信息学奥赛笔记13

高精度加法复习，高精度减法，高精度乘法。

本节课新知识——高精度减法

$$\begin{array}{r} 123 \\ -90 \\ \hline 33 \end{array}$$

这里展示的是一个减法的竖式运算，同学们可以发现，相较于加法的竖式，减法的操作是，我们从第一个数减第二个数。如果够减，直接减去，写出结果。那如果第一个数比第二个数小呢？比如$4-6$这个形式，我们需要给第一个数a 先$+10$ 再去减去，相当于问高位借了一位，那从高位再做运算时就需要预先先减少一个。

在加法运算中，当前结果大于10，进位为1，下次计算需要$+1$

在减法运算中，当前结果小于0，借位为1，下次计算需要-1

所以像加法一样，我们需要一个x来记录是否借位了，如果有借位x为1，下次-1，如果$x==0$，下次-0。

那么，既然我们是逐位做减法的，同学们可以设想一下这个例子：

$9999-9998$，根据逐位减法，结果应该是多少呢？

应该是0001，对于一个数，前面的占位0，我们称之它为**前导0**。那么减法还需要考虑到前导0的情况。

由于我们在保存答案时，像加法一样是倒过来做的，所以保存下来的答案就是1000，我们可以通过弹出末尾的0的形式来去除前导0。

同学们还记得在学习动态数组的时候，是怎么样在末尾添加数字的吗？

```
v.push_back(val);
```

那今天老师告诉大家，这个push_back，不仅仅适用于vector，还适用于string。

那么，既然可以从末尾添加，那也可以从末尾删除吧。

```
v.pop_back();
```

在英文中，push是推，把数字推到末尾，就是添加。

pop的意思是爆裂，最后一位爆炸了，所以就弹出了，删除了，没了。

```
v.back();
这个函数可以获取数组或者字符串的最后一个元素。
例如：
vector<int> v = {1, 2, 3, 4, 5};
int b = v.back();  //b = 5
v.pop_back(); // v = {1, 2, 3, 4}
int c = v.back();  //c = 4
```

那我们可以通过把$back()$函数和$pop_back()$函数连用的形式来去除末尾的0

具体操作如下

```
while (ans.back() == '0') {
    ans.pop_back();
}
```

那么同学们可以进一步思考一下，假如是这种情况呢？

$9999-9999=0000$

那么这句代码会发生啥？

第一个问题：所有的0全被去掉了

第二个问题：如果字符串成空的了，那么$ans.back()$是不是会导致越界？

所以我们要规定一下，如果ans的长度还剩1位，那就直接退出循环了。

```
while (ans.size() > 1 && ans.back() == '0') {
    ans.pop_back();
}
```

前导0是解决了，还有一个问题，如果是$4-6=-2$呢

我们不妨设被减数为a，减数为b，差为c

所以$c=a-b=-(b-a)$

那么也就是说如果$a-b$是个负数，那么我们可以把符号先提出来，然后结果就是$b-a$

那该如何判断被减数比减数要小呢？

那么大家可以想想，如果$a.size()<b.size()$是不是就说明被减数的长度更小，位数更少，一定更小。

那如果$a.size()==b.size()$呢？

是不是就不太好比了。

这里我们学习了一个字符串的**字典序**

我们查英文字典的时候，要先找到这个单词的首字母，然后找第二个字母的位置……

那么计算机内部是可以对字符串进行排序的，它是根据字典序来比较的

两个字符串先比较第一位，如果相等比较第二位……

那么大家可以想想

11111 和 9计算机会认为哪一个更大？

因为1 比 9小，所以9更大。

所以对于两个字符串长度不同的时候，我们是不能直接用字典序比较字符串大小的。

但是长度相同的时候就不一样了，逐位比较，从高位比较，那么一定是大的那个数，最终反馈的字符串结果就更大

我们就可以利用这个方法，假如被减数a小于减数b，我们就直接交换a和b，并且添加一个负号。

至此，我们就把高精度减法已经全部分析完了，接下来请同学们看一下课堂上讲的完整代码

高精度减法的课堂代码

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
string sub(string s1, string s2) {
    string ans;
    bool flag = false; //标识，记录是否为负数
    if((s1.size() < s2.size()) || (s1.size() == s2.size() && s1 < s2)) { //如果被减数短于减数 或者 他俩一样长但是被减数更小
        swap(s1,s2);
        flag = true; //负数标记为true
    }
    int n1 = s1.size() - 1, n2 = s2.size() - 1, x = 0;
    while (n1 >= 0 || n2 >= 0) {
        int a = 0, b = 0;
        if(n1 >= 0) {
            a = s1[n1--] - '0'; //和加法一样，把要减的两个数摘出来
        }
        if(n2 >= 0) {
            b=s2[n2--] - '0';
        }
        int num = a - b - x;    //结果 = a - b - 借位x
        if(num < 0) {  //如果都减成负的了，说明需要借位
            num += 10; //当前的答案 + 10
            x = 1;     //借位数设置为1
        } else {
            x = 0;
        }
        ans.push_back(num + '0');  //把当前的结果插进答案，注意需要整数转字符
    }
    while (ans.size() > 1 && ans.back() == '0') {   //消除前导0
        ans.pop_back();
    }
    if (flag == true) {   //如果是负数，在末尾添加负号
        ans.push_back('-');
    }
    reverse(ans.begin(),ans.end());  //答案翻转
    return ans;
}
int main() {
    string s1, s2;
    cin >> s1 >> s2;
    cout << sub(s1, s2) << endl;
    return 0;
}
```

又到了大家最喜欢的优化环节

以下内容是课堂上没讲的，请同学们认真观看并自己学习！

被减数的数是a ,减数的数是b，借位数为x，所以

结果$num = a - b - x$

假如我们把结果直接先补上借位呢？不管为不为负数，都借位，那同学们可以想一想

$6 - 4 = 2$ 会变成$6 - 4 + 10 = 12$

$4 - 6 = -2$，借位为$4 - 6 + 10 = 8$

那就算借不借位，我们是不是都可以把结果先+10再对10取余。

所以结果$num = (a - b + 10 - x)\%10$

那么大家认为借位符号呢？应该是多少？

是不是，结果$num \geq 10$说明不需要借位，所以借位数为0，否则说明加了10，结果还小于10，借位了，借位数要变成1。

为了节省变量，我们可以先把结果暂时保存进借位数里，再利用借位数的结果对他自身处理。也就是$x = a - b + 10 - x$

所以是不是可以写成

```
x = a - b + 10 - x;
int num = x % 10;
if (x >= 10) {
    x = 0;
} else {
    x = 1;
}
```

再次用到们上次学习到的**三目运算符**，代码可以写成

```
x = a - b + 10 - x;
int num = x % 10;
x = x >= 10 ? 0 : 1;
```

也可以写成

```
x = a - b + 10 - x;
ans.push_back(x % 10 + '0');
x = x >= 10 ? 0 : 1;
```

这里老师再交给大家一个知识，**字符串的相加**

字符串$a =$"abc"， 字符串$b =$"def"

所以$stringc = a + b$，这里的'+'代表的是拼接，所以$c =$"$abcdef$"

那么，我们不就是把最后一个数拼接进ans的吗？

所以这里的代码可以进一步优化写成

```
x = a - b + 10 - x;
ans += x % 10 + '0';
x = x >= 10 ? 0 : 1;
```

进一步地，如果是负数的情况，我还有必要再程序的最后再添加负号吗？

我可不可以在判断出这是负数的时候直接输出负号呢？

当然可以！

再把程序当中其他部分写成三目运算符，最终代码优化如下：

以下代码需要同学们熟练背诵

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
string sub(string s1, string s2) {
    string ans;
    if(s1.size() < s2.size() || s1.size() == s2.size() && s1 < s2) {
        swap(s1, s2);
        cout << '-';  //判断为负数，立刻输出负号并交换被减数与减数
    }
    int n1 = s1.size() - 1, n2 = s2.size() - 1, x = 0;
    while (n1 >= 0 || n2 >= 0) {
        int a = n1 >= 0 ? s1[n1--] - '0' : 0;
        int b = n2 >= 0 ? s1[n2--] - '0' : 0;
        x = a - b + 10 - x;  //把a - b + 10 - x的结果保存进x中
        ans += x % 10 + '0';  //答案添加一位x % 10并转成字符
        x = x >= 10 ? 0 : 1;  //根据x的值的情况来写进位情况。
    }
    while (ans.size() > 1 && ans.back() == '0') {
        ans.pop_back();
    }
    reverse(ans.begin(),ans.end());
    return ans;
}
int main() {
    string s1, s2;
    cin >> s1 >> s2;
    cout << sub(s1, s2) << endl;
    return 0;
}
```

高精度乘法

乘法在竖式上的规则和加减都不一样的地方是我们需要用到一个乘数很多遍。

比如说123×456 我们要利用到123×6 、 123×5、123×4

所以一层循环根本无法搞定。像加法那样，我们可以先把两个数全部翻转回来

变成654×321 。那么，列成竖式后结果大体上是这样的

```
   654
  x321
_________
   837
    516
     294
_________
   88065
```

再将结果反转就是56088

那我们可以观察一下，这是怎么乘出来的

837是怎么来的，是不是6分别去乘321从左往右得到的结果

516呢？用5去乘321。那么，我们的第一层循环就应该代表第一个乘数的位数，第二层循环用来表示第二个乘数的位数。

那么大家还能发现一个问题，乘法的和是错位相加，该怎么办？

我们是不是可以把结果先堆到一起，然后像加法进位那样来解决呢

在这里就是

```
8  3  7
   5  1  6
+     2  9  4
_____________
8  8  10 15 4
```

然后我们对[8，8，10，15，4]再进行进位操作

[8，8，10，15，4]

[8，8，0，16，4]

[8，8，0，6，5]

所以在这里我们可以先把乘法变成int类型的动态数组来做。

那为了给结果预留出位置，所以我们需要开辟的空间应该为多大呢？

大家可以思考一下$999\times999=998001$ 那它是不是可以看成结果比1000×1000要小一点

$1000\times1000=1000000$是个7位数，所以两个999相乘结果就应该是6位数，大体可以看成是两个乘数的数位之和

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
string mul(string s1, string s2) {
    reverse(s1.begin(), s1.end());
    reverse(s2.begin(), s2.end()); //先翻转
    vector<int> s3(s1.size() + s2.size(), 0); //储存结果的变量需要开辟s1.size()
+ s2.size()
    for (int i = 0;i < s1.size(); i++) {
```

```
        for (int j = 0; j < s2.size(); j++) {
            int num = (s1[i] - '0') * (s2[j] - '0');  //计算结果为a * b
            s3[i + j] += num;  //将结果加到第i + j 位，乘法需要移位，不是直接把结果放到第i位的
        }
    }
    int x = 0;  //进位数
    string ans; //保存答案字符串
    for (int i = 0; i < s3.size(); i++) {
        s3[i] += x;
        x = s3[i] / 10;
        s3[i] %= 10;
        ans += s3[i]+'0');
    }
    while (ans.size() > 1 && ans.back() == '0') {  //如果11111 * 0 = 00000，我们要去除前导0
        ans.pop_back();
    }
    reverse(ans.begin(),ans.end());  //答案反转回来
    return ans;
}
int main() {
    string s1, s2;
    cin >> s1 >> s2;
    cout << mul(s1, s2) << endl;
    return 0;
}
```

有关乘法运算的优化部分，在本次笔记就不给大家说了，因为代码改动过大，在我们下次上课的时候再给大家去说，老师在这里先贴上代码

学有余力的同学可以去研究高精度乘法的优化代码，其他同学毋需掌握

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
string mul(string s1, string s2) {
    if (s1 == "0" || s2 == "0") {
        return "0";
    }
    int l1 = s1.size() - 1, l2 = s2.size() - 1;
    string ans(l1 + l2 - 1, '0');
    for (int i = l1; i >= 0; i--) {
        int x = 0;
        for (int j = l2; j >= 0; j--) {
            int a = s1[i] - '0', b = s2[i] - '0';
            x += a * b + ans[i + j] - '0';
            ans[i + j] = x % 10 + '0';
            x /= 10;
        }
        if (i) ans[i - 1] += x;
```

```
    }
    if (x) ans = to_string(x) + ans;
    return ans;
}
int main() {
    string s1, s2;
    cin >> s1 >> s2;
    cout << mul(s1, s2) << endl;
    return 0;
}
```