信息学奥赛笔记15

最大公因数(gcd)，最小公倍数(lcm)，高精度除以高精度

最大公因数

因数(约数Divisor)

一个数`a`的因数(Divisor)也叫约数，指的是$a/b(b \neq 0)$的商正好是整数而没有余数，则称`b`是`a`的因数

例如：`6`的因数有`[1，2，3，6]`。所有的数都可以被`1`整除，所以`1`是所有数的因数。

例如：12的因数有`1 2 3 6`

那么`1`可以被任何数整除，所以`1`是任何数的因数。

一个数的因数一定包含和它自身。所以每个数起码都有两个因数：`1`和`它自身`。

倍数(Multiple)

一个整数能够被另一个整数整除，这个整数就是另一整数的倍数。

例如：`15`是`3`和`5`的倍数。

更相减损术

现在我们有两个整数，我们将两个整数的因数一一列举出来，这两个整数因数当中的最大的共同值称为最大公因数(Greatest Common Divisor(GCD))

更相减损术是中国古代数学《九章算术》中的一个方法。

现在我们有两个整数`1547`和`1275`，想要求他们的最大公因数(GCD)。同学们此时就会发现，想要把他们的因数一一列举出来找共同最大值就不太现实了。

那么我们可以遵循以下的方法：

第一步，判断两个数是否有一个为`0`，如果是，说明最大公因数是另一个数。否则则进行到第二步

第二步，将两个数当中的较大数替换成这两个数的差。

$1547 - 1275 = 272$

$1275 - 272 = 1003$

$1003 - 272 = 731$

$731 - 272 = 459$

$459 - 272 = 187$

$272 - 187 = 85$

$187 - 85 = 102$

$102 - 85 = 17$

$85 - 17 = 68$

$68 - 17 = 51$

$51 - 17 = 34$

$34 - 17 = 17$

$17 - 17 = 0$

最大公因数为`17`。

通过以上思路我们可以发现，思路重复，可以使用循环，次数不定，应该用while循环解决。

```
#include<iostream>
#include<algorithm>
using namespace std;
int main() {
    long long a, b, t;
    cin >> a >> b;
    t = abs(a - b);  //t为a与b的差。
    while (t != 0) {
        a = b;          //a保存b的值，刚刚的最大值没了
        b = t;          //b保存刚刚减出来的差。
        t = abs(a - b);  //计算新的差
    }
    cout << b;
    return 0;
}
```

时间复杂度为$O(N)$，N为两数当中的较大数，因为极端情况需要求$(N, 1)$的最大公因数。需要执行N次循环。

空间复杂度$O(1)$，我们需要使用常数个变量就可以完成要求。

辗转相除法（欧几里得算法(Euclid's Algorithm))

在刚刚的更相减损术当中，我们可以发现，如果一个数特别大，一个数特别小，比如说我们要求`100,000,002`和`3`的最大公因数（显然是`3`）但我们发现通过相减的过程，一定会减上很多次`3`，也就是说，虽然我们已经用了一个较快的方式，但仍然会因为极端数据导致时间过慢，

比如说再刚刚的例子当中，我们在求`(1275, 272)`的最大公因数时，减去了很多次`272`，那直到什么时候为止呢？直到那个大的数比`272`小，也就是`272`变成了相对来说较大的值，那么我们可以通过取模定理的第一条结论：

$$m \quad \% \quad n = x \qquad x\text{的范围在}(0, m-1)\text{之间}$$

所以我们希望把`1275`变成一个小于`272`的值，我们可以计算$1275 \quad \% \quad 272 = 187$

所以我们可以把刚刚的减法部分全部变成$\%$，这样就变成了辗转相除法。

```
#include<iostream>
#include<algorithm>
using namespace std;
int main() {
    long long a, b, t;
    cin >> a >> b;
    t = a;                  // t先保存初始的a，这里我们会在后面提到。
    while(t != 0) {
```

```
        t = a % b;              // 求两个数的模
        a = b;                  // 将b的值保存给a，相当于第一步中我们做了一次减法，在这做了多次减法
        b = t;                  // b保存的是a与b取模后的值。
    }
    cout << a;                  // 由于最后 t为0时退出循环，说明a 在最后一次执行了a是b的倍数，那么a = b, b = 0, 答案被留在了a身上
    return 0;
}
```

递归优化

刚刚的代码如果写成while循环，假如我们需要在一道题目中求多次最大公因数，我们总不能每次在求的时候写这么一长串的代码，我们需要将他封装成函数，如果能够直接调用这个函数，那么我们在每次想要求最大公因数的时候，直接调用这个函数即可。

```
#include<iostream>
using namespace std;
long long gcd(long long a, long long b) {
    if (b == 0) return a; // 如果b是0了，结束递归，直接返回a，因为a不是0。
    return gcd(b, a % b); // 否则则继续辗转相除，此时相当于a = b, b = a % b;
}
int main() {
    long long a, b;
    cin >> a >> b;
    cout << gcd(a, b);
}
```

那么大家会产生一个疑问，如果`a` 比 `b` 小，这个程序是怎么运转的呢?

那么只要`a`和`b`中不产生`0`的话，程序会优先执行`gcd(b, a % b)`此时a会保存b的值，b会变成$a\%b$，但由于`a`更小，所以就是`a`本身，这样`a`和`b`就交换了，所以在这个递归中，只要存在了`a`比`b`小，就会先执行一次交换递归，再继续辗转相除。

三目运算符

```
如果一个变量的赋值不是x则是y我们可以这么写
int a;
if (条件成立) {
    a = x;
} else {
    a = y;
}
以上代码一共是5行，在C++中，有一种运算规则叫做三目运算符，它为我们提供了非常简短的写法，比如刚刚的例子
int a;
if (a % 2 == 1) {
    a = 119;
} else {
    a = 911;
}
我们可以用三目运算符优化为
```

```
int a = a % 2 == 1 ? 119 : 911;
三目运算符可以这么拆解:
a = 一个数，这个数有条件
a % 2 == 1 成立吗?
如果成立的话a = 119, 否则 a = 911;
int a = （条件）? 成立值 : 不成立值
```

三目运算符优化后的gcd，重要！！！！！！！

所以刚刚的递归可以进一步优化成:

```
#include<iostream>
using namespace std;
long long gcd(long long a, long long b) {
    return b != 0 ? gcd(b, a % b) : a;
}
int main() {
    long long a, b;
    cin >> a >> b;
    cout << gcd(a, b);
}
```

最小公倍数(Least Common Multiple(LCM))

我们表述$gcd(a,b)$为`a`和`b`的最大公因数

表述$lcm(a,b)$为`a`和`b`的最小公倍数

有如下定理:

$a * b = gcd(a,b) * lcm(a,b)$

所以移项得$lcm(a,b) = a * b / gcd(a,b)$

但由于$a * b$容易越界，我们一般会写成

$lcm(a,b) = a / gcd(a,b) * b$。

```
#include<iostream>
using namespace std;

long long gcd(long long a, long long b) {
    return b != 0 ? gcd(b, a % b) : a;
}

long long lcm(long long a, long long b) {
    return a / gcd(a, b) * b;
}
int main() {
    long long a, b;
    cin >> a >> b;
    cout << lcm(a, b);
```

```
}
```

时间复杂度$O(logN)$，N是$[M,N]$中的较大值。

空间复杂度$O(logN)$，对于每次递归我们需要使用$O(1)$的常数级变量。

高精度除以高精度

各位同学先联想一下我们的竖式除法，在做竖式除法的时候，需要涉及到几个操作呢？

① 从被除数中补数进入临时被除数。

② 用临时被除数除以除数得商。

③ 用临时被除数减商`*`除数继续执行步骤①

$$
\begin{array}{rll}
 & 772 & \\
16\overline{)} & 12345 & \\
 & 112 & (16 \times 7 = 112) \\
 & 114 & (123 - 112 = 11) \\
 & 112 & (16 \times 7 = 112) \\
 & 25 & (4 - 2 = 2) \\
 & 16 & (16 \times 1 = 16) \\
 & 9 &
\end{array}
$$

在上表中，临时被除数就是先`123`再`114`。

那么，在步骤③中，有乘法的过程，那么我们可以把乘法的过程，拆成一个个减法，$123 - 112 = 11$可以改成$123 - 16 * 7 = 11$相当于减去7次除数，那么最终能够减去`7`次就说明应该商`7`。所以我们可以把**验商**的过程改成高精度减法。比如说`123`除以`16`的商是`7`就可以减去`7`次`16`。

就需要在高精度除法中融入高精度减法。

我们仍然需要考虑两个问题。

① 如果我们最后发现临时被除数-商×除数，发现结果刚好是`0`的话，那么我们再执行步骤①的时候，就会发现补出的结果是`0`开头的。

② 如果我们的答案还未记录，此时发现应该商`0`，说明我们的临时被除数还不够大，此时不应该记录答案，如果记录答案`0`会导致答案出现前导`0`。

所以我们需要在代码中解决这两个问题。

```cpp
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
string sub(string s1, string s2) {
    string ans;
    int l = s1.size() - 1, r = s2.size() - 1, carry = 0, minus = 0;
    if (l < r || (l == r && s1 < s2)) return "-";
    while (l >= 0 || r >= 0) {
        int a = l < 0 ? 0 : s1[l--] - '0';
        int b = r < 0 ? 0 : s2[r--] - '0';
        carry = a + 10 - b - carry;
        ans += carry % 10 + '0';
        carry = carry / 10 == 1 ? 0 : 1;
    }
    while (ans.size() > 1 && ans.back() == '0') ans.pop_back();
```

```
    reverse(ans.begin(), ans.end());
    return ans;
}

string div(string s1, string s2) {
    int l1 = s1.size(), l2 = s2.size();
    if (l1 < l2 || l1 == l2 && s1 < s2) return "0";
    string ans, tmp;
    for (int i = 0; i < l1; i++) {
        if (tmp == "0") tmp = "";        //解决特殊问题1，结果刚好是0的时候抹去临时被除数
        tmp += s1[i];                    //对临时被除数补数
        int j = 0;
        for (j = 0; j <= 9; j++) {
            string ret = sub(tmp, s2);  //保存高精度减法的答案
            if (ret == "-") break;      //如果减完发现出现了负数，则直接退出。
            tmp = ret;                  //将结果保存进临时被除数
        }
        if (ans == "" && j == 0) continue;   //解决特殊问题2，如果结果为空，商是0则不商
        ans += j + '0';
    }
    return ans;
}
int main() {
    string s1, s2;
    cin >> s1 >> s2;
    cout << div(s1, s2);
    return 0;
}

```

时间复杂度$O(n)$，n是原数据的长度，需要对每一位求10次商。

空间复杂度$O(n)$，n是原数据的长度，需要把数据临时存储