数值计算篇

数值计算在工业界是非常实用而且常见的具体问题，所以在面试中出现频率非常高，几乎可以说是必考题目。

LeetCode 中关于数值运算的有以下题目：

1. Palindrome Number
2. Reverse Integer
3. Sqrt(x)
4. Pow(x, n)
5. Divide Two Integers
6. Max Points on a Line

在LeetCode中，关于数值运算的题目分为三种类型，下面将进行一一讲解。

第一种类型是最简单的，就是对整数进行直接操作，一般来说就是逐位操作，比如反转，比较等。LeetCode中这类题目有Palindrome Number和Reverse Integer。这类题目通常思路很清晰，要注意的点就是对于边界情况的考虑，对于数值而言，主要问题是对于越界情况的考虑。实际上越界问题是贯穿于所有数值计算题目的常见问题，下面大多问题都会强调这点。

在Palindrome Number中因为只是进行判断，并不需要修改数字，所以没有越界问题。思路比较简单，就是每次取出最高位和最低位进行比较，直到相遇或者出现违背条件（也就是不相等）即可返回。对于Reverse Integer因为需要对数字进行反转，所以需要注意反转后的数字可能会越界。对于越界一般都是两种处理方法，一种是返回最大（或者最小）数字，一种则是抛出异常，这个可以跟面试官讨论，一般来说，面试只要简单的返回最大最小或者dummy数字就可以了，但是处理和检查这种corner case（也就是越界）的想法一定要有和跟面试官讨论。

1. Palindrome Number，这道题跟Reverse Integer差不多，也是考查对整数的操作，相对来说可能还更加简单，因为数字不会变化，所以没有越界的问题。基本思路是每次取第一位和最后一位，如果不相同则返回false，否则继续直到位数为0。这个题和Longest Palindromic Substring也很接近，只是处理的数据结构有所不同。

代码如下：

```
public boolean isPalindrome(int x) {
    if(x < 0){
        return false;
    }
    int digit = 1;
    while(digit <= x / 10){
        digit *= 10;
    }
    while(x > 0){
        if(x / digit != x % 10){
            return false;
        }
        x %= digit;
        x /= 10;
        digit /= 100;
    }
    return true;
}
```

做题时的感悟：

1. 默认负数不是回文数。

2. 如何取一个数的最高位是一个非常常用的方法,因为很多数字处理题目中，都要求我们求到最高位再做相应的比较或者处理。

```
int digit = 1;
while(digit <= x / 10){
    digit *= 10;
}
```

x / digit 即为最高位数。

我们也可以从高位到低位一位一位的取出相应位数字。

2. Reverse Integer，这道题思路非常简单，就是按照数字位反转过来就可以，基本数字操作。但是这种题的考察重点并不在于问题本身，越是简单的题目越要注意细节，一般来说整数的处理问题要注意两点，一点是符号，另一点是整数越界问题。

代码为了后面方便处理，先将数字转为正数。注意 Integer.MIN_VALUE 的绝对值是比 Integer.MAX_VALUE 大 1 的，所以当取绝对值的时候会越界，经常要单独处理。如果不先转为正数也可以，只是在后面要对符号进行一下判断。这种题目考察的就是数字的基本处理，面试的时候尽量不能错，而且对于 corner case 要尽量进行考虑，一般来说都是面试的第一道门槛。

代码如下：

```
public int reverse(int x) {
    if(x == Integer.MIN_VALUE){
        return 0;
    }
    int num = Math.abs(x);
    int res = 0;
    while(num > 0){
        int digit = num % 10;
        num /= 10;
        if(res > (Integer.MAX_VALUE - digit) / 10){
            return 0;
        }
        res = res * 10 + digit;
    }
    return x > 0 ? res : -res;
}
```

做题时的感悟：

1．凡是要对整数求绝对值处理的，都要考虑Integer.MIN_VALUE比Integer.MAX_VALUE大1的问题，所以要对这种情况进行特殊处理。

2．整数处理过程中，越界是最重要的考虑点。只要有乘或者使数变大的操作，基本都要进行越界判断。而且在写判断语句的时候，尽量使用除法，这样可以防止越界。

第二种题型是算术运算的题目，比如乘除法，阶乘，开方等，LeetCode中这类题目有Sqrt(x)，Pow(x, n)和Divide Two Integers。这种题目有时候看似复杂，其实还是有几个比较通用的解法的，下面主要介绍三种方法：

（1）二分法。二分法是数值计算中很常用和易懂的方法。基本思路是对于所求运算进行对半切割，有时是排除一半，有时则是得到可重复使用的历史数据。Sqrt(x)就是属于每次排除一半的类型，对于要求的开方数字进行猜测，如果大于目标，则切去大的一半，否则切去小的一半，原理跟二分查找是一样的。Pow(x, n)则是属于重复利用数据的类型，因为x的n次方实际上是两个x的n/2次方相乘，所以我们只需要递归一次求出当前x的当前指数的1/2次方，然后两个相乘就可以最后结果。二分法很明显都是每次解决一半，所以时间复杂度通常是O（logn）量级的。

（2）牛顿法。这种方法可以说主要是数学方法，不了解原理的朋友可以先看看牛顿法-维基百科。Sqrt(x)就非常适合用牛顿法来解决，因为它的递推式中的项都比较简单。Pow(x, n)当然原理上也可以用牛顿法解答，但是因为他的递推式中有项是进行x的开n次方的，这个计算代价也是相当大的，如果为了求x的n次方而去每一步求开n次方就没有太大实际意义了，所以对于这种题我们一般不用牛顿法。

（3）位移法。这种方法主要基于任何一个整数可以表示成以 2 的幂为底的一组基的线性组合，对一个整数进行位数次迭代求解，因为复杂度是位数的数量，所以也跟二分法一样是O（logn）量级的。Pow(x, n)和 Divide Two Integers 就是比较典型可以用这种方法解决的题目。对于 Pow(x, n)可以把 n 分解成位，每次左移恰好是当前数的平方，所以进行位数次迭代后即可以得到结果，代码中很大的篇幅都是在处理越界问题，而关于逐位迭代代码却很简短。Divide Two Integers 同样把结果分解成位，每次对除数进行位移并且减去对应的除数来确定每一位上的结果。这种方法理解起来没有二分法那么直观，需要消化一下。

3. Sqrt(x)，和Divide Two Integers不同，这道题一般采用数值中经常用的另一种方法：二分法。基本思路是跟二分查找类似，要求知道结果的范围，取定左界和右界，然后每次砍掉不满足条件的一半，直到左界和右界相遇。时间复杂度是O(logx)，空间复杂度是O(1)。其实这个题目还有另一种方法，称为牛顿法。一般牛顿法是用数值方法来解一个 f(y)=0 的方程（为什么变量在这里用 y 是因为我们要求的开方是 x，避免歧义）。对于这个问题我们构造 f(y)=y^2-x，其中 x 是我们要求平方根的数，那么当 f(y)=0 时，即 y^2-x=0,所以

y=sqrt(x)，即是我们要求的平方根。f(y)的导数 f'(y)=2*y，根据牛顿法的迭代公式我们可以得到 y_(n+1)=y_n-f(n)/f'(n)=(y_n+x/y_n)/2。最后根据以上公式来迭代解以上方程。

做题时的感悟:

1. 可以把r设置成x / 2 + 1，这样可以避免对1的讨论，否则要在前面讨论当x == 0 | | x == 1时，直接return x。

2. 注意这里找到m的条件并不一定是要相等，而是在一个范围内即可。我们要灵活的改变二分查找的判断条件来应对不同的题。

```
if(m <= x / m && x / (m + 1) < (m + 1)){
    return m;
}
```

这里用除来比较可以防止int型越界。

3. 因为牛顿法在数值计算中很常用，所以记一下它的公式：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

核心代码为：

```
double lastY = 0;
double y = 1;
while(y != lastY){
    lastY = y;
    y = (y + x / y) / 2;
}
return (int)y;
```

注意这里起始的 y 可以取任何值，因为最后都会收敛到同一值。

4. Pow(x, n)，这道题是一道数值计算的题目，因为指数是可以使结果变大的运算，所以要注意越界的问题。如同Sqrt(x)这道题中提到的，一般来说数值计算的题目可以用两种方法来解，一种是以2为基进行位处理的方法，另一种是用二分法。这道题这两种方法都可以解。第一种方法在Divide Two Integers使用过，就是把n看成是以2为基的位构成的，因此每一位是对应x的一个幂数，然后迭代直到n到最高位。比如说第一位对应x，第二位对应x*x，第

三位对应x^4,...,第k位对应x^(2^(k-1)),可以看出后面一位对应的数等于前面一位对应数的平方，所以可以进行迭代。因为迭代次数等于n的位数，所以算法的时间复杂度是O(logn)。
接下来我们介绍二分法的解法，如同我们在Sqrt(x)的方法。不过这道题用递归来解比较容易理解，把x的n次方划分成两个x的n/2次方相乘，然后递归求解子问题，结束条件是n为0返回1。因为是对n进行二分，算法复杂度和上面方法一样，也是O(logn)。
以上代码比较简洁，不过这里有个问题是没有做越界的判断，因为这里没有统一符号，所以越界判断分的情况比较多，不过具体也就是在做乘除法之前判断这些值会不会越界。不过实际应用中健壮性还是比较重要的，而且递归毕竟会占用递归栈的空间，所以我可能更推荐第一种解法。

做题时的感悟:

1. 在数值计算的题目中，对越界的判断和特殊情况的处理是至关重要的，要分类而有序的进行处理避免混乱。首先，针对指数运算特有的情况，当n == 0时，直接返回1.0即可。
2. 当n < 0时，意味着我们要取x的倒数来进行运算。在做取倒数的运算之前，我们要进行判断是否可能因为x过于接近0而使取的倒数越界。
x >= 1.0 / Double.MAX_VALUE || x <= 1.0 / Double.MIN_VALUE
3. 只要是对int型做取绝对值的操作，就要先讨论该值是否会为Integer.MIN_VALUE，因为Integer.MIN_VALUE比Integer.MAX_MALUE大1，所以取绝对值会越界。所以该题也要对n进行判断。
4. 一般来说求的时候都先转成正数，这样可以避免需要双向判断（就是根据符号做两种判断）在将x取绝对值前，要先判断最后结果的符号。
5. 在求指数的过程中，我们也得检查有没有越界。
6. 判断某一位是不是有1，直接(n & 1) == 1即可，注意括号一定要有。然后，n >>= 1。

5. Divide Two Integers，对于整数处理的问题，比较重要的注意点在于符号和处理越界的问题。我们使用位运算。我们知道任何一个整数可以表示成以2的幂为底的一组基的线性组合，即num=a_0*2^0+a_1*2^1+a_2*2^2+...+a_n*2^n。基于以上这个公式以及左移一位相当于乘以2，我们先让除数左移直到大于被除数之前得到一个最大的基。然后接下来我们每次尝试减去这个基，如果可以则结果增加加2^k,然后基继续右移迭代，直到基为0为止。因为这个方法的迭代次数是按2的幂直到超过结果，所以时间复杂度为O(logn)。

做题时的感悟:

1. 数值处理的题目，要考虑的情况比较多。单纯的去想容易混乱。一个好的办法就是仅想下一个操作可能出现的问题，然后进行判断和处理。开始，讨论通用情况，比如divisor == 0，直接返回Integer.MAX_VALUE即可。接下来，为了避免双向讨论，我们需要把dividend和divisor都转换为正数。转换正数需要取绝对值，就要处理Integer.MIN_VALUE比Integer.MAX_VALUE大1，取绝对值越界的问题。如果dividend == Integer.MIN_VALUE，res = 1; dividend += Math.abs(divisor); 接下来判断如果divisor == Integer.MIN_VALUE，直接return res；因为如果dividend == Integer.MIN_VALUE，res已经为1，如果dividend != Integer.MIN_VALUE，res一定为0，因为是int除法。并且，在将2数转换为正数前，要确定最终结果的符号，大神巧妙的用了位运算，boolean isNeg = ((dividend ^ divisor) >>> 31) == 1；接下来就为核心运算部分了。

2. 核心运算部分代码如下：

```
int digit = 0;
while(divisor <= (dividend >> 1)){
    divisor <<= 1;
    digit++;
}
while(digit >= 0){
    if(dividend >= divisor){
        dividend -= divisor;
        res += 1 << digit;
    }
    digit--;
    divisor >>= 1;
}
return isNeg ? -res : res;
```

开始类似于取最高位那个过程，找到大于dividend一半的最小2的基，并用digit记录移动位数。计算循环注意digit >= 0，否则会漏掉最后1位产生错误结果。

第三种题目是解析几何的题目，一般来说解析几何题目的模型都比较复杂，而且实现细节比较多，在面试中并不常见，LeetCode中也只有Max Points on a Line是属于这种题型。这种题目没有什么通法，主要就是要理清数学和几何模型，比如Max Points on a Line中主要是理解判断点在直线的判断公式，然后进行迭代实现。实现细节还是比较多的，需要对一些边界情况仔细考虑。

6. Max Points on a Line，每次迭代以某一个点为基准，看后面每一个点跟它构成的直线，维护一个HashMap，key是跟这个点构成直线的斜率的值，而value就是该斜率对应的点的数量，计算它的斜率，如果已经存在，那么就多添加一个点，否则创建新的key。这里只需要考虑斜率而不用考虑截距是因为所有点都是对应于一个参考点构成的直线，只要斜率相同就必然在同一直线上。最后取map中最大的值，就是通过这个点的所有直线上最多的点的数量。对于每一个点都做一次这种计算，并且后面的点不需要看扫描过的点的情况了，因为如果这条直线是包含最多点的直线并且包含前面的点，那么前面的点肯定统计过这条直线了。因此算法总共需要两层循环，外层进行点的迭代，内层扫描剩下的点进行统计，时间复杂度是O（n^2），空间复杂度是哈希表的大小，也就是O（n），比起Brute Force做法这里用哈希表空间省去了一个量级的时间复杂度。

代码如下：

```
public int maxPoints(Point[] points) {
    if(points == null || points.length == 0){
        return 0;
    }
    int max = 1;
    double ratio = 0.0;
    for(int i = 0; i < points.length - 1; i++){
        HashMap<Double, Integer> map = new HashMap<Double, Integer>();
        int numOfSame = 0;
        int localMax = 1;
        for(int j = i + 1; j < points.length; j++){
            if(points[i].x == points[j].x && points[i].y == points[j].y){
                numOfSame++;
                continue;
            } else if(points[i].x == points[j].x){
                ratio = (double)Integer.MAX_VALUE;
            } else if(points[i].y == points[j].y){
                ratio = 0.0;
```

```
                } else {
                    ratio = (double)(points[j].y - points[i].y) / (double)(points[j].x -
points[i].x);
                }
                int num;
                if(map.containsKey(ratio)){
                    num = map.get(ratio) + 1;
                } else {
                    num = 2;
                }
                map.put(ratio, num);
            }
            for(int value : map.values()){
                localMax = Math.max(localMax, value);
            }
            localMax += numOfSame;
            max = Math.max(localMax, max);
        }
        return max;
    }
```

做题时的感悟:

1. 这里max和localMax的初值都要从1开始，因为最少有1个点即这个点本身。否则遇到只有1个点的集合会出问题。

2. 每次循环中，要新增一个变量sameNum对与该点相同的点单独进行统计，因为他们的ratio值无法计算。

```
                if(points[i].x == points[j].x && points[i].y == points[j].y){
                    sameNum++;
                    continue;
                }
```

3. 之后就是对特殊 ratio 进行处理，计算出 ratio. map 中有，则 value+1，没有新建该 ratio 为 key，value 为 2. 然后从 map 中找 localMax，再从 max 和 localMax + sameNum 中，得到全局 max。