简介
1.基础知识
1.1 数字在计算机中的表示
1.2 位运算规则
1.2.1 与、或、异或和取反
1.2.2 移位运算
1.2.3 移位运算与乘除法的关系
1.2.4 位运算常用技巧
2.位运算高频算法题
2.1 位移的妙用
2.1.1 位1的个数
2.1.2 比特位计数
2.1.3 颠倒无符号整数
2.2 溢出问题
2.2.1 整数反转
2.2.2 字符串转整数
2.2.3 回文数
2.3.进制专题
2.3.1 七进制数
2.3.2 进制转换
2.4. 数组实现加法
2.5 位实现运算
2.5.1 位运算实现加法
2.5.2 递归乘法
2.6 幂运算
2.6.1 求2的幂
2.6.2 求3的幂
2.6.3 求4的幂
2.7 只出现一次的数字
3.总结

简介

从本章开始介绍三个不是数据结构的结构：位运算、数学和字符串。这三者在算法中出现的频率特别高，而且三者密切相关，放在一起学习更高效。

位运算是计算机的核心基础，数据的表示和计算几乎都少不了，在JVM以及很多高性能代码里大量使用，甚至很多算法本身就是基于位进行的。在算法方面，很多位相关的算法有很多技巧，不学真不知道。另外很多算法虽然看起来与位运算无关， 但是用位操作优化一下，性能会提升很多，所以位运算的问题值得好好学习。

数学是学生时代掉头发的学科，算法是毕业后掉头发的学科。而两者又是相通的，很多算法本来就是数学问题，而很多数学问题也需要借助算法才能用代码实现。数学涉及的范围很广，本章还是选择最热门最重要的问题来讲解，主要包括：位运算问题、进制问题、数学计算（加减乘除拓展问题、指数对数幂和阶乘）问题、日期问题、初等数论等等。

字符串也是非常特殊的结构，常见的算法题从数组到高难度的动态规划全都有。本章内容比较多，为此将位和数学放一个讲义，字符串单独一个讲义。但是整体难度不是很大，各位加油！

1.基础知识

在学习位操作之前，我们先明确数据在计算机中怎么表示的。我们明确原码、反码和补码的概念和表示方法，之后介绍位运算相关的问题。

1.1 数字在计算机中的表示

机器数 一个数在计算机中的二进制表示形式，叫做这个数的机器数。机器数是带符号的，在计算机用一个数的最高位存放符号，正数为0，负数为1。比如，十进制中的数 +3 ，计算机字长为8位，转换成二进制就是00000011。如果是 -3 ，就是 10000011 。这里的 00000011 和 10000011 就是机器数。

真值 因为机器数第一位是符号位，所以机器数的形式值就不等于真正的数值。例如上面的有符号数 10000011，其最高位1代表负，其真正数值是 -3 而不是形式值131（10000011转换成十进制等于131）。所以，为了好区别，将带符号位的机器数对应的真正数值称为机器数的真值。例：0000 0001的真值 = +000 0001 = +1，1000 0001的真值 = –000 0001 = –1。

计算机对机器数的表示进一步细化：原码， 反码， 补码。

原码 就是符号位加上真值的绝对值，即用第一位表示符号，其余位表示值， 比如如果是8位二进制:

```
[+1]原 = 0000 0001
[–1]原 = 1000 0001
```

第一位是符号位，因为第一位是符号位，所以8位二进制数的取值范围就是:

[1111 1111 , 0111 1111]，也即 [-127 , 127]

反码 的表示方法是：正数的反码是其本身，而负数的反码是在其原码的基础上，符号位不变，其余各个位取反。例如：

```
[+1] = [00000001]原 = [00000001]反
[–1] = [10000001]原 = [11111110]反
```

可见如果一个反码表示的是负数，人脑无法直观的看出来它的数值，通常要将其转换成原码再计算。

在应用中，因为**补码** 能保持加和减运算的统一，因此应用更广，其表示方法是:

- 正数的补码就是其本身；
- 负数的补码是在其原码的基础上，符号位不变，其余各位取反，最后+1(即在反码的基础上+1)。

```
[+1] = [00000001]原 = [00000001]反 = [00000001]补
[–1] = [10000001]原 = [11111110]反 = [11111111]补
```

对于负数， 补码表示方式也是人脑无法直观看出其数值的，通常也需要转换成原码在计算其数值。

拓展 为何会有原码，反码和补码

既然原码就能表示数据，那为什么实际软件中更多使用的是补码呢？接下来我们就看一看。

现在我们知道了计算机可以有三种编码方式表示一个数，对于正数因为三种编码方式的结果都相同:

> [+1] = [00000001]原 = [00000001]反 = [00000001]补

但是对于负数:

> [-1] = [10000001]原 = [11111110]反 = [11111111]补

可见原码， 反码和补码是完全不同的。既然原码才是被人脑直接识别并用于计算表示方式，为何还会有反码和补码呢?

首先，因为人脑可以知道第一位是符号位，在计算的时候我们会根据符号位选择对真值区域的加减。但是计算机要辨别"符号位"就必须获得全部的位的数据才可以，显然会让计算机的基础电路设计变得十分复杂！ 于是人们想出了将符号位也参与运算的方法。 我们知道， 根据运算法则减去一个正数等于加上一个负数， 即: 1-1 = 1 + (-1) = 0, 所以机器可以只有加法而没有减法，这样计算机运算的设计就更简单了。于是人们开始探索 将符号位参与运算，并且只保留加法的方法。

看个例子，计算十进制的表达式: 1-1=0，首先看原码的表示:

```
1 - 1 = 1 + (-1) = [00000001]原 + [10000001]原 = [10000010]原 = -2
```

如果用原码表示，让符号位也参与计算，显然对于减法来说，结果是不正确的，这也是为何计算机内部不使用原码表示一个数。

为了解决原码做减法的问题就出现了反码，此时计算十进制的表达式为: 1-1=0

```
1 - 1 = 1 + (-1)
= [0000 0001]原 + [1000 0001]原
= [0000 0001]反 + [1111 1110]反
= [1111 1111]反 = [1000 0000]原
= -0
```

可以看到用反码计算减法结果的真值部分是正确的，但是"0"的表示有点奇怪，+0和-0是一样的，而且0带符号是没有任何意义，而且要浪费[0000 0000]原和[1000 0000]原两个编码来表示0。于是补码的出现，解决了0的符号以及两个编码的问题:

```
1-1 = 1 + (-1) =
[0000 0001]原 + [1000 0001]原
= [0000 0001]补 + [1111 1111]补
= [0000 0000]补=[0000 0000]原
```

这样0用[0000 0000]表示， 而以前出现问题的-0则不存在了，而且可以用[1000 0000]表示-128:

```
(-1) + (-127) =
[1000 0001]原 + [1111 1111]原
= [1111 1111]补 + [1000 0001]补
= [1000 0000]补
```

-1-127的结果应该是-128，我们正好可以用[1000 0000]来表示-128，这样使用补码表示的范围为[-128, 127]，这一点也比原码的[-127,127]好。拓展一下，对于编程中常用到的32位int类型，可以表示范围是: [-2^31, 2^31-1] ，这也是我们在应用中经常见到的定义方式。

1.2 位运算规则

本节的内容很多你可能学过，但是请再认真学一遍，因为大量的算法解决思路都是从这里引申出来的

计算机采用的是二进制，二进制包括两个数码：0，1。在计算机的底层，一切运算都是基于位运算实现的，所以研究清楚位运算可以加深我们对很多基础原理的理解程度。

在算法方面，不少题目都是基于位运算拓展而来的，而且还有一定的技巧，如果不提前学一学，面试时很难想到。

位运算主要有：与、或、异或、取反、左移和右移，其中左移和右移统称移位运算，移位运算又分为算术移位和逻辑移位。

1.2.1 与、或、异或和取反

与运算的符号是 &，运算规则是：对于每个二进制位，当两个数对应的位都为 1 时，结果才为 1，否则结果为 0。

```
0 & 0=0
0 & 1=0
1 & 0=0
1 & 1=1
```

或运算的符号是 |，运算规则是：对于每个二进制位，当两个数对应的位都为 0 时，结果才为 0，否则结果为 1。

```
0 | 0=0
0 | 1=1
1 | 0=1
1 | 1=1
```

异或运算的符号是 ⊕（在代码中用∧ 表示异或），运算规则是：对于每个二进制位，当两个数对应的位相同时，结果为 0，否则结果为 1。

```
0⊕0=0
0⊕1=1
1⊕0=1
1⊕1=0
```

取反运算的符号是 ~，运算规则是：对一个数的每个二进制位进行取反操作，0 变成 1，1 变成 0。

```
~0=1
~1=0
```

以下例子显示上述四种位运算符的运算结果，参与运算的数字都采用有符号的 8 位二进制表示。

46 的二进制表示是 00101110，51 的二进制表示是 00110011。考虑以下位运算的结果。

- 46&51的结果是34，对应的二进制表示是 00100010。
- 46|51 的结果是63，对应的二进制表示是 00111111。
- 46⊕51 的结果是29，对应的二进制表示是 00011101。
- ~46 的结果是−47，对应的二进制表示是 11010001。
- ~51 的结果是 −52，对应的二进制表示是 11001100。

1.2.2 移位运算

移位运算按照移位方向分类可以分成左移和右移，按照是否带符号分类可以分成算术移位和逻辑移位。

左移运算的符号是 <<，左移运算时，将全部二进制位向左移动若干位，高位丢弃，低位补 0。对于左移运算，算术移位和逻辑移位是相同的。

右移运算的符号是 >>。右移运算时，将全部二进制位向右移动若干位，低位丢弃，高位的补位由算术移位或逻辑移位决定：

- 算术右移时，高位补最高位；
- 逻辑右移时，高位补 0。

以下例子显示移位运算的运算结果，参与运算的数字都采用有符号的 8 位二进制表示。

- 示例1：29 的二进制表示是 00011101。29左移 2 位的结果是 116，对应的二进制表示是 01110100；29 左移 3 位的结果是 −24，对应的二进制表示是 11101000。
- 示例2：50的二进制表示是 00110010。50 右移 1 位的结果是 25，对应的二进制表示是 00011001；50 右移 2 位的结果是 12，对应的二进制表示是 00001100。对于 0和正数，算术右移和逻辑右移的结果是相同的。
- 示例3：-50的二进制表示是 11001110(补码)。-50 算术右移 2 位的结果是 −13，对应的二进制表示是 11110011；−50 逻辑右移 2位的结果是 51，对应的二进制表示是 00110011。

右移运算中的算术移位和逻辑移位是不同的，计算机内部的右移运算采取的是哪一种呢?

- 对于 C/C++ 而言，数据类型包含有符号类型和无符号类型，其中有符号类型使用关键字signed 声明，无符号类型使用关键字 unsigned 声明，两个关键字都不使用时，默认是有符号类型。对于有符号类型，右移运算为算术右移；对于无符号类型，右移运算为逻辑右移。
- 对于 Java 而言，不存在无符号类型，所有的表示整数的类型都是有符号类型，因此需要区分算术右移和逻辑右移。在Java 中，算术右移的符号是 >>，逻辑右移的符号是 >>>。

1.2.3 移位运算与乘除法的关系

观察上面的例子可以看到，移位运算可以实现乘除操作。由于计算机的底层的一切运算都是基于位运算实现的，因此使用移位运算实现乘除法的效率显著高于直接乘除法的。

左移运算对应乘法运算。将一个数左移 k位，等价于将这个数乘以 2^k。例如，29 左移 2 位的结果是 116，等价于 29×4。当乘数不是 2 的整数次幂时，可以将乘数拆成若干项 2 的整数次幂之和，例如，a×6 等价于 (a<<2)+(a<<1)。对于任意整数，乘法运算都可以用左移运算实现，但是需要注意溢出的情况，例如在 8 位二进制表示下，29 左移 3 位就会出现溢出。

算术右移运算对应除法运算，将一个数右移 k 位，相当于将这个数除以 2^k。例如，50 右移 2 位的结果是 12，等价于 50/4，结果向下取整。

从程序实现的角度，考虑程序中的整数除法，是否可以说，将一个数（算术）右移 k 位，和将这个数除以 2^k等价？对于 0 和正数，上述说法是成立的，整数除法是向 0 取整，右移运算是向下取整，也是向 0 取整。但是对于负数，上述说法就不成立了，整数除法是向 0 取整，右移运算是向下取整，两者就不相同了。例如，(−50)>>2 的结果是 −13，而 (−50)/4 的结果是 −12，两者是不相等的。因此，将一个数（算术）右移 k 位，和将这个数除以 2^k是不等价的。算法出题这早就考虑到了这一点，因此在大部分算法题都将测试数据限制在正数和0的情况，因此可以放心的左移或者右移。

1.2.4 位运算常用技巧

位运算的性质有很多，此处列举一些常见性质，假设以下出现的变量都是有符号整数。

- 幂等律：a &a=a，a | a=a（注意异或不满足幂等律）；
- 交换律：a & b=b & a，a | b=b | a，a⊕b=b⊕a；
- 结合律：(a & b) & c=a & (b & c)，(a | b) | c=a | (b | c)，(a⊕b)⊕c=a⊕(b⊕c)；
- 分配律：(a & b) | c=(a | c) & (b | c)，(a | b) & c=(a & c) | (b & c)，(a⊕b) & c=(a & c)⊕(b & c)；
- 德摩根律：~(a & b)=(~a) | (~b)，~(a | b)=(~a) & (~b)；
- 取反运算性质：−1=~0，−a=~(a−1)；
- 与运算性质：a & 0=0，a & (−1)=a，a & (~a)=0；
- 或运算性质：a | 0=a，a | (~a)=−1；
- 异或运算性质：a⊕0=a，a⊕a=0；

根据上面的性质，可以得到很多处理技巧，这里列举几个：

- a & (a−1) 的结果为将 a 的二进制表示的最后一个 1 变成 0；
- (补码)a & (−a)的结果为只保留 a 的二进制表示的最后一个 1，其余的 1 都变成 0。

处理位操作时，还有很多技巧，不要死记硬背，理解其原理对解决相关问题有很大帮助。下面的示例中，1s和0s分别表示与x等长的一串1和一串0：

```
x ^ 0s = x      x & 0s = 0      x | 0s = x
x ^ 1s = ~x     x & 1s = x      x | 1s = 1s
x ^ x = 0       x & x = x       x | x = x
```

而如何获取、设置和更新某个位的数据，也有固定的套路。例如：

1.获取

该方法是将1左移i位，得到形如00010000的值。接着堆这个值与num执行”位与“操作，从而将i位之外的所有位清零，最后检查该结果是否为零。不为零说宁i位为1，否则i位为0。代码如下：

```
def getBit(num, i):
    return ((num & (1 << i)) != 0)
```

2.设置

setBit先将1左移i位，得到形如00010000的值，接着堆这个值和num执行”位或“操作，这样只会改变i位的数据。这样除i位外的位均为零，故不会影响num的其余位。代码如下：

```
def setBit(int num,int i):
 return num |(1<<i);
```

3. 清零

该方法与setBit相反，首先将1左移i位获得形如00010000的值，对这个值取反进而得到类似11101111的值，接着对该值和num执行”位与“，故而不会影响到num的其余位，只会清零i位。

```
 def clearBit(int num ,int i):
  int mask=~(1<<i);
  return num&mask;
```

4.更新

这个方法是将setBit和clearBit合二为一，首先用诸如11101111的值将num的第i位清零。接着将待写入值v左移i位，得到一个i位为v但其余位都为0的数。最后对之前的结果执行”位或“操作，v为1这num的i位更新为1，否则为0：

```
def updateBit(int num,int i,int v):
  int mask=~(1<<i);
  return (num&mask)|(v<<i);
```

上面的几种方式都要在透彻理解，不要死记硬背，很多棘手问题就能逐步拆解出解决的方法。

2.位运算高频算法题

与位运算和数学有关的题目真不少，而且很多都有一定的技巧，好在这些技巧相对是固定的，我们做好积累就行了。

2.1 位移的妙用

位移操作是一个很重要的问题，可以统计数字中1的个数，在很多高性能软件中也大量应用，我们看几个高频题目。

2.1.1 位1的个数

LeetCode191 编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数。

```
示例1：
输入：00000000000000000000000000001011
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。

示例2：
输入：00000000000000000000000010000000
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。
```

首先我们可以根据题目要求直接计算，题目给定的 n 是 32 位二进制表示下的一个整数，计算位 1 的个数的最简单的方法是遍历 n 的二进制表示的每一位，判断每一位是否为 1，同时进行计数。

那问题就是如何通过位运算来识别到1，例如：00001001001000100001100010001001，首先我们注意到要识别到最低位的1，可以这么做：

```
  00001001001000100001100010001001
& 00000000000000000000000000000001
= 00000000000000000000000000000001
```

也就说将原始数字和1进行&运算就能知道最低位是不是1了，那其他位置怎么算呢?

我们可以有两种思路，让1不断左移或者将原始数据不断右移。例如将原始数据右移就是：

```
  00000100100100010000110001000100
& 00000000000000000000000000000001
= 00000000000000000000000000000000
```

很明显此时就可以判断出第二位是0，然后继续将原始数据右移就可以依次判断出每个位置是不是1了。因此是不是1，计算一下(n>>i) & 1就可以了，所以代码顺理成章：

```
def hammingWeight(self, n) :
        ret = sum(1 for i in range(32) if n & (1 << i))
        return ret
```

这个题也可以通过将1左移来实现的，该问题作为一个作业题，请你改造上面的代码来实现。

上面的代码写出来，这个题基本就达标了，但这还不是最经典的解法，我们继续分析：

按位与运算有一个性质：对于整数 n，计算n & (n−1) 的结果为将 n 的二进制表示的最后一个 1 变成 0。利用这条性质，令 n=n & (n−1)，则 n 的二进制表示中的 1 的数量减少一个。重复该操作，直到 n 的二进制表示中的全部数位都变成 0，则操作次数即为 n 的位 1 的个数。什么意思呢？我们继续看上面的例子：

```
n:       00000100100100010000110001000100
n-1:     00000100100100010000110001000011
n&(n-1)= 00000100100100010000110001000000
```

可以看到此时n&(n-1)的结果比n少了一个1，此时我们令n=n&(n-1)，继续执行上述操作：

```
n:        00000100100100010000110001000000
n-1:      00000100100100010000110000111111
n&(n-1)= 00000100100100010000110000000000
```

可以看到此时n&(n-1)的结果比上一个n又少了一个1，所以我们令n=n&(n-1)，循环执行上述操作，我们统计一下循环执行的次数就能得到结果了。

那循环该什么时候停下呢？很显然当n变成0的时候，否则说明数据里面还有1，可以继续循环。所以当且仅当 n=0 时，n 的二进制表示中的全部数位都是 0，代码也很好写了：

```
def hammingWeight(self, n: int) -> int:
        ret = 0
        while n:
            n &= n - 1
            ret += 1
        return ret
```

上面两种解法，第一种的循环次数取决于原始数字的位数，而第二种的取决于1的个数，效率自然要高出不少，使用n = n & (n - 1)计算是位运算的一个经典技巧，该结论可以完美用到下面的题目中：

2.1.2 比特位计数

LeetCode338.给你一个整数 `n` ，对于 `0 <= i <= n` 中的每个 `i` ，计算其二进制表示中 **`1` 的个数** ，返回一个长度为 `n + 1` 的数组 `ans` 作为答案。

```
示例1：
输入：n = 2
输出：[0,1,1]
解释：0到n有 0 1 2 三个数字，每个数字含有1的个数分别为0 1 1 个，如下：
0 --> 0
1 --> 1
2 --> 10

示例2：
输入：n = 5
输出：[0,1,1,2,1,2]
解释：0到n有 0 1 2 3 4 5 六个数字，每个数字含有1的个数分别为0,1,1,2,1,2个，如下：
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101
```

最直观的方法是对从 0 到 num 的每个数直接计算"一比特数"。每个int 型的数都可以用 32 位二进制数表示，只要遍历其二进制表示的每一位即可得到1 的数目。

```
def countBits(self, n):
        bits = [0]
        highBit = 0
        for i in range(1, n + 1):
            if i & (i - 1) == 0:
                highBit = i
            bits.append(bits[i - highBit] + 1)
        return bits
```

利用位运算的技巧，可以提升计算速度。按位与运算（&）的一个性质是：对于任意整数 x，令 x=x&(x−1)，该运算将 x 的二进制表示的最后一个 1 变成 0。因此，对 x 重复该操作，直到 x 变成0，则操作次数即为 x 的「一比特数」。

```
def countBits(self, n: int) -> List[int]:
        def countOnes(x: int) -> int:
            ones = 0
            while x > 0:
                x &= (x - 1)
                ones += 1
            return ones

        bits = [countOnes(i) for i in range(n + 1)]
        return bits
```

有没有发现比特位计数和位1的个数计算规则完全一样？ 这就是为什么我们说研究清楚一道题，可以干掉一大票的题目。

2.1.3 颠倒无符号整数

LeetCode190 .颠倒给定的 32 位无符号整数的二进制位。 提示：输入是一个长度为32的二进制字符串。

```
示例1：
输入：n = 00000010100101000001111010011100
输出：964176192 (00111001011110000010100101000000)
解释：输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596，
     因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。

示例2：
输入：n = 11111111111111111111111111111101
输出：3221225471 (10111111111111111111111111111111)
解释：输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293，
     因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111 。
```

首先这里说是无符号位，那不必考虑正负的问题，最高位的1也不表示符号位，这就省掉很多麻烦。

我们注意到对于 n 的二进制表示的从低到高第 i 位，在颠倒之后变成第 31-i 位(0≤i<32)，所以可以从低到高遍历 n 的二进制表示的每一位，将其放到其在颠倒之后的位置，最后相加即可。

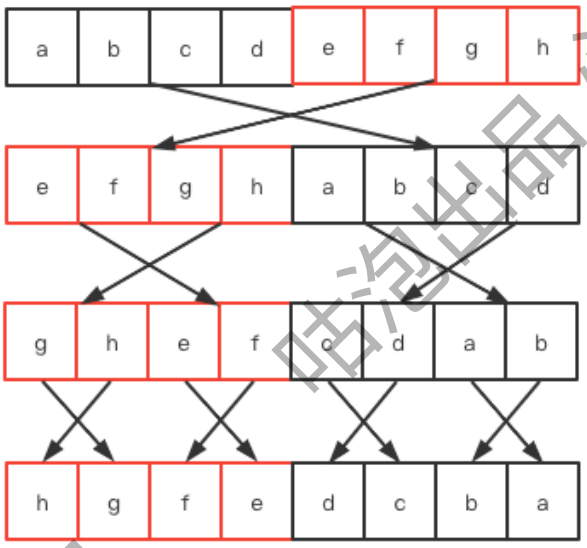
看个例子，为了方便我们使用比较短的16位演示：

```
原始数据：1001 1111 0000 0110(低位)
第一步：获得n的最低位0，然后将其左移16-1=15位，得到：
reversed：  0*** **** **** ****
n左移一位：  0100 1111 1000 0011

第二步：继续获得n的最低位1，然后将其左移15-1=14位，并与reversed相加得到：
reversed：01** **** **** ****
n左移一位：0010 0111 1100 0001

继续，一直到n全部变成0：
```

理解之后，实现就比较容易了。由于 Java不存在无符号类型，所有的表示整数的类型都是有符号类型，因此需要区分算术右移和逻辑右移，在Java 中，算术右移的符号是 >>，逻辑右移的符号是 >>>。

```
def reverseBits(self, n):
        res = 0
        for i in range(32):
            res = (res << 1) | (n & 1)
            n >>= 1
        return res
```

本题的解法还有很多，例如还有一种分块的思想， n 的二进制表示有 32 位，可以将 n 的二进制表示分成较小的块，然后将每个块的二进制位分别颠倒，最后将每个块的结果合并得到最终结果。这分治的策略，将 n 的 32 位二进制表示分成两个 16 位的块，并将这两个块颠倒；然后对每个 16 位的块重复上述操作，直到达到 1 位的块。为了方便看清楚，我们用字母代替01，如下图所示。



具体做法是：

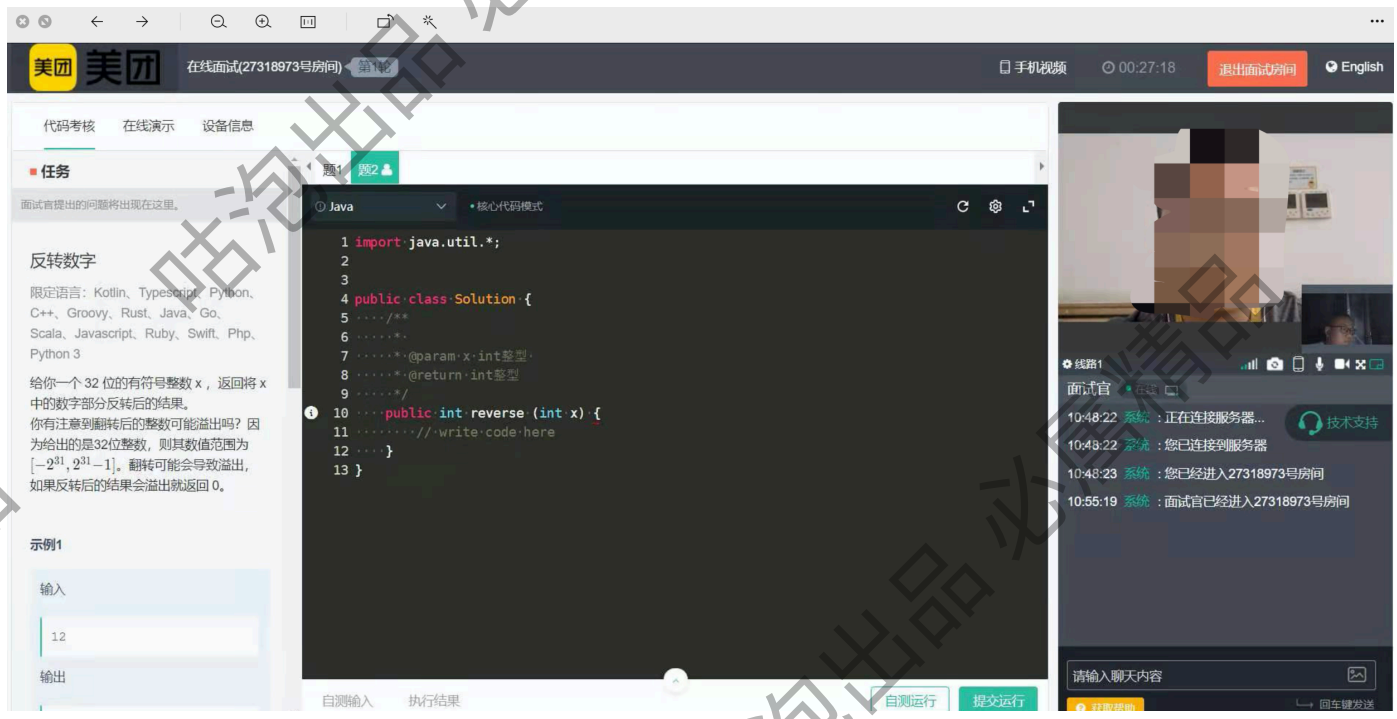
下面的代码中，每一行分别将 n 分成16 位、8 位、4 位、2 位、1 位的块，即把每个块分成两个较小的块，并将分成的两个较小的块颠倒。同样需要注意，使用 Java 实现时，右移运算必须使用逻辑右移。由于是固定的32位，我们不必写循环或者递归，直接写：

```
def reverseBits(self, n):
        n = (n >> 16) | (n << 16);
        n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
        n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
        return n;
```

这种方法在JDK、Dubbo等源码中都能见到，特别是涉及协议解析的场景几乎都少不了位操作。积累相关的技巧，可以方便面试，也有利于阅读源码。

2.2 溢出问题

溢出问题是一个极其重要的问题，只要涉及到输出一个数字，都可能遇到，典型的题目有三个：数字反转，将字符串转成数字和回文数。 不过溢出问题一般不会单独考察，甚至面试官都不会提醒你，但他就像捕捉猎物一样盯着你，看你会不会想到有溢出的问题，例如这道题是一个小伙伴面美团时拍的。所以凡是涉及到输出结果为数字的问题，必须当心！



溢出处理的技巧都是一致的，接下来我们就看一下如何处理。

2.2.1 整数反转

LeetCode7 给你一个 32 位的有符号整数 x ，返回将 x 中的数字部分反转后的结果。如果反转后整数超过 32 位的有符号整数的范围 [−2^31，2^31 − 1] ，就返回 0。假设环境不允许存储 64 位整数（有符号或无符号）。

示例：

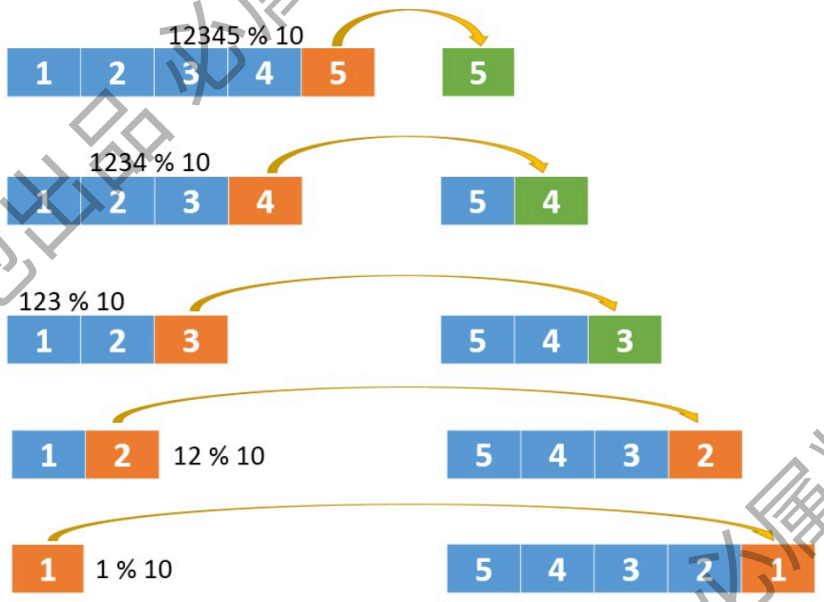
```
输入：x = 123   输出：321
输入：x = -123  输出：-321
输入：x = 120   输出：21
输入：x = 0     输出：0
```

这个题的关键有两点，一个是如何进行数字反转，另一个是如何判断溢出。反转好说，那为什么会有溢出问题呢？例如1147483649这个数字，它是小于最大的32位整数2147483647的，但是将这个数字反转过来后就变成了9463847411，这就比最大的32位整数还要大了，这样的数字是没法存到int里面的，所以就溢出了。

首先想一下，怎么去反转一个整数。用栈？或者把整数变成字符串再反转字符串？都可以但都不好。我们只要一边左移一边处理末尾数字就可以了。以12345为例，先拿到5，再拿到4，之后是3，2，1，然后就可以反向拼接出一个数字了。那如何获得末尾数字呢？好办，循环取模运算即可。例如：

```
1.将12345 % 10 得到5，之后将12345 / 10=1234
2.将1234 % 10 得到4，再将1234 / 10=123
3.将123 % 10 得到3，再将123 / 10=12
4.将12 % 10 得到2，再将12 / 10=1
5.将1 % 10 得到1，再将1 / 10=0
```

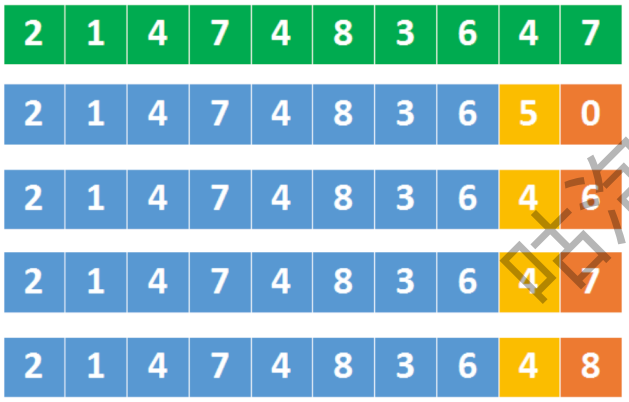
画成图就是：



这样的话，是不是将循环的判断条件设为x>0就可以了呢？不行！因为忽略了负数的问题，应该是while(x!=0)。去掉符号，剩下的数字，无论正数还是负数，按照上面不断的/10这样的操作，最后都会变成0，所以判断终止条件就是!=0。
有了取模和除法操作，就可以轻松解决第一个问题，如何反转。

接下来看如何解决溢出的问题。我们知道32位最大整数是MAX=2147483647，如果一个整数num>MAX，那么应该有以下规律：

nums/10 > MAX/10=214748364，也就是如果底数第二位大于4了，不管最后一位是什么都已经溢出了，如下：



所以我们要从到最大数的1/10时，就要开始判断，也即：

- 如果 num>214748364 那后面就不用再判断了，肯定溢出了。
- 如果num= 214748364，这对应到上图中第三、第四、第五排的数字，需要要跟最大数的**末尾数字**比较，如果这个数字比`7`还大，说明溢出了。
- 如果num<214748364，则没问题，继续处理。

这个结论对于负数也是一样的，所以实现代码就是：

```
def reverse(self, x: int) -> int:
        INT_MIN, INT_MAX = -2**31, 2**31 - 1

        rev = 0
        while x != 0:
            # INT_MIN 也是一个负数，不能写成 rev < INT_MIN // 10
            if rev < INT_MIN // 10 + 1 or rev > INT_MAX // 10:
                return 0
            digit = x % 10
            # Python3 的取模运算在 x 为负数时也会返回 [0, 9) 以内的结果，因此这里需要进行特殊判断
            if x < 0 and digit > 0:
                digit -= 10

            # 同理，Python3 的整数除法在 x 为负数时会向下（更小的负数）取整，因此不能写成 x //= 10
            x = (x - digit) // 10
            rev = rev * 10 + digit

        return rev
```

2.2.2 字符串转整数

LeetCode8.意思就是字符串转整数(atoi函数)，题目比较长，解决过程中要涉及很多异常情况的处理，我们在《字符串》部分再详细讲解，这里只看一下代码里是如何处理数字溢出问题的。

代码前部分是在处理字符串中可能存在的空格、前导0等等，后部分有【4.2】位置处，就是在判断溢出。不过呢，在python里，数字本来就能表示到64位的，因此这里就不用考虑溢出的问题了。虽然如此，但是我们必须要理解。

```
    def myAtoi(self, s):
```

```
        if s.isspace() or s == "":
            return 0
        s = s.strip(" ")
        # print(s)
        mid = s.split()
        # print(mid)
        char = 1
        sum1 = ""
        if s[0].isalpha():
            return 0
        for i in mid:
            if i[0].isalpha():
                continue
            elif i[0] == "-":
                target = i
                break
            elif i[0] == "+":
                target = i
                break
            elif i[0].isdigit():
                target = i
                break
            elif i[0] == ".":
                return 0
        if target[0].isdigit():
            for t in target:
                if t.isdigit():
                    sum1 += t
                else:
                    break
        if target[0] == "-":
            char = 0
            for t in target[1:]:
                if t.isdigit():
                    sum1 += t
                else:
                    break
        if target[0] == "+":
            for t in target[1:]:
                if t.isdigit():
                    sum1 += t
                else:
                    break
        if sum1 == "":
            return 0
        # #print(char)
        if char == 1:
            result = int(sum1)
        else:
```

```
            result = 0 - int(sum1)
        if result >= 2 ** (31) - 1:
            result = 2 ** (31) - 1
        elif result <= -2 ** (31):
            result = -2 ** (31)
        return result
```

2.2.3 回文数

LeetCode9 .给你一个整数 x ，如果 x 是一个回文整数，返回 true ；否则，返回 false 。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

例如，121 是回文，而 123 不是。

这个题可以将整数转换成字符串，然后通过字符串反转来实现比较，可以避免繁琐的数字处理。

如果非要使用数字来处理，我们首先想到的方式是计算原始数字反转后的结果，然后比较两者是否相等来确定，这样一反转就出现前面说的溢出问题。

不过在python中，直接直接使用切片方式轻松搞定。先将数转化为字符串，再利用python反向输出语法，判断是否为回文数。

```
    def isPalindrome(self, x):
        s = str(x)
        return s == s[::-1]
```

甚至还能用一行搞定：

```
def isPalindrome(self, x) :
        return str(x) == str(x)[::-1]
```

2.3.进制专题

进制问题也是一个非常重要的专题，有的直接处理还挺费劲，我们看两道题。

2.3.1 七进制数

LeetCode504.给定一个整数 `num`，将其转化为 **7 进制**，并以字符串形式输出。其中-10^7 <= num <= 10^7。

```
示例1:
输入: num = 100
输出: "202"
```

我们先通过二进制想一下7进制数的变化特征。在二进制中，先是0，然后是1，而2就是10(2)，3就是11(2)，4就是(100)。同样在7进制中，计数应该是这样的：

```
0 1 2 3 4 5 6 10 11 12 13 14 15 16 20 21 22 ...
```

给定一个整数将其转换成7进制的主要过程是循环取余和整除 ， 最后将所有的余数反过来即可。例如， 将十进制数 100 转成七进制：

```
100÷7=14 余 2
14÷7=2 余 0
2÷7=0 余 2
```

向遍历每次的余数，依次是 2、0、2，因此十进制数 100 转成七进制数是202 。如果num<0，则先对 num 取绝对值，然后再转换即可。使用代码同样可以实现该过程，需要注意的是如果单纯按照整数来处理会非常麻烦，既然题目说以字符串形式返回，那我们干脆直接用字符串类，代码如下：

```
 def convertToBase7(self, num) :
        if num == 0:
            return "0"
        negative = num < 0
        num = abs(num)
        digits = []
        while num:
            digits.append(str(num % 7))
            num //= 7
        if negative:
            digits.append('-')
        return ''.join(reversed(digits))
```

2.3.2 进制转换

给定一个十进制数M，以及需要转换的进制数N，将十进制数M转化为N进制数。M是32位整数，2<=N<=16。

这个题目的思路不复杂，但是想写正确却很不容易，甚至越写越糊涂。本题有好几个需要处理的问题：

- 1.超过进制最大范围之后如何准确映射到其他进制，特别是ABCDEF这种情况。简单的方式是大量采用if 判断，但是这样会出现写了一坨，最后写不下去。
- 2.需要对结果进行一次转置。
- 3.需要判断负号。

下面这个是我总结出的最精简，最容易理解的实现方案，其中最核心的是定义jz = "0123456789ABCDEF"，保存的是2到16的各个进制的值对应的标记，这样赋值时只计算下标，不必考虑不同进制的转换关系了。

```
#
# 进制转换
# @param M int整型 给定整数
# @param N int整型 转换到的进制
# @return string字符串
#
class Solution:
    def solve(self , M , N ):
        if M == 0: return "0" #如果M=0就直接返回
        flag = False #记录是不是负数
        if M &lt; 0 :
```

```
            #如果是负数flag=true，M 取相反数
            flag = True
            M = -M
        jz = "0123456789ABCDEF" #对应进制的某一位
        res = ""#返回最终的结果
        while M != 0:
            #就对应转换为N进制的逆序样子
            res += jz[M % N]
            M //= N
        if flag: #如果是负数就加一个－号
            res += "-"
        return res[::-1] #直接逆序返回
```

2.4. 数组实现加法

数字加法，小学生都会的问题，但是如果让你用数组来表示一个数，如何实现加法呢？理论上仍然从数组末尾向前挨着计算就行了，但是实现的时候会发现有很多问题，例如算到A[0]位置时发现还要进位该怎么办呢？

看一个用数组实现逐个加一的问题。LeetCode66.具体要求是由整数组成的非空数组所表示的非负整数，在其基础上加一。这里最高位数字存放在数组的首位， 数组中每个元素只存储**单个**数字。并且假设除了整数 0 之外，这个整数不会以零开头。例如：

```
输入：digits = [1,2,3]
输出：[1,2,4]
解释：输入数组表示数字 123。
```

这个看似很简单是不？从后向前依次加就行了，如果有进位就标记一下，但是如果到头了要进位怎么办呢？

例如如果digits = [9,9,9]，从后向前加的时候，到了A[0]的位置计算为0，需要再次进位但是数组却不能保存了，该怎么办呢？

这里的关键是A[0]什么时候出现进位的情况，我们知道此时一定是9，99，999...这样的结构才会出现加1之后再次进位，而进位之后的结果一定是10，100，1000这样的结构，由于java中数组默认初始化为0，所以我们此时只要申请一个空间比A[]大一个的数组B[]，然后将B[0]设置为1就行了，在python里默认值需要自己设置，这里设置一下就好了。代码就会变得非常简洁：

```
def plusOne(self, digits: List[int]):
        n = len(digits)
        for i in range(n - 1, -1, -1):
            if digits[i] != 9:
                digits[i] += 1
                for j in range(i + 1, n):
                    digits[j] = 0
                return digits

        # digits 中所有的元素均为 9
        return [1] + [0] * n
```

这里使用数组默认初始化为0的特性来大大简化了处理的复杂程度。如果使用的是C等默认值不是0的语言，我们只要在申请的时候先将所有的元素初始化为0就行了。

上面的题可以再拓展，假如给定的两个数，一个用数组存储的，另外一个是普通的整数，又该如何处理?

再拓展，如果两个整数是用字符串表示的呢？如果要按照二进制加法的规则来呢？这就是LeetCode67题，感兴趣的可以研究一下。

2.5 位实现运算

在计算机中，位运算的效率比加减乘数效率更高，因此在高性能软件的源码中大量应用，而且计算机里各种运算本质上都是位运算。本专题我们就研究几个相关问题。

2.5.1 位运算实现加法

LeetCode371 给你两个整数 `a` 和 `b` ，**不使用** 运算符 `+` 和 `-` ，计算并返回两整数之和。

```
示例1:
输入: a = 1, b = 2
输出: 3
```

既然不能使用+和-，那只能使用位运算了。我们看一下两个二进制位相加的情况：

```
[1] 0 + 0 = 0
[2] 0 + 1 = 1
[3] 1 + 0 = 1
[4] 1 + 1 = 0  (发生了进位，应该是10的)
```

两个位加的时候，我们无非就考虑两个问题：进位部分是什么，不进位部分是什么。从上面的结果可以看到，对于a和b两个数不进位部分的情况是：相同为0，不同为1，这不就是a⊕b吗?

而对于进位，我们发现只有a和b都是1的时候才会进位，而且进位只能是1，这不就是a&b=1吗？然后位数由1位变成了两位，也就是上面的[4]的样子，那怎么将1向前挪一下呢？手动移位一下就好了，也就是(a & b) << 1。所以我们得到两条结论：

- 不进位部分：用a⊕b计算就可以了。
- 是否进位，以及进位值使用(a & b) << 1计算就可以了。

于是，我们可以将整数 a 和 b 的和，拆分为 a 和 b 的无进位加法结果与进位结果的和，

在 Python 中，整数不是 32 位的，也就是说你一直循环左移并不会存在溢出的现象，这就需要我们手动对 Python 中的整数进行处理，手动模拟 32 位 INT 整型。

具体做法是将整数对 0x100000000 取模，保证该数从 32 位开始到最高位都是 0，代码就是：

```
class Solution(object):
    def getSum(self, a, b):
        # 2^32
        MASK = 0x100000000
        # 整型最大值
        MAX_INT = 0x7FFFFFFF
```

```
        MIN_INT = MAX_INT + 1
        while b != 0:
            # 计算进位
            carry = (a & b) << 1
            # 取余范围限制在 [0, 2^32-1] 范围内
            a = (a ^ b) % MASK
            b = carry % MASK
        return a if a <= MAX_INT else ~((a % MIN_INT) ^ MAX_INT)
```

2.5.2 递归乘法

LeetCode里面试08.05，递归乘法。 写一个递归函数，不使用 * 运算符， 实现两个正整数的相乘。可以使用加号、减号、位移，但要吝啬一些。

```
示例1:
输入：A = 1, B = 10
输出：10
```

如果不让用*来计算，一种是将一个作为循环的参数，对另一个进行累加，但是这样效率太低，所以我们还是要考虑位运算。

首先，求得A和B的最小值和最大值，对其中的最小值当做乘数（为什么选最小值，因为选最小值当乘数，可以算的少），将其拆分成2的幂的和，即min = a_0 * 2^0 + a_1 * 2^1 + ... + a_i * 2^i + ...其中a_i取0或者1。其实就是用二进制的视角去看待min，比如12用二进制表示就是1100，即1000+0100。例如：

13 * 12 = 13 * (8 + 4) = 13 * 8 + 13 * 4 = (13 << 3) + (13 << 2);

上面仍然需要左移5次，存在重复计算，可以进一步简化：

假设我们需要的结果是ans，

定义临时变量：tmp=13<<2 =52计算之后，可以先让ans=52

然后tmp继续左移一次tmp=52<<1=104，此时再让ans=ans+tmp

这样只要执行三次移位和一次加法，实现代码：

```
def multiply(self, A, B) :
        result = 0
        if A > B:
            for i in range(B):
                result+=A
        else:
            for i in range(A):
                result+=B
        return result
```

拓展 除法处理起来略微复杂的，感兴趣的同学可以研究一下LeetCode29，位运算实现除法。

2.6 幂运算

幂运算是常见的数学运算，其形式为 a^b ，即 a 的b次方，其中 a 称为底数，b 称为指数，a^b为合法的运算（例如不会出现 a=0且b≤0 的情况）。幂运算满足底数和指数都是实数。根据具体问题，底数和指数的数据类型和取值范围也各不相同。例如，有的问题中，底数是正整数，指数是非负整数，有的问题中，底数是实数，指数是整数。

力扣中，幂运算相关的问题主要是判断一个数是不是特定正整数的整数次幂，以及快速幂的处理。

2.6.1 求2的幂

LeetCode231. 给你一个整数 n，请你判断该整数是否是 2 的幂次方。如果是，返回 true ； 否则，返回 false 。

如果存在一个整数 x 使得 n == 2^x ，则认为 n 是 2 的幂次方。

```
示例1：
输入：n = 16
输出：true
解释：2^4 = 16

示例2：
输入：n = 3
输出：false
```

本题的解决思路还是比较简单的，我们可以用除的方法来逐步缩小n的值，另外一个就是使用位运算。

逐步缩小的方法就是如果 n 是 2 的幂，则 n>0，且存在非负整数 k 使得 n=2^k。

首先判断 n 是否是正整数，如果 n 是0 或负整数，则 n 一定不是 2 的幂。

当 n 是正整数时，为了判断 n 是否是 2 的幂，可以连续对 n 进行除以 2 的操作，直到 n 不能被 2 整除。此时如果 n=1，则 n 是 2 的幂，否则 n 不是 2 的幂。代码就是：

```
def isPowerOfTwo(int n) :
        if (n <= 0) :
            return false
        while (n % 2 == 0) :
            n /= 2
        return n == 1;
```

如果采用位运算，该方法与我们前面说的统计数字转换成二进制数之后1的个数思路一致。当 n>0 时，考虑 n 的二进制表示。如果存在非负整数 k 使得 n=2^k，则 n 的二进制表示为 1 后面跟 k 个0。由此可见，正整数 n 是2 的幂，当且仅当 n 的二进制表示中只有最高位是 1，其余位都是 0，此时满足 n & (n−1)=0。因此代码就是：

```
def isPowerOfTwo(self, n) :
        return n > 0 and (n & (n - 1)) == 0
```

2.6.2 求3的幂

leetcode 326 给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 true ；否则，返回 false 。整数 n 是 3 的幂次方需满足：存在整数 x 使得 n == 3^x

对于这个题，可以直接使用数学方法来处理，如果n是3 的幂，则 n>0，且存在非负整数 k 使得 n=3^k。

首先判断 n是否是正整数，如果 n是 0或负整数，则 n一定不是 3的幂。

当 n 是正整数时，为了判断 n 是否是 3 的幂，可以连续对 n 进行除以 3 的操作，直到 n 不能被 3 整除。此时如果 n=1，则 n 是 3 的幂，否则 n 不是 3 的幂。

```
def isPowerOfThree(self, n) :
       while n and n % 3 == 0:
           n //= 3
       return n == 1
```

这个题的问题和上面2的次幂一样，就是需要大量进行除法运算，我们能否优化一下呢？这里有个技巧。

由于给定的输入 n 是int 型，其最大值为 2^31-1。因此在int 型的数据范围内存在最大的 3 的幂，不超过 2^31-1 的最大的 3 的幂是 3^19=1162261467。所以如果在1~ 2^31-1内的数，如果是3的幂，则一定能被1162261467整除，所以这里可以通过一次除法就获得：

```
def isPowerOfThree(self, n: int) :
       return n > 0 and 1162261467 % n == 0
```

当然这个解法只是拓展思路的，没必要记住1162261467这个数字。

思考 如果这里将3换成4 ，5，6，7，8，9可以吗？如果不可以，那如果只针对素数 3 、5、 7、 11、 13可以吗?

2.6.3 求4的幂

LeetCode342 给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true ；否则，返回 false 。整数 n 是 4 的幂次方需满足：存在整数 x 使得 n == 4^x。

一种方法自然还是数学方法一直除，代码如下：

```
def isPowerOfFour(int n) :
  if (n <= 0):
        return false
  while (n % 4 == 0):
       n /= 4;
   return n == 1;
```

这个题可以利用2的次幂进行拓展来优化，感兴趣的同学自行查阅一下吧。

除了幂运算，指数计算的思路与之类似，感兴趣的同学可以研究一下LeetCode50，实现pow(x,n)这个题。

2.7 只出现一次的数字

LeetCode136 给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

```
示例1:
输入: [4,1,2,1,2]
输出: 4
```

对于这个题，你可能想到使用Hash、集合等等方法遍历寻找，这里介绍一种出奇简单的方法，只要将数组中的全部元素的异或即可，但是我们要搞清楚为什么可以这样做。

要将空间复杂度降到 O(1)，则需要使用按位异或运算⊕。异或运算具有以下三个性质：

- 任何数和 0 做异或运算，结果仍然是原来的数，即 a⊕0=a；
- 任何数和其自身做异或运算，结果是 0，也即a⊕a=0；
- 异或运算满足交换律和结合律，即 a⊕b⊕a=b⊕a⊕a=b⊕(a⊕a)=b⊕0=b。

假设数组 nums 的长度为 n，由于数组 nums 中只有一个元素出现了一次，其余的元素都出现了两次，因此 n 是奇数。令 m=(n-1)/2，则 n=2m+1，即数组nums 中有 m 个元素各出现两次，剩下一个元素出现一次。假设出现两次的元素分别是 a_1,a_2,...,a_m，只出现一次的元素是a*(m+1)* 。*利用异或运算的性质，对全部元素进行异或运算，结果即为a*(m+1)：

$$
\begin{aligned}
& nums[0] \oplus nums[1] \oplus \ldots \oplus nums[n-1] \\
&= a_1 \oplus a_1 \oplus a_2 \oplus a_2 \oplus \ldots \oplus a_m \oplus a_m \oplus a_{m+1} \\
&= (a_1 \oplus a_1) \oplus (a_2 \oplus a_2) \oplus \ldots \oplus (a_m \oplus a_m) \oplus a_{m+1} \\
&= 0 \oplus 0 \oplus \ldots \oplus 0 \oplus a_{m+1} \\
&= a_{m+1}
\end{aligned}
$$

因此，本题的解法非常简单，只要将数组中的全部元素的异或运算结果即为数组中只出现一次的数字，而使用python3的lamba方式，可以变成一行搞定：

```
def singleNumber(self, nums: List[int]) -> int:
        return reduce(lambda x, y: x ^ y, nums)
```

3.总结

本章主要介绍了位运算和基本的数学问题，数学的问题还有很多，LeetCode里还有大量的算法题目，通过上面的学习你可以感受到这里有很多解题技巧，如果不提前练习的话，基本不可能想到 。所以这部分题目我们有必要持续积累。

本章的位运算部分介绍了很多经典的技巧，这些问题本身就经常出现在各厂的面试中，而且介绍的技巧有助于我们学习更多题目。因此请各位小伙伴务必重视。

除此之外，还有很多题目都值得研究，而且上面的题目很多都可以继续拓展出新题目，有精力的同学可以继续研究，例如出现次数问题可以继续进行如下拓展：

- LeetCode137 给你一个整数数组 `nums` ，除某个元素仅出现 **一次** 外，其余每个元素都恰出现 **三次** 。请你找出并返回那个只出现了一次的元素。
- LeetCode260 给定一个整数数组 `nums`，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出

只出现一次的那两个元素。你可以按 **任意顺序** 返回答案。