```
        if (b == 0) {
                throw new RuntimeException("divisor is 0");
        }
        if (a == Integer.MIN_VALUE && b == Integer.MIN_VALUE) {
                return 1;
        } else if (b == Integer.MIN_VALUE) {
                return 0;
        } else if (a == Integer.MIN_VALUE) {
                int res = div(add(a, 1), b);
                return add(res, div(minus(a, multi(res, b)), b));
        } else {
                return div(a, b);
        }
}
```

整数的二进制表达中有多少个 1

【题目】

给定一个 32 位整数 *n*，可为 0，可为正，也可为负，返回该整数二进制表达中 1 的个数。

【难度】

尉 ★★☆☆

【解答】

最简单的解法。整数 *n* 每次进行无符号右移一位，检查最右边的 bit 是否为 1 来进行统计。具体请参看如下代码中的 count1 方法。

```
public int count1(int n) {
        int res = 0;
        while (n != 0) {
                res += n & 1;
                n >>>= 1;
        }
        return res;
}
```

如上方法在最复杂的情况下要经过 32 次循环，下面看一个循环次数只与 1 的个数有关的解法，如下代码中的 count2 方法。

```
public int count2(int n) {
```

```
        int res = 0;
        while (n != 0) {
                n &= (n - 1);
                res++;
        }
        return res;
}
```

每次进行 n&=(n-1)操作，接下来在 while 循环中就可以忽略掉 bit 位上为 0 的部分。

例如，n=01000100，n-1=01000011，n&(n-1)=01000000，说明处理到 01000100 之后，下一步还得处理，因为 01000000!=0。n=01000000，n-1=00111111，n&(n-1)=00000000，说明处理到 01000000 之后，下一步就不用处理，因为接下来没有 1。所以，n&=(n-1)操作的实质是抹掉最右边的 1。

与 count2 方法复杂度一样的是如下代码中的 count3 方法。

```
public int count3(int n) {
        int res = 0;
        while (n != 0) {
                n -= n & (~n + 1);
                res++;
        }
        return res;
}
```

每次进行 n-=n&(~n+1)操作时，这也是移除最右侧的 1 的过程。等号右边 n & (~n + 1)的含义是得到 n 中最右侧的 1，这个操作在位运算的题目中经常出现。例如，n=01000100，n&(~n+1)=00000100 ， n-(n&(~n+1))=01000000 。 n=01000000 ， n&(~n+1)=01000000 ，n-(n&(~n+1)) = 00000000。接下来不用处理了，因为没有 1。

接下来介绍一种看上去很“超自然”的方法，叫作平行算法，参看如下代码中的 count4 方法。

```
public int count4(int n) {
        n = (n & 0x55555555) + ((n >>> 1) & 0x55555555);
        n = (n & 0x33333333) + ((n >>> 2) & 0x33333333);
        n = (n & 0x0f0f0f0f) + ((n >>> 4) & 0x0f0f0f0f);
        n = (n & 0x00ff00ff) + ((n >>> 8) & 0x00ff00ff);
        n = (n & 0x0000ffff) + ((n >>> 16) & 0x0000ffff);
        return n;
}
```

下面解释一下这个过程。

0x55555555 即 01010101010101010101010101010101。(n & 0x55555555) + ((n>>>1) &

0x55555555)的结果描述了每两个 bit 成一组 1 的数量分布。以 n=-1(11111111111111111111111111111111)为例进行说明，n=(n & 0x55555555) + ((n >>> 1) & 0x55555555)为10101010101010101010101010101010，可以看到每两个 bit 成一组 1 的数量状况为 10，也就是每组 2 个。

接下来，0x33333333 即 00110011001100110011001100110011，所以(n & 0x33333333) + ((n >>> 1) & 0x33333333)就描述了 4 个 bit 成一组 1 的数量分布。此时 n=(n & 0x33333333) + ((n >>> 1) & 0x33333333)为01000100010001000100010001000100，它就代表 4 个 bit 位成一组的 1 数量状况为 0100，也就是每组 4 个。

接下来 n 依次为00001000000010000000100000001000，代表 8 个 bit 位成一组 1 的数量状况为 00001000，也就是每组 8 个。00000000000100000000000000010000 代表 16 个 bit 成一组 1 的数量状况为 0000000000010000，也就是每组 16 个。00000000000000000000000000100000 代表 32 个 bit 成一组 1 的数量状况为 00000000000000000000000000100000，也就是每组 32 个。

类似并归的过程，组与组之间的数量合并成一个大组，进行下一步的并归。

除此之外，还有很多极为逆天的算法可以解决这个问题，比如 MIT hackmem 算法等。有兴趣的读者可以去网上查找，但对面试来说，那些方法实在太偏、难、怪，所以本书不再介绍。

在其他数都出现偶数次的数组中找到出现奇数次的数

【题目】

给定一个整型数组 arr，其中只有一个数出现了奇数次，其他的数都出现了偶数次，打印这个数。

【进阶】

有两个数出现了奇数次，其他的数都出现了偶数次，打印这两个数。

【要求】

时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。