null){
        return;
    }
    post(root.left,list);
    post(root.right,list);
    list.add(root.val);
}
```

14. 斐波拉切数列

```java
public int fib(int n) {
        int a = 0, b = 1;
        if(n < 2){
            return n;
        }
        else{
            for(int i = 2; i <=n; i++){
                int sum = a + b;
                a = b;
                b = sum;
            }
            return b;
        }
}
```

15. 无序数组中位数

思路1：首先将数组排序，然后直接从排序数组中找出中位数。这个算法的复杂度是O（nlogn），就是排序的复杂度。

思路2：类似于快速排序，采用的是分而治之的思想。基本思路是：任意挑一个元素，以该元素为支点，将数组分成两部分，左部分是小于等于支点的，右部分是大于支点的。如果你的运气爆棚，左部分正好是（n－1）／2个元素，那么支点的那个数就是中位数。否则，先通过数组长度算出中位数所在位置，然后判断支点位置与中位数的位置，将中位数所在位置的区间重新进行快排。

16. String替换字符串

Java 复制代码

```java
    public static String replace(String str, String target, String replacement) {
        // 正常这里需要对str，target，replacement做输入校验，这里我省略，
        // 比如str比target端的时候可以直接返回空字符串
        StringBuilder res = new StringBuilder();
        for (int i = 0; i < str.length(); ) {
            if (isMatch(str, i, target)) {
                i += target.length();  // 如果匹配，需要直接向前跳target.length
                res.append(replacement);
                continue;
            }
            res.append(str.charAt(i++));
        }
        return res.toString();
    }

    // 单纯确认从str的pos位置开始，是否和target相匹配
    private static boolean isMatch(String str, int pos, String target) {
        for (int i = 0; i < target.length() && i + pos < str.length(); i++) {
            if (str.charAt(i + pos) != target.charAt(i)) {
                return false;
            }
        }
        return true;
    }
```

17.找出数组中唯一的出现奇数次的数

Java 复制代码

```java
import java.util.*;
public class FindOdd {
    public static void main(String[] args) {
        int [] arr = {1, 4, 7, 3, 1, 3, 7, 4, 6, 9, 6};

        // 方法一：通过hashMap计数；
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
        for(int i=0; i<arr.length; i++) {
            if(map.containsKey(arr[i])) {
                map.remove(arr[i]);//出现了偶数次则删掉
            } else {
                map.put(arr[i], 1);//还未出现过则次数设为1
            }
        }
        Iterator iterator = map.keySet().iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 方法二：异或运算 n^0=n; n^n=0;
        int a = 0;
        for(int i: arr){
            a = a ^ i;
        }
        System.out.println(a);
    }
}
```

18. 判断二叉树是否为镜像二叉树

Java 复制代码

```java
private static boolean isSymmeric(TreeNode root) {
        if (root != null)
            return compTree(root.left, root.right);
        else
            return true;
}

//比较两棵树是否镜像对称
private static boolean compTree(TreeNode p, TreeNode q){
        if(p == null && q == null)
            return true;
        //包含具体值判断（&& p.val == q.val），不判断值不加即可
        if (p != null && q != null && p.val == q.val)
            return compTree(p.left, q.right) && compTree(p.right, q.left);
        else
        return false;
}

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
}
```

19. 合并两个有序链表

```java
public ListNode mergeTwoLists2(ListNode l1, ListNode l2) {
        //定义一个节点数据域是 -1 的节点
        ListNode prehead = new ListNode(-1);
        //p 指针指向 -1 节点
        ListNode p = prehead;

        while (l1 != null && l2 != null) {
            //l1 和 l2 谁小，p.nexrt 就等于谁
            if (l1.val <= l2.val) {
                //prehead 连接 l1
                p.next = l1;
                //继续判断 l1 的下一个节点
                l1 = l1.next;
            } else {
                //否则 prehead 连接 l2
                p.next = l2;
                //继续判断 l2 的下一个节点
                l2 = l2.next;
            }
            p = p.next;
        }
        p.next = l1 == null ? l2 : l1;
        return prehead.next;
}
```

场景题：

1. 给定 a、b 两个文件，各存放 50 亿个 URL，每个 URL 各占 64B，内存限制是 4G。请找出 a、b 两个文件共同的 URL。

每个 URL 占 64B，那么 50 亿个 URL 占用的空间大小约为 320GB,

> 5, 000, 000, 000 _ 64B ≈ 5GB _ 64 = 320GB

由于内存大小只有 4G，因此，我们不可能一次性把所有 URL 加载到内存中处理。对于这种类型的题目，一般采用分治策略 ，即：把一个文件中的 URL 按照某个特征划分为多个小文件，使得每个小文件大小不超过 4G，这样就可以把这个小文件读到内存中进行处理了。

首先遍历文件 a，对遍历到的 URL 求 hash(URL) % 1000 ，根据计算结果把遍历到的 URL 存储到 a0, a1, a2, ..., a999，这样每个大小约为 300MB。使用同样的方法遍历文件 b，把文件 b 中的 URL 分别存储到文件 b0, b1, b2, ..., b999 中。这样处理过后，所有可能相同的 URL 都在对应的小文件中，即 a0 对应 b0, ..., a999 对应 b999，不对应的小文件不可能有相同的 URL。那么接下来，我们只需要求出这 1000 对小文件中相同的 URL 就好了。

接着遍历 ai(i∈[0,999])，把 URL 存储到一个 HashSet 集合中。然后遍历 bi 中每个 URL，看在 HashSet 集合中是否存在，若存在，说明这就是共同的 URL，可以把这个 URL 保存到一个单独的文件中。

> hash（url）%1000，已经能够保证相同的url一定在相同的文件下标的小文件中了。

2. 64匹马，8个赛道，找最快的4匹马

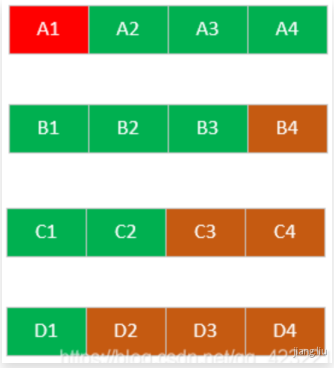
如果能计时，果断8场。

如果不能计时：

首先将马分成八组，赛八场，每场后四名直接淘汰了。

然后将八组中，每组第一进行比赛，淘汰后四名所在的整组。

这个时候剩下四组从快到慢排一下，每组4匹马，如下图：



其中A1是最快的毋庸置疑，考虑到速度 A1 > B1 > C1 > D1的原因，D2 、D3、D4、C3、C4、B4无缘前四名。

下面要从绿色的方块找到前三名。可是绿色的方块有9个，一般至少比较两次。总共11次。

不过也有可能比较1次，比如说除了D1，其他八匹马比赛，结果C1没有进入前三。那么D1就必定进不了前4.

top4已经出来了。

3. 亿级别query其中有重复，如何找到重复率top(100)，单机场景

4. ip黑名单