算法基础（二十六）： 数学基础 - 组合数学 - 分解质因数与高精度

在这里我们不再考虑取模的情况，计算出来的是精确的组合的值，所以结果得用高精度

组合数的基本公式是

$$C_a^b = \frac{a*(a-1)*\dots(a-b+1)}{1*2*3*\dots b} = \frac{a!}{b!(a-b)!}$$

我们如果使用高精度来直接计算的话会发现时间复杂度过高，所以在此处我们先进行质因数分解，由于分子中`a`比`b`以及`a - b`都大，并且组合数的结果是一个整数，所以分母中分解得到的所有素因子分子中都有，如果分子与分母具有相同的素因数的话就用分子这个素因数的个数减去分母这个素因数的个数，这样就不用进行除法了，但是在此之前，先介绍一下阶乘的素因数分解

阶乘的素因数分解

先说结论，阶乘`a!`的素因子`p`的个数为：

$\lfloor a/p \rfloor + \lfloor a/p^2 \rfloor + \lfloor a/p^3 \rfloor \dots$

一直到后面的结果是`0`

我们以`12`为例来说明这个过程，`12! = 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1`，`12`的素因数是`2 3 5 7 11`，下面我们来分析素因子`2`的个数

首先，`12!`里面的`2 4 5 8 10 12`都是`2`的倍数，所以`12!`可以写作下面这样：

12 ! = (3 * 2 * 2) * 11 * (5 * 2) * 9 * (2 * 2 * 2) * 7 * (3 * 2) * 5 * (2 * 2) * 3 * 2 * 1

红色的`2`就是`12!`中`2`的倍数因子对应的第一种`2`，也就是$\lfloor 12/2 \rfloor$

接下来，我们发现`4 8 12`这三个因子中含有第二个`2`：

12 ! = (3 * 2 * 2) * 11 * (5 * 2) * 9 * (2 * 2 * 2) * 7 * (3 * 2) * 5 * (2* 2) * 3 * 2 * 1

绿色的`2`就是`12!`中对应的第二种`2`，其刚好对应着`4`的倍数因子个数，也就是$\lfloor 12/2^2 \rfloor$

最后，我们发现`8`含有第三种`2`：

12 ! = (3 * 2 * 2) * 11 * (5 * 2) * 9 * (2* 2 * 2) * 7 * (3 * 2) * 5 * (2* 2) * 3 * 2 * 1

蓝色的`2`就是`12!`中对应的第三种`2`，其刚好对应着`8`的倍数的因子的个数，也就是$\lfloor 12/2^3 \rfloor$

以上就是`12!`中素因子`2`的个数的分析过程，从中也就可以发现求阶乘`a!`中某个素因子`p`的规律，先把`a!`中`p`的倍数的因子个数求出来，他们含有第一种`p`，然后就是`p^2`的倍数的个数，他们含有第二种`p`，依次类推，从而知道了`a!`的素因子后就可以利用这种公式求得这些素因子的幂

代码实现

输入 a, b，求 C_a^b 的值。

注意结果可能很大，需要使用高精度计算。

输入格式

共一行，包含两个整数 a 和 b。

输出格式

共一行，输出 C_a^b 的值。

数据范围

$1 \leq b \leq a \leq 5000$

输入样例：

```
5 3
```

输出样例：

```
10
```

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;

const int N = 5010;

//primes[i]表示1 ~ a 中第i个质数
int primes[N], cnt;
//st[i] = false表示这时一个质数
bool st[N];

//sum[i]表示总的式子中1 ~ a第i个质数的次数
int sum[N];


//线性筛法筛质数
void get_primes(int n)
{
    for(int i = 2; i <= n; i ++)
    {
        //如果i是质数，则记录
        if(!st[i]) primes[cnt ++] = i;
        //筛掉这个合数，且保证筛掉的合数不超过n
        for(int j = 0; primes[j] <= n / i; j ++)
        {
            //用这个合数的最小质因子筛掉他
            st[primes[j] * i] = true;
            //假如i可以被primes[j]整除，就说明primes[j]不再是最小质因子，退出循环
```

```
            if(i % primes[j] == 0) break;
        }
    }
}

//求n!里面质因子p的个数
int get(int n, int p)
{
    int res = 0;
    while(n)
    {
        res += n / p;
        n /= p;
    }

    return res;
}

//高精度乘法
//计算的时候整数的低位存放在低地址
vector<int> mul(vector<int> a, int b)
{
    vector<int> c;
    int t = 0;//表示进位
    for(int i = 0; i < a.size(); i ++)
    {
        t += a[i] * b;//计算乘积并加上上一位的进位
        c.push_back(t % 10);//存入个位
        t /= 10;//得到当前位对下一位的进位
    }

    //如果t对后面还有进位，超过了原本的位数
    while(t)
    {
        c.push_back(t % 10);//存入个位
        t /= 10;//计算当前位对下一位的进位
    }

    //由于是低位存放在低地址，所以不会出现前导0 的情况
    return c;

}

int main()
{
    int a, b;
    cin >> a >> b;

    //利用线性筛法预处理1 ~ a中的所有的质数
    //注意a!并不包含1 ~ a中的所有的质数，在这里我们只是先进行一个预处理
    get_primes(a);

```

```
    //得到1 ~ a中所有的质数，一共有cnt个

    //计算总的式子中质数的个数
    for(int i = 0; i < cnt; i ++)
    {
        //当前这个质数是primes[i]
        int p = primes[i];

        //计算当前这个质数p在总的公式里占的个数，也就是它的次数
        //p不可能是负数，因为如果p是负数，就说明分母中含有分子中不存在的素因数，就说明结果不是一个整数显然不成立
        sum[i] = get(a, p) - get(b, p) - get(a - b, p);
    }

    //用高精度将所有的质因子乘起来
    vector<int> res;
    //初始时为1
    res.push_back(1);

    //从前往后枚举所有的质数
    for(int i = 0; i < cnt; i ++)
    {
        //再枚举这个质数的所有的次数
        for(int j = 0; j < sum[i]; j ++)
        {
            //高精度乘法乘积
            res = mul(res, primes[i]);
        }
    }

    //从后往前输出答案
    for(int i = res.size() - 1; i >= 0; i --)
    {
        printf("%d", res[i]);
    }
    puts("");
    return 0;
}
```

时间复杂度：

线性筛的复杂度最多计算`5000`，高精度乘法的复杂度与`res`的位数有关，最多计算的次数是C_a^b的最大值的位数，其中`a b`的范围是`1 ~ 5000`，总的来说复杂度是很低的