简介

本章介绍最后一种重要的结构数学和数字问题。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。数学的门类很多，涉及的范围很广，很多难度也超大，但是在算法中，一般只会选择各个学科的基础问题来考察，例如素数问题、幂、对数、阶乘、幂运算、初等数论、几何问题、组合数学等等，本章还是选择最热门最重要的问题来讲解。

1. 数字统计专题

统计一下特定场景下的符号，或者数字个数等是一类非常常见的问题 。如果按照正常方式强行统计，可能会非常复杂，所以掌握一些技巧是非常必要的。

1.1 符号统计

例如，LeetCode1822 给定一个数组，求所有元素的乘积的符号，如果最终答案是负的返回-1，如果最终答案是正的返回1，如果答案是0返回0。

仔细分析一下这道题目，如果将所有数都乘起来，再判断正负，工作量真不少，而且还可能溢出。我们发现，一个数如果是 -100 和 -1，对符号位的贡献是完全一样的，所以只需要看有多少个负数，就能够判断最后乘积的符号了。

```
class ArraySign:
    def arraySign(self, nums):
        sign = 1
        for num in nums:
            if num == 0:
                return 0
            if num < 0:
                sign = -sign
        return sign
```

1.2 阶乘0的个数

很多数学相关算法的关键在于找到怎么通过最简洁的方式来解决问题，而不是硬算。例如：面试题16.05：设计一个算法，算出 n 阶乘有多少个尾随零。

这个题如果硬算，一定会超时，其实我们可以统计有多少个 0，实际上是统计 2 和 5 一起出现多少对，不过因为 2 出现的次数一定大于 5 出现的次数，因此我们只需要检查 5 出现的次数就好了，那么在统计过程中，我们只需要统计 5、10、15、 25、 ... 5^n 这样 5 的整数倍项就好了，最后累加起来，就是多少个 0。代码就是：

```
class TrailingZeroes:
    def trailingZeroes(self, n):
        cnt = 0
        num = 5
        while n / num > 0:
            cnt += n / num
            num *= 5
        return cnt
```

数学不仅与算法难以区分 ，很多算法问题还与位运算密不可分，有些题目真不好说是该归类到数学中呢，还是位运算中。我们干脆就放在一起来看。

1.3 统计2的个数

本题是一种非常常见的问题，考察的热度很多。

题目要求：编写一个方法，输出从0到n(含)中数字2出现的次数。示例：输入25，输出9，(2,1,2,21,22,23,24,25,注意22有两个2)。

面对此题，我们想到的第一个方式是暴力计算。如果没有好的思路，我们完全可以从最简单的方式开始，这也是体现了自己不断思考的过程。

我们可以从2开始循环到n，依次判断每个数里2的个数。这个方法最大的问题就是效率，当n非常大时，就需要很长的运行时间。 另外，如果将2换成其他数字又会怎么样呢?

想要提高效率，就要避开暴力法，从数字中找出规律。假设一个5位数N=abcde，我们现在来考虑百位上出现2的次数，即，从0到abcde的数中， 有多少个数的百位上是2。分析完它，就可以用同样的方法去计算个位，十位，千位， 万位等各个位上出现2的次数。

当百位c为0时，比如说12013，0到12013中哪些数的百位会出现2？我们从小的数起， 200-299, 1200-1299, 2200-2299, ... , 11200-11299, 也就是固定低3位为200-299，然后高位依次从0到11，共12个。再往下12200-12299 已经大于12013，因此不再往下。所以，当百位为0时，百位出现2的次数只由更高位决定， 等于更高位数字(12)x当前位数(100)=1200个。

当百位c为1时，比如说12113。分析同上，并且和上面的情况一模一样。 最大也只能到11200-11299，所以百位出现2的次数也是1200个。

上面两步综合起来，可以得到第一个结论：**当某一位的数字小于2时，那么该位出现2的次数为：更高位数字x当前位数**

当百位c为2时，比如说12213。那么，我们还是有200-299, 1200-1299, 2200-2299, ... , 11200-11299这1200个数，他们的百位为2。但同时，还有一部分12200-12213， 共14个(低位数字+1)。所以，当百位数字为2时， 百位出现2的次数既受高位影响也受低位影响，结论如下：

当某一位的数字等于2时，那么该位出现2的次数为：更高位数字x当前位数+低位数字+1

当百位c大于2时，比如说12313，那么固定低3位为200-299，高位依次可以从0到12， 这一次就把12200-12299也包含了，同时也没低位什么事情。因此出现2的次数是： (更高位数字+1)x当前位数。结论如下：

当某一位的数字大于2时，那么该位出现2的次数为：(更高位数字+1)x当前位数

根据上面结论我们可以写出如下代码：

```
def countNumberOf2s(int n):
     count = 0;
     high;
    int low;
    int cur;
    for (int i = 1;i <= n;i *= 10){
        high = (n / i) / 10;//高位（不包含当前位置）
        low = n % i;//低位（包含当前位置）
        cur = (n / i) % 10;//当前位置
        if (cur < 2){
            count += high * i;
        }else if (cur > 2){
            count += (high + 1) * i;
        }else if (cur == 2){
            count += high * i + low + 1;
        }
    }
    return count;
}
```

如果我们把问题一般化一下：写一个函数，计算0到n之间i出现的次数，i是1到9的数。 这里为了简化，i没有包含0，因为按以上的算法计算0出现的次数， 比如计算0到11间出现的0的次数，会把1，2，3，4...视为01，02，03，04... 从而得出错误的结果。所以0是需要单独考虑的，为了保持一致性，这里不做讨论。 将上面的三条结论应用到这就是：

- 当某一位的数字小于i时，那么该位出现i的次数为：更高位数字x当前位数
- 当某一位的数字等于i时，那么该位出现i的次数为：更高位数字x当前位数+低位数字+1

- 当某一位的数字大于i时，那么该位出现i的次数为：(更高位数字+1)x当前位数

代码如下：

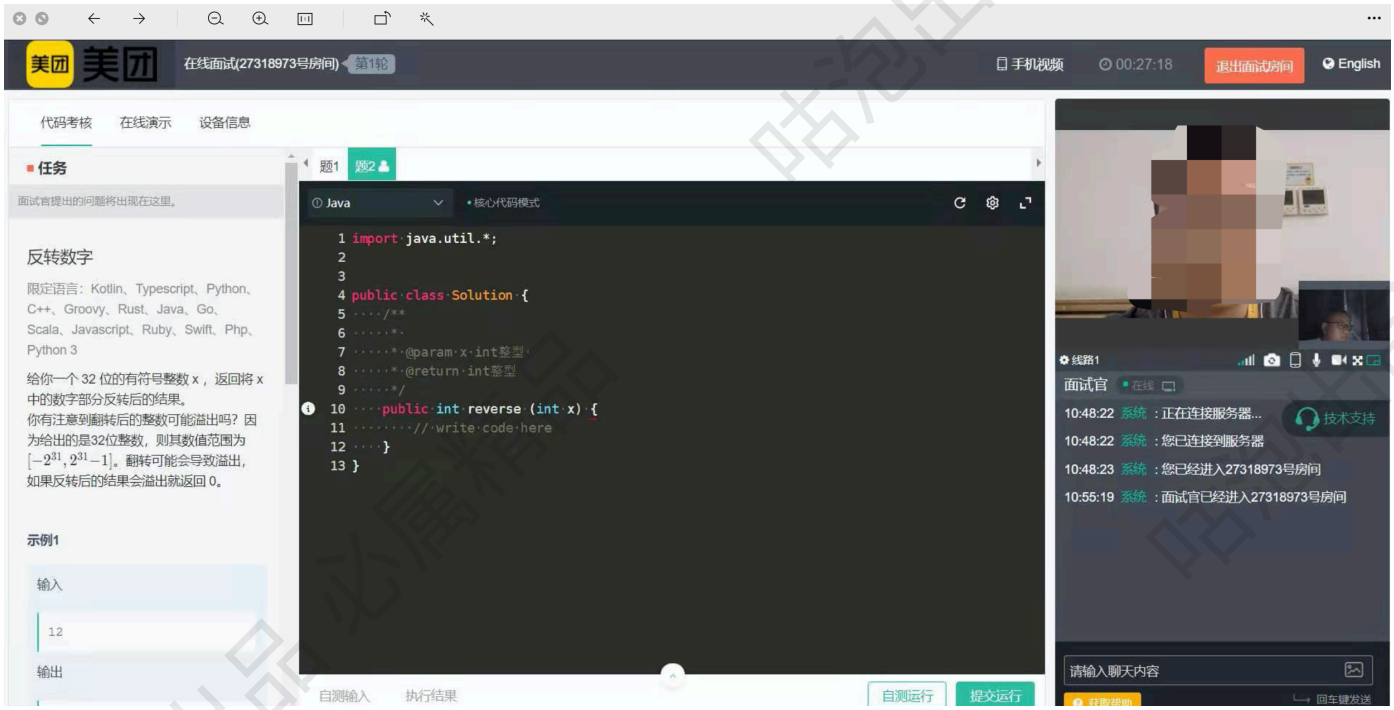
```
public int countNumberOfnums(int n,int num){
    if (num < 1 || num > 9){
        return -1;
    }
    int count = 0;
    int high;
    int low;
    int cur;
    for (int i = 1;i <= n;i *= 10){
        high = (n / i) / 10;//高位（不包含当前位置）
        low = n % i;//低位（包含当前位置）
        cur = (n / i) % 10;//当前位置
        if (cur < num){
            count += high * i;
        }else if (cur > num){
            count += (high + 1) * i;
        }else if (cur == num){
            count += high * i + low + 1;
        }
    }
    return count;
}
```

2.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 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 。

2.1.1 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：

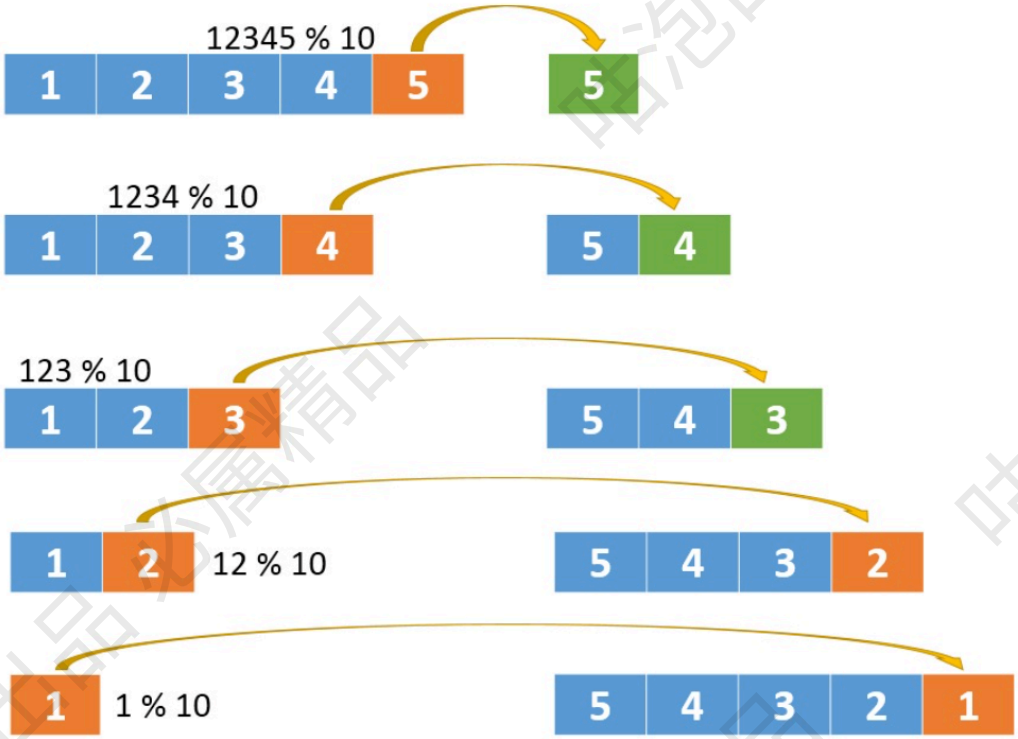
```
输入：x = 123  输出：321
输入：x = -123 输出：-321
输入：x = 120  输出：21
输入：x = 0    输出：0
```

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
```

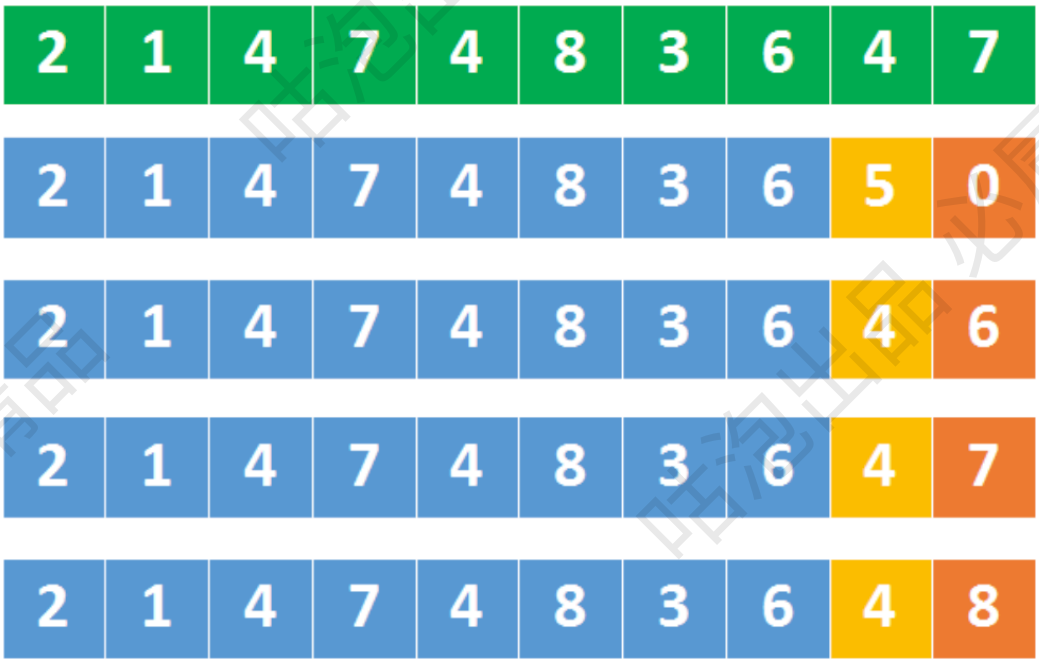
画成图就是：



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
```

2.1.2 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(" ")
        mid = s.split()
        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
        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
```

2.1.3 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]
```

2.2.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。

2.2.1 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))
```

2.2.2 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
```

2.3. 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。

2.4 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。

2.4.1 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
```

2.4.2 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?

2.4.3 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。

2.5 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。

如果对于数论等还有需求，请和我说，我们后面继续补充。