高精度篇

我们常见的一些基本的数据结构比如整型 int 或者浮点型 float 由于位数过多无法用内置类型存储，这时候我们就需要自己实现高精度的数据类型来进行存储和运算。这种问题在实际产品中还是比较实用的，所以相对来说也是面试中的常客。

LeetCode 中关于高精度的题目有以下几道：

1. Add Binary
2. Add Two Numbers
3. Plus One
4. Multiply Strings

Add Binary 和 Add Two Numbers 是同一类型的题目，都是高精度中的加法运算，只是一个是二进制的，一个是十进制的，其实进制是无所谓的，代码基本可以统一起来用一种思路来实现。思路也很简单，就是从低位开始相加，一直维护进位就可以了。

1. Add Binary，这道题跟Add Two Numbers很类似，代码结构很接近。从低位开始，一直相加并且维护进位。和Add Two Numbers的区别是这个题目低位在后面，所以要从string的尾部往前加。时间复杂度是O(m+n)，m和n分别是两个字符串的长度，空间复杂度是结果的长度O(max(m,n))。最后有一个小细节要注意一下，就是我们维护的res是把低位放在前面，为了满足结果最后要进行一次reverse。

代码如下：

```
public String addBinary(String a, String b) {
    if(a == null || a.length() == 0){
        return b;
    }
    if(b == null || b.length() == 0){
        return a;
    }
    StringBuilder res = new StringBuilder();
    int carry = 0;
    int idxA = a.length() - 1;
    int idxB = b.length() - 1;
```

```
        while(idxA >= 0 && idxB >= 0){
            int digit = a.charAt(idxA) + b.charAt(idxB) - ('0' << 1) + carry;
            res.append(digit & 1);
            carry = digit >> 1;
            idxA--;
            idxB--;
        }
        while(idxA >= 0){
            int digit = a.charAt(idxA) - '0' + carry;
            res.append(digit & 1);
            carry = digit >> 1;
            idxA--;
        }
        while(idxB >= 0){
            int digit = b.charAt(idxB) - '0' + carry;
            res.append(digit & 1);
            carry = digit >> 1;
            idxB--;
        }
        if(carry == 1){
            res.append(1);
        }
        return res.reverse().toString();
    }
```

做题时的感悟:

1. 由于从字符串中取出的是字符数字，所以应当减去字符0的数值:

int value = a.charAt(idxA) + b.charAt(idxB) - ('0' << 1) + carry;

2. 注意最后对carry的处理，如果最后carry为1，还要在结尾加上1.

2. Add Two Numbers，这道题思路很明确，就是按照位数读下去，维护当前位和进位，时间复杂度是O(n),空间复杂度是O(1).实现中注意维护进位，陷阱的话记住最后还要判一下有没有进位，如果有再生成一位。这道题还是有一些扩展的，比如这个其实是BigInteger的相加，数据结果不一定要用链表，也可以是数组，面试中可能两种都会问而且实现。然后接下来可以考一些OO设计的东西，比如说如果这是一个类应该怎么实现，也就是把数组或者链表作为成为成员变量，再把这些操作作为成员函数，进一步的问题可能是如何设计

constructor，这个问题除了基本的还得要有对内置类型比如int，long进行处理的constructor，类似于BigInteger(int num)，BigInteger(int long)。

代码如下：

```
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode cur = dummy;
        int carry = 0;
        while(l1 != null && l2 != null){
            int digit = l1.val + l2.val + carry;
            ListNode node = new ListNode(digit % 10);
            cur.next = node;
            cur = cur.next;
            carry = digit / 10;
            l1 = l1.next;
            l2 = l2.next;
        }
        while(l1 != null){
            int digit = l1.val + carry;
            ListNode node = new ListNode(digit % 10);
            cur.next = node;
            cur = cur.next;
            carry = digit / 10;
            l1 = l1.next;
        }
        while(l2 != null){
            int digit = l2.val + carry;
            ListNode node = new ListNode(digit % 10);
            cur.next = node;
            cur = cur.next;
            carry = digit / 10;
            l2 = l2.next;
        }
        if(carry != 0){
            ListNode node = new ListNode(carry);
            cur.next = node;
        }
        return dummy.next;
}
```

做题时的感悟:

1. 开始可以对l1, l2其中是否有空链表进行判断，这样可以避免后面剩一个链表时对表头情况分情况讨论。

2. 依然要注意最后 carry 不为 0 的时候，要新建一个结点存 carry 接上。

Plus One 也是一道常见的题目，他其实就是实现 C++中++的运算符，因为只需要+1，所以其实比上面的题目更加简单。这道题的小陷阱就是它是用数组从高位到低位进行存储的，所以如果出现进位，那么需要重新分配空间，并给最高位赋 1，其他位赋 0 即可。这里恰好引入一个点，就是高精度存储应该低位到高位存储还是反过来好，这也是面试中可能问到的问题。

3. Plus One，这是一道比较简单的题目，对一个数组进行加一操作。但是可不要小看这个题，这个题被称为“Google最喜欢的题”，因为在google面试中出现的频率非常高。思路是维护一个进位，对每一位进行加一，然后判断进位，如果有继续到下一位，否则就可以返回了，因为前面不需要计算了。有一个小细节就是如果到了最高位进位仍然存在，那么我们必须重新new一个数组，然后把第一个为赋成1（因为只是加一操作，其余位一定是0，否则不会进最高位）。因为只需要一次扫描，所以算法复杂度是O(n)，n是数组的长度。而空间上，一般情况是O(1)，但是如果数是全9，那么是最坏情况，需要O(n)的额外空间。

代码如下:

```
public int[] plusOne(int[] digits) {
    if(digits == null || digits.length == 0){
        return null;
    }
    int carry = 1;
    for(int i = digits.length - 1; i >= 0; i--){
        int value = digits[i] + carry;
        digits[i] = value % 10;
        carry = value / 10;
        if(carry == 0){
            break;
        }
    }
    if(carry != 0){
        digits = new int[digits.length + 1];
```

```
            digits[0] = 1;
        }
        return digits;
    }
```

做题时的感悟：

1．因为在计算carry和digit[i]的时候，要互相用到对方的值，所以不能在计算完其中一个时直接赋值，不然另一个就会用赋值后的来计算出错误的结果。所以我们应该用一个临时变量记录下当前的digit，然后计算carry，然后再把digit赋值给digit[i]。

2．因为只是加1，所以当carry == 0时，即可以结束计算了。如果到最后carry还不为0，这时我们要新建一个数组，并将res[0] = 1即可。

4. Multiply Strings，这道题属于数值操作的题目，其实更多地是考察乘法运算的本质。基本思路是和加法运算还是近似的，只是进位和结果长度复杂一些。我们仍然是从低位到高位对每一位进行计算，假设第一个数长度是n，第二个数长度是m，我们知道结果长度为m+n或者m+n-1（没有进位的情况）。对于某一位i，要计算这个位上的数字，我们需要对所有能组合出这一位结果的位进行乘法，即第1位和第i-1位，第2位和第i-2位，… ，然后累加起来，最后我们取个位上的数值，然后剩下的作为进位放到下一轮循环中。这个算法两层循环，每层循环次数是O(m+n)，所以时间复杂度是O((m+n)^2)。算法中不需要额外空间，只需要维护一个进位变量即可，所以空间复杂度是O(1)。

实现中有两个小细节，一个是循环中最后有一个if判断，其实就是看最高一位是不是0（最高第二位不可能是0，除非两个源字符串最高位带有0），如果是就不需要加入字符串中了。另一个小问题是我们是由低位到高位放入结果串的，所以最后要进行一次reverse，因为是一个O(m+n)的操作，不会影响算法复杂度。

这道题其实还有更加优的做法，就是用十大最牛的算法之一--Fast Fourier transform(FFT)。使用FFT可以在O(nlogn)时间内求出多项式的乘法，只需要知道这个思路和基本工作原理就可以了。

代码如下：

```
    public String multiply(String num1, String num2) {
        if(num1 == null || num2 == null || num1.length() == 0 || num2.length() == 0){
            return "";
        }
```

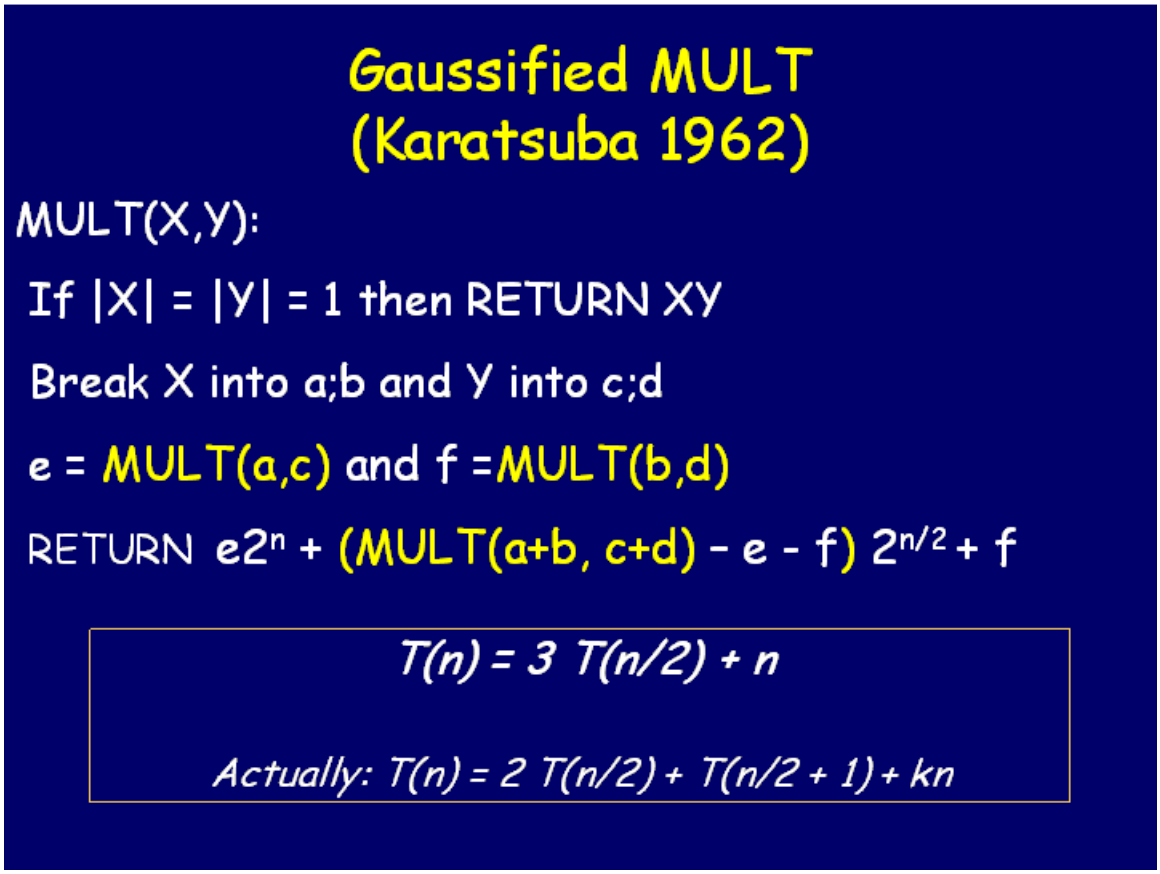
```
        if(num1.charAt(0) == '0'){
            return "0";
        }
        if(num2.charAt(0) == '0'){
            return "0";
        }
        StringBuilder res = new StringBuilder();
        int num = 0;
        for(int i = num1.length() + num2.length(); i > 0; i--){
            for(int j = Math.min(i - 1, num1.length()); j > 0; j--){
                if(i - j <= num2.length()){
                    num += (int)(num1.charAt(j - 1) - '0') * (int)(num2.charAt(i - j - 1) - '0');
                }
            }
            if(i != 1 || num > 0){
                res.append(num % 10);
            }
            num /= 10;
        }
        return res.reverse().toString();
    }
```

做题时的感悟:

1. 这里第一层循环的i代表的是结果的第i位，j代表的是取num1的第几个字符，而i - j代表的是取num2的第几个字符。如果i - 1 < num1.length()时，即j取极限num1.length()时，i - j < 1，说明num2只能取第0个以前的字符，说明num2长度不够了，j只能取i - 1。内层循环的i - j <= num2.length()判断的是,是否超出第二个字符串的长度,如果超出则跳过即可。
2. 最后注意2个细节，首先是最高位是否为0，不为0才添加，为0直接跳过。其次，我们是从最低为开始append的，结果是反的，所以最后需要做一次reverse()操作。
3. 我们可以通过Karatsuba方法把时间复杂度降至n的1.58次方。

参考资料:

http://www.cs.cmu.edu/afs/cs/academic/class/15251-s04/Site/Materials/Lectures/Lecture16/lecture16.html



总结：

虽然题目不多，但是这类题目的出现率却是非常高的，主要原因倒不是这种题目本身有很多的考点，而是它们特别好扩展，基本上来说问到这种题目，首先是考察一下代码能力，一般来说都是这种加减乘除的运算，接下来一定会是关于数据结构（或者说面向对象）的设计。这些题目的本身都是为高精度 BigInteger 服务的，面试官会问一些关于这个数据结构设计的问题，比如说如果让你来设计这个类，用什么数据结构来存（比如数组还是链表，各有什么利弊），需要哪些接口（构造函数，加减乘除运算等等），还有比如要设计构造函数，需要什么接口的构造函数（这里赋值构造函数，赋值运算符这些肯定是需要的，但是要注意必须提供对于常规类型比如 int，long 这些的接口，一个好的高精度类肯定是要对比它更弱的数据结构进行兼容的）。