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。

```
public int arraySign(int[] nums) {
    int prod = 1;
    for(int i = 0; i < nums.length; ++i) {
        if(nums[i] == 0) {
            return 0;//一切归零
        }else if(nums[i] < 0) {
        //交替就够了，很好的处理技巧
            prod = -prod;
        }
     }
     return prod;
    }
```

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：

```
public int trailingZeroes(int n) {
        int cnt = 0;
        for (long num = 5; n / num > 0; num *= 5) {
            cnt += n / num;
        }
        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。因为要循环调用，所以将判断过程抽取成单独的方法numberOf2s()更好，代码如下：

```
public class QuestionBrute {
  public static int numberOf2sInRange(int n) {
    int count = 0;
    for (int i = 2; i <= n; i++) { // Might as well start at 2
      count += numberOf2s(i);
    }
    return count;
  }
  //统计每个数中2的个数。
  public static int numberOf2s(int n) {
    int count = 0;
    while (n > 0) {
      if (n % 10 == 2) {
        count++;
      }
      n = n / 10;
    }
    return count;
  }
  }
```

测试方法：

```
public static void main(String[] args) {
  for (int i = 0; i < 1000; i++) {
    int v = numberOf2sInRange(i);
    System.out.println("Between 0 and " + i + ": " + v);
  }
```

上述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：

```
    public int countNumberOf2s(int n){
        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 < 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：

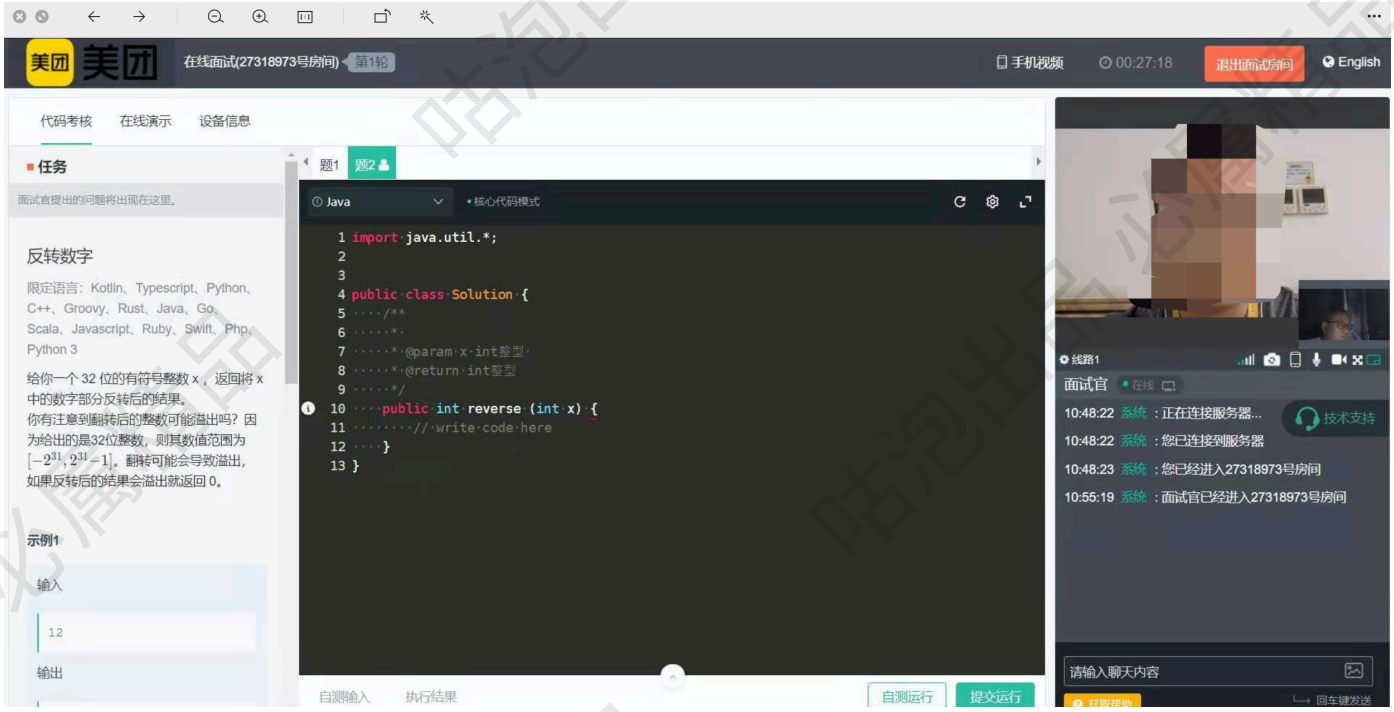
```
    public int countNumberOfnums(int n,int num){
        if (num < 1 || num > 9){
            return -1;
        }
```

```
        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. 数字问题

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：

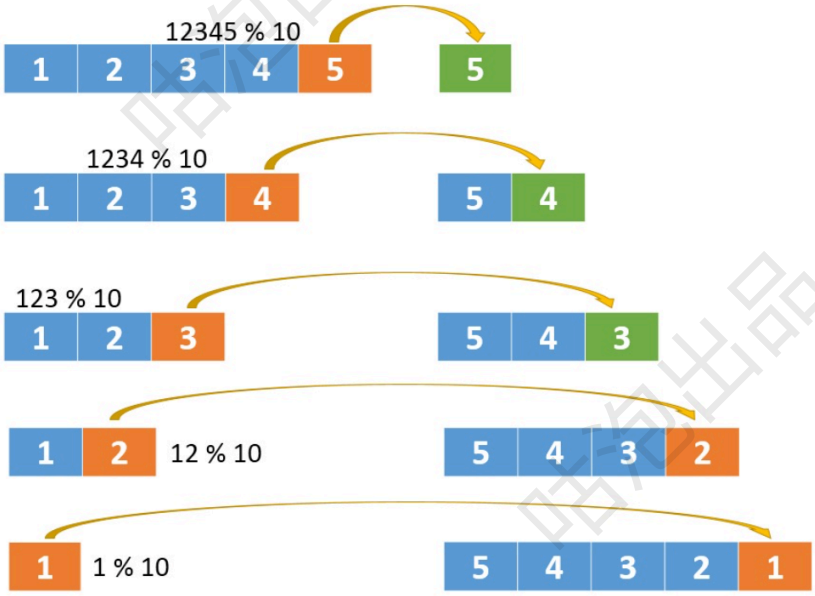
```
输入：x = 123   输出：321
输入：x = -123  输出：-321
输入：x = 120   输出：21
输入：x = 0     输出：0
```

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
```

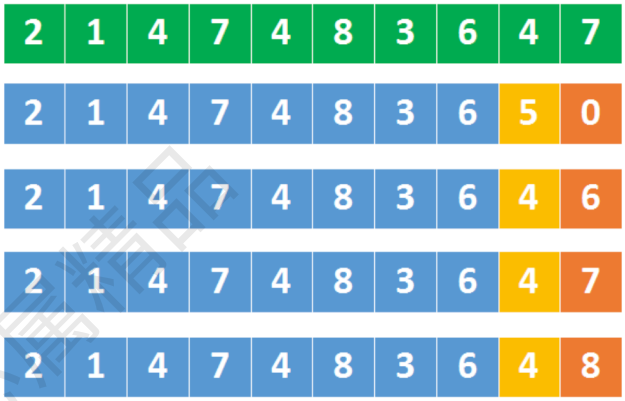
画成图就是：



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：

```
public int reverse(int x) {
        int res = 0;
        while(x!=0) {
            //获得末尾数字
            int tmp = x%10;
            //判断是否大于最大32位整数，也可以使用Integer.MAX_VALUE/10来代替214748364
            if (res>214748364 || (res==214748364 && tmp>7)) {
                return 0;
            }
            //判断是否小于最小的32位整数
            if (res<-214748364 || (res===-214748364 && tmp<-8)) {
                return 0;
            }
            res = res*10 + tmp;
            x /= 10;
        }
        return res;
    }
```

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。

```
public  int myAtoi(String str) {
        int len = str.length();
        char[] charArray = str.toCharArray();

        // 【1】去除前导空格
        int index = 0;
        while (index < len && charArray[index] == ' ') {
            index++;
        }

        //【2】如果已经遍历完成（针对极端用例 "       "）
        if (index == len) {
            return 0;
        }

        //【3】如果出现符号字符，仅第 1 个有效，并记录正负
        int sign = 1;
        char firstChar = charArray[index];
        if (firstChar == '+') {
            index++;
        } else if (firstChar == '-') {
            index++;
            sign = -1;
        }

        // 【4】将后续出现的数字字符进行转换
        // 不能使用 long 类型，这是题目说的
        int res = 0;
        while (index < len) {
            char currChar = charArray[index];
            //【4.1】先判断不合法的情况
            if (currChar > '9' || currChar < '0') {
                break;
            }

        // 【4.2】提前判断乘以10以后是否越界,但res*10可能会越界，所以这里使用
Integer.MAX_VALUE/10，这样一定不会越界。
            if (res > 214748364 || (res == Integer.MAX_VALUE / 10 && (currChar - '0') >
Integer.MAX_VALUE % 10)) {
                return Integer.MAX_VALUE;
            }
```

```
            if (res < Integer.MIN_VALUE / 10 || (res == Integer.MIN_VALUE / 10 &&
(currChar - '0') > -(Integer.MIN_VALUE % 10))) {
                return Integer.MIN_VALUE;
            }
            // 合法的情况下，才考虑转换，每一步都把符号位乘进去
            // 想想这里为什么要带着sign乘
            res = res * 10 + sign * (currChar - '0');
            index++;
        }
        return res;
    }
```

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，代码如下：

```
   public boolean isPalindrome(int x) {
        if(x<0) {
            return false;
        }
        String oldStr = ""+x;
        String newStr = new StringBuilder(""+x).reverse().toString();
        return oldStr.equals(newStr);
    }
```

如果非常使用数字来处理，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，我们还是不想处理溢出的问题，可以将反转结果转换成long类型：

```
    public boolean isPalindrome(int x) {
       if(x<0) {
            return false;
        }
        long ans = 0;
        int old = x;
        while(x>0) {
            ans = ans*10 + x%10;
            x /= 10;
        }
        return ans==old;
    }
```

奇怪的是这个题目不考虑溢出的问题，将上面的ans直接定义为int类型，也能通过，可能官方的case就没有给出溢出的情况吧，请小伙伴想一下，如果考虑溢出该如何调整。

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。

2.3.1 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:

```
    public String convertToBase7(int num) {
        StringBuilder sb = new StringBuilder();
        //先拿到正负号
        boolean sign = num < 0;
        //这样预处理一下，后面都按照正数处理num就行了
        if(sign)
            num *= -1;
        //循环取余和整除
        do{
            sb.append(num%7 + "");
            num/=7;
        }while(num > 0);
        //添加符号
        if(sign)
            sb.append("-");
         //上面的结果是逐个在末尾加的，需要反过来
        return sb.reverse().toString();
    }
```

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。注意采取三个措施来方便处理：

- 1.定义大小为16的数组F，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。
- 2.使用StringBuffer完成数组转置等功能，如果不记得这个方法，工作量直接飙升。
- 3.通过一个flag来判断正数还是负数，最后才处理。

```
public class Solution {
     // 要考虑到 余数 > 9 的情况，2<=N<=16.
    public static final String[] F = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
"A", "B", "C", "D", "E", "F"};
    //将十进制数M转化为N进制数
    public String convert (int M, int N) {
       Boolean flag=false;
        if(M<0){
            flag=true;
            M*=-1;
        }
        StringBuffer sb=new StringBuffer();
        int temp;
        while(M!=0){
            temp=M%N;
            //技巧一：通过数组F[]解决了大量繁琐的不同进制之间映射的问题
            sb.append(F[temp]);
            M=M/N;
        }
        //技巧二：使用StringBuffer的reverse()方法，让原本麻烦的转置瞬间美好
        sb.reverse();
        //技巧三：最后处理正负，不要从一开始就揉在一起。
        return (flag? "-":"")+sb.toString();
    }
}
```

2.3. 数组实现加法专题

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？

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？

2.3.1 数组实现整数加法

先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。这样代码就会变得非常简洁：

```
public static int[] plusOne(int[] digits) {
    int len = digits.length;
    for (int i = len - 1; i >= 0; i--) {
        digits[i]++;
        digits[i] %= 10;
        if (digits[i] != 0)
            return digits;
    }
    // 比较巧妙的设计
    digits = new int[len + 1];
    digits[0] = 1;
    return digits;
}
```

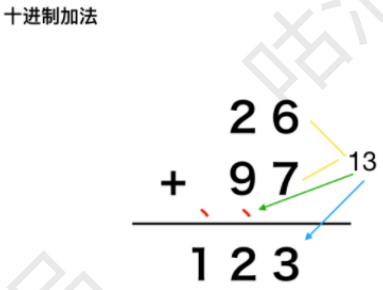
这里使用数组默认初始化为0的特性来大大简化了处理的复杂程度。如果使用的是C等默认值不是0的语言，我们只要在申请的时候先将所有的元素初始化为0就行了。

2.3.2 字符串加法

我们继续看将数字保存在字符串中的情况： 字符串加法就是使用字符串来表示数字，然后计算他们的和。具体要求如下：给定两个字符串形式的非负整数 num1 和num2 ，计算它们的和并同样以字符串形式返回。你不能使用任何內建的用于处理大整数的库（比如 BigInteger）， 也不能直接将输入的字符串转换为整数形式。

```
例如：
输入：num1 = "456", num2 = "77"
输出："533"
```

我们先想一下小学里如何计算两个比较大的数相加的，经典的竖式加法是这样的：



从低到高逐位相加，如果当前位和超过 10，则向高位进一位？因此我们只要将这个过程用代码写出来即可。先定义两个指针 i 和j 分别指向num1和num2的末尾，即最低位，同时定义一个变量 add 维护当前是否有进位，然后从末尾到开头逐位相加。

这里可能有个问题：两个数字位数不同该怎么处理？简单，补0即可。具体可以看下面的代码：

```
    public String addStrings(String num1, String num2) {
        int i = num1.length() - 1, j = num2.length() - 1, add = 0;
        StringBuffer ans = new StringBuffer();
        while (i >= 0 || j >= 0 || add != 0) {
            int x = i >= 0 ? num1.charAt(i) - '0' : 0;
            int y = j >= 0 ? num2.charAt(j) - '0' : 0;
            int result = x + y + add;
            ans.append(result % 10);
            add = result / 10;
            i--;
            j--;
        }
        // 计算完以后的答案需要翻转过来
        ans.reverse();
        return ans.toString();
    }
```

2.3.3 二进制加法

我们继续看，如果这里是二进制该怎么处理呢？详细要求：leetcode76.给你两个二进制字符串，这个字符串是用数组保存的，返回它们的和（用二进制表示）。其中输入为 非空 字符串且只包含数字 1 和 0。

```
示例1:
输入: a = "11", b = "1"
输出: "100"

示例2:
输入: a = "1010", b = "1011"
输出: "10101"
```

这个题也是用字符串来表示数据的，也要先转换为字符数组。我们熟悉的十进制，是从各位开始，逐步向高位加，达到10就进位，而对于二进制则判断相加之后是否为二进制的10，是则进位。本题解中大致思路与上述一致，但由于字符串操作原因，不确定最后的结果是否会多出一位进位，下面 2 种处理方式都可以：

- 第一种，在进行计算时直接拼接字符串，得到一个反向字符，最后再翻转。
- 第二种，按照位置给结果字符赋值，最后如果有进位，则在前方进行字符串拼接添加进位

我们这里采用第二种实现。

```
    public String addBinary(String a, String b) {
        StringBuilder ans = new StringBuilder();
        int ca = 0;
        for(int i = a.length() - 1, j = b.length() - 1;i >= 0 || j >= 0; i--, j--) {
            int sum = ca;
            sum += i >= 0 ? a.charAt(i) - '0' : 0;
            sum += j >= 0 ? b.charAt(j) - '0' : 0;
            ans.append(sum % 2);
            ca = sum / 2;
        }
        ans.append(ca == 1 ? ca : "");
        return ans.reverse().toString();
    }
```

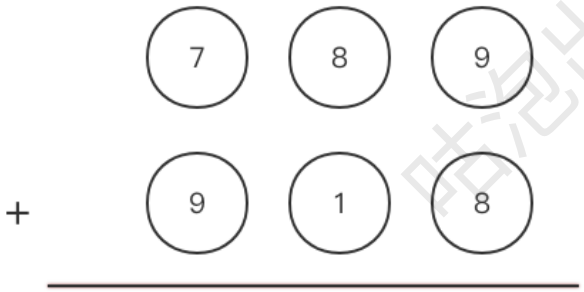
这里还有人会想，先将其转换成十进制，加完之后再转换成二进制可以吗？这么做实现非常容易，而且可以使用jdk的方法直接转换，但是还是那句话，工程里可以这么干，稳定可靠，但是算法里不行，太简单了。

2.4 高精度计算

如果面试官问你a+b等于几，那你肯定会直接甩出代码给他看，但是如果说这里的a和b是一个超过long类型的数，该怎么办呢？例如然让你计算123456789123456789123456789123456789+123456789123456789123456789123456788。在很多语言中会提供类似BigInteger的api来执行，那现在让你实现一下BigInteger来完成加减乘除，该怎么办呢？

2.4.1 高精度加法

在小学的时候，我们就学习过，所谓的加法其实是这么加的：



从低位开始，两个相加，如果和超过9就开始进位，我们用a和b表示被加的两个数字，next表示上一位的进位值。每个位置计算的时候是这样的：

`target=a+b+next`

那target是直接写还是要进位则取决于target是否小于10，小于的时候直接放，否则就进位，判断依据是：

```
存入值：c=target 模10
进位值：next=target/10
```

这样，我们可以返璞归真，使用数组来表示数字，然后通过数组来实现加法。不过要注意的是加法是从低位向高位相加，而数组的0号位是最大值，因此遍历的时候要从后向前遍历。

```
public static List<Integer> add(List<Integer> listA, List<Integer> listB) {
    if (listA.size() < listB.size()) {
        return add(listB, listA);
    }
    int next = 0;
    List<Integer> listC = new ArrayList<>();
    for (int i = 0; i < listA.size(); i++) {
        next += listA.get(i);
        if (i < listB.size()) {
            next += listB.get(i);
        }
        listC.add(next % 10);
        next /= 10;
    }
    if (next != 0) {
        listC.add(next);
    }
    return listC;
}
```

2.4.2 高精度乘法

上面说的加法也适用于乘法，只要将两个数的加操作换成乘法就可以了。但是乘法要将乘数从小到大逐个与被乘数运算，因此需要针对乘数再执行一次循环。这里我们只考虑乘数是一个数字的情况。

这里还有一个问题，我们要从list的末尾向前计算乘法，如果到了首位置出现了进位比较难处理，因此最好的方式是将输入的list先反转再计算，最后计算完了再给反转回去。

```
public static List<Integer> mul(List<Integer> listA, int b) {
    Collections.reverse(listA);
    int next = 0;
    List<Integer> listC = new ArrayList<>();
    for (int i = 0; i < listA.size(); i++) {
        next += listA.get(i) * b;
        listC.add(next % 10);
        next /= 10;
    }
    //进位可能超过10
    while (next != 0) {
        listC.add(next % 10);
```

```
        next /= 10;
    }
    Collections.reverse(listC);
    return listC;
}
```

2.4.3 高精度减法

使用数组完成减法的原理与加法的相反，从低位开始，如果不够了，需要借位。另外，减到的结果可能为负数，为此我们可以创建的单独的比较方法，让sub(a，b)方法里只执行a大于b的情况。

```
public static List<Integer> sub(List<Integer> listA, List<Integer> listB) {

    List<Integer> listC = new ArrayList<>();
    int next = 0;
    for (int i = 0; i < listA.size(); i++) {
        next = listA.get(i) - next;
        if (i < listB.size()) {
            next -= listB.get(i);
        }
        listC.add((next + 10) % 10);
        if (next < 0) {
            next = 1;
        } else {
            next = 0;
        }
    }
    return listC;
}

public static boolean cmp(List<Integer> listA, List<Integer> listB) {
    if (listA.size() != listB.size()) {
        return listA.size() > listB.size();
    }
    for (int i = listA.size() - 1; i >= 0; i--) {
        if (listA.get(i) != listB.get(i)) {
            return listA.get(i) > listB.get(i);
        }
    }
    return true;
}
```

对于负数，我们将列表定义为Integer类型，就不能存符号了，一种方式是将列表改成`List<String>`,但是这样每个位置都要先转换再计算，我们这里是在main()方法里简单处理该情况。测试方法：

```
public static void main(String[] args) {
    List listA = new ArrayList<>();
    listA.add(1);listA.add(2);listA.add(3);
```

```
    List listB = new ArrayList<>();
    listB.add(1);listB.add(2);listB.add(4);

    List<Integer> listC = new ArrayList<>();
    if (cmp(listA, listB)) {
        listC = sub(listA, listB);
        System.out.println(listC);
    } else {
        listC = sub(listB, listA);
        System.out.println("-" + listC);
    }
}
```

2.4.4高精度除法

对于除法，我们仍然可以按照小学时学习的方法来逐步计算。我们仍然先将被除数反转一下，然后直接借位计算即可。不过除数大于10时，处理起来非常繁琐，我们这里只考虑一下除数小于10 情况，代码如下：

```
public static List<Integer> div(List<Integer> listA, int b) {
    Collections.reverse(listA);
    List<Integer> listC = new ArrayList<>();
    int next = 0;
    for (int i = listA.size() - 1; i >= 0; i--) {
        next = next * 10 + listA.get(i);
        listC.add(next / b);
        next %= b;
    }
    return listC;
}
```

测试方法：

```
public static void main(String[] args) {
    List lista = new ArrayList<>();
    lista.add(1);
    lista.add(2);
    lista.add(0);
    System.out.println((div(lista, 5)));
}
```

2.5 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。

2.5.1 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：

```
 boolean 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：

```
public boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0;
}
```

2.5.2 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。

```
public boolean isPowerOfThree(int n) {
        if (n <= 0) {
            return false;
        }
        while (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：

```
public boolean isPowerOfThree(int n) {
        return n > 0 && 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?

2.5.3 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。

第一种方法自然还是数学方法一直除，代码如下：

```
boolean 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。

2.6 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。

```
int singleNumber(int[] nums) {
        int single = 0;
        for (int num : nums) {
            single ^= num;
        }
        return single;
    }
```

3.数论问题

3.1 辗转相除法

辗转相除法又叫做欧几里得算法,是公元前 300 年左右的希腊数学家欧几里得在他的著作《几何原本》提出的。最大公约数（greatest common divisor，简写为gcd），是指几个数的共有的因数之中最大的一个，例如 8 和 12 的最大公因数是 4，记作 gcd(8,12)=4。
辗转相除法最重要的规则是，若 r 是 a ÷ b 的余数, 则gcd(a,b) = gcd(b,r)。例如计算 gcd(546, 429)：

```
由于 546=1(429)+117,429=3(117)+78,117=1(78)+39,78=2(39),因此
gcd(546, 429)
=gcd(429, 117)
=gcd(117, 78)
=gcd(78, 39)
=39
```

该规则的证明我们不做过多解释，感兴趣的同学可以看一下https://www.zhihu.com/question/51427771。
我们只看基于该结论如何实现，循环实现代码如下：

```
  int gcd(int a, int b) {// 循环实现
    int k = 0;
    do {
      k = a % b;// 得到余数
      a = b;// 根据辗转相除法,把被除数赋给除数
      b = k;// 余数赋给被除数
    } while (k != 0);
    return a;// 返回被除数
  }
```

3.2 素数和合数

我们看一下素数和合数的问题。素数又称为质数，素数首先要满足大于等于2，并且除了1和它本身之外，不能被任何其他自然数整除。其他数都是合数。比较特殊的是1即非素数，也非合数。2是唯一的同时为偶数和素数的数字。

有了定义，自然第一个问题就是如何判断一个正整数是否为素数。题目要求：给定一个正整数n(n<10^9)，判断它是否为素数。

基本的方式是从2开始依次与n取余测试，看看是否出现n%i==0的情况，如果出现了则说明当前的n能被i整除，所以就不是。理论上一直测试到n-1，假如都不是，那就是素数了。

而事实上不需要测试这么多，只要从2开始遍历一直到n^(1/2)就可以 ，不用执行到n-1。这个是有明确的数学证明的，我们不再赘述，如果不知道请回家问高中老师。所以实现代码就是：

```
boolean isPrime(int num) {
  int max = (int)Math.sqrt(num);
  for (int i = 2; i <= max; i++) {
        if (num % i == 0) {
          return false;
      }
    }
  return true;
    }
```

基于该基础，就可以造题了，例如LeetCode204 给定整数 n ，返回 所有小于非负整数 n 的质数的数量 。

```
示例1：
输入：n = 10
输出：4
解释：小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7 。
```

这个问题几乎就是将上面的代码再套一层就行了，直接遍历：

```
public int countPrimes(int n) {
        int cnt = 0;
        for (int i = 2; i < n; i++) {
            if (isPrime(i)) {
                cnt++;
            }
        }
        return cnt;
}
```

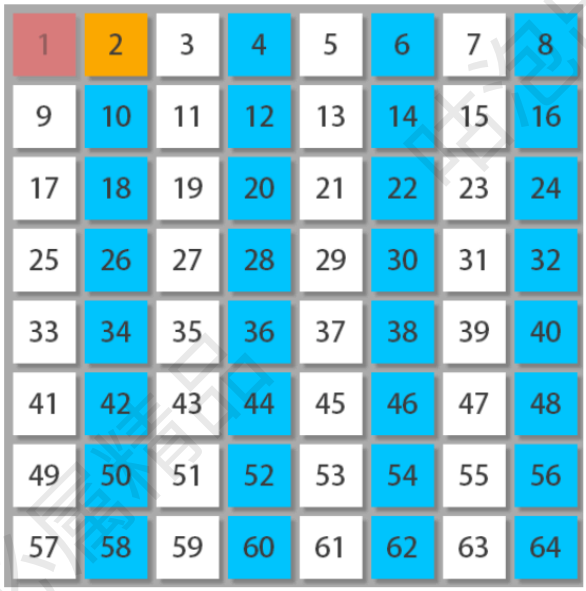
这个方法计算小的数据没有错，但是计算比较大的n仍然会超时，还是性能不够。我们下一节继续研究。

3.3 拓展 埃氏筛

上面LeetCode204 的题找素数的方法虽然能解决问题，但是效率太低，能否有效率更高一些的方法呢?

解决这个题有一个有效的方法，叫做埃氏筛，后来又产生了线性筛， 奇数筛等改进的方法。

基本思想是如果 x是质数，那么大于 x 的 xy的倍数 2x. 3x... 一定不是质数，因此我们可以从这一点入手。如下图所示:



1.选中素数2，然后并排除2的倍数

2.选中素数3，然后排除3的倍数

我们先选中数字2，2是素数，然后将2的倍数全部排除（在数组里将该位置标记为0就行了）。

接着我们选中数字3，3是素数，然后将3的倍数全部排除。

接着我们选择数字5，5是素数，然后将5的倍数全部排除。

接着我们选择 7，11，13一直到n，为什么 4 、6、8 、9 ...不会再选择了呢？因为我们已经在前面的步骤中，将其变成0了。所以实现代码如下：

```
    public int countPrimes(int n) {
        int[] isPrime = new int[n];
        Arrays.fill(isPrime, 1);
        int ans = 0;
        for (int i = 2; i < n; ++i) {
            if (isPrime[i] == 1) {
                ans += 1;
                if ((long) i * i < n) {
                    for (int j = i * i; j < n; j += i) {
                        isPrime[j] = 0;
                    }
                }
            }
        }
        return ans;
    }
```

这个是典型的以空间换时间的算法，这种思想在解决一些问题的时候可以参考，例如下面的丑数这个题。

这个是剑指offer中的题目，我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number），求按从小到大的顺序的第 n 个丑数。

```
示例:
输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
```

所谓一个数m是另一个数n的因子，是指n能被m整除，也就是n%m==0。因此，我们可以逐个判断每个整数是不是丑数。丑数只能被2、3、5整除，也就是说，如果一个数能被2整除，就连续除以2；如果能被3整除，就连续除以3；如果能被5整除，就连续除以5；如果最后得到的是1，那么这个数就是丑数，否则不是。由此，按照顺序依次判断每个整数，并进行计数，就可以找到第N个丑数。这种方法简单直观，但是需要判断每一个整数，不够高效。代码就是：

```
int getUglyNumber(int index) {
        if (index <= 0) {
            return 0;
        }
        int number = 0;
        int uglyFound = 0;
        while (uglyFound < index) {
            ++number;
            if (nthUglyNumber1(number)) {
                ++uglyFound;
            }
        }
```

```
        return number;
    }
 //判断当前的数字是不是丑数
 100 50 25
  boolean nthUglyNumber1(int index) {
        while (index % 2 == 0) {
            index /= 2;
        }
        while (index % 3 == 0) {
            index /= 3;
        }
        while (index % 5 == 0) {
            index /= 5;
        }
        return index == 1 ? true : false;
}
```

为此，我们参考上面埃氏筛的思想，创建数组保存已经找到的丑数，用空间换时间。为了提高效率，我们可以只计算丑数，不在非丑数上浪费时间。根据丑数的定义，丑数应该是另一个丑数乘以2、3或者5的结果（1除外）。因此，我们可以创建一个数组，里面保存的是排好序的丑数，每一个丑数都可以由前面的丑数乘以2、3或者5得到。

这个思路的关键在于如何保证数组里的丑数都是排好序的，也就是每次怎样生成新的丑数。对乘以2而言，肯定存在某个丑数M2，排在它之前的所有丑数乘以2都会小于已有的最大丑数，在它之后的丑数乘以2都会比他大。而对乘以3和5而言，也存在同样的M3和M5，因此只需要维护三个索引值即可判断下一次生成的丑数应该是多少。

```
    public int nthUglyNumber(int index) {
       if(index<1)
            return 0;
        int[] pUglyNumbers=new int[index]; //依次保存第n个丑数
        pUglyNumbers[0]=1; //第一个丑数是1
        int pMultiply2=0,pMultiply3=0,pMultiply5=0;

        for(int i=1;i<index;i++){
            int min=getMin(pUglyNumbers[pMultiply2]*2,pUglyNumbers[pMultiply3]*3,
                           pUglyNumbers[pMultiply5]*5);
            pUglyNumbers[i]=min;
            while(pUglyNumbers[pMultiply2]*2<=min)
                pMultiply2++;
            while(pUglyNumbers[pMultiply3]*3<=min)
                pMultiply3++;
            while(pUglyNumbers[pMultiply5]*5<=min)
                pMultiply5++;
        }
        return pUglyNumbers[index-1];
    }
     public int getMin(int a,int b,int c){
        int min=a>b?b:a;
        return c<min?c:min;
    }
```

4.总结

本章主要介绍了基本的数学问题，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。

除了上面的题目，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。