数论

```cpp
/**
 * 简单结论 conclusion
 * 最大公约数 gcd
 * 基础数论-质数：  prime
 * 线性逆元 inv
 * 快速幂 qmi
 * 排列组合 permutation_and_combination
*/

#include <iostream>
#include <cstring>
#include <vector>
#include <algorithm>
using namespace std;
typedef long long LL;
const int N = 1e5 + 21;

namespace golitter {
namespace conclusion {
    // y = x + b函数性质
    // 调和级数 harmonic_progression  时间复杂度 O( nlog(n))
    // 哥德巴赫猜想
    // 完美正方形
/**
 * y = x + b函数性质
 * 四面八方：y = -x + b , y = x + b , y = b, x = a 的结论
 * 恒有:  y + x === ;
 *        y - x === ;
 *         b  === ;
 *         a  === ;
 * url: https://codeforces.com/contest/1850/problem/G
*/

/**
 * 调和级数 https://www.zhihu.com/search?
hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%2264
5232342%22%7D&hybrid_search_source=Entity&q=%E8%B0%83%E5%92%8C%E7%BA%A7%E6%95%B0
&search_source=Entity&type=content
 * 时间复杂度为 O(nlog(n))
 * 1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 ... + 1/n
 * n = 1e6时, O(16.7 * n == 1e7 )
 * n =  1e8时 O(21.3 * n == 2e7)
 * ** 赛时想到了这种解法，但是直觉感觉是 O(n*n) ，赛后看证明调和级数发现是 O(nlog(n))
 * ** 反直觉
*/
void harmonic_progression() { // https://codeforces.com/contest/1850/problem/F
    int n; cin>>n;
    vector<LL> ans(n+1), cnt(n+1);
    for(int i = 0; i < n; ++i) {
```

```
        int a; cin>>a;
        if(a <= n) cnt[a]++;
    }
    for(int i = 1; i <= n; ++i) {
        for(int j = i; j <= n; j += i) {
            ans[j] += cnt[i];
        }
    }
    cout<<*max_element(ans.begin(), ans.end())<<endl;

}

/**
 // 哥德巴赫猜想

哥德巴赫猜想的现代陈述为：任一大于5的整数都可写成三个质数之和。

由此可以得到两种情况：

1.强哥德巴赫猜想（强哥猜）：即任一大于2的偶数都可写成两个质数之和；（未被证明，但是同时也没有被推翻，即在本体的范围内强哥猜成立）

2.弱哥德巴赫猜想（弱哥猜）:任何一个大于5的奇数都能被表示成三个奇质数的和。（一个质数可以被多次使用）;（已经被证明）

*/

/**
 * 截至2018年，已经知道21~35阶完美正方形的个数：1，8，12，30，172，541，1372，3949，10209，26234，71892，196357，528866，1420439，3784262。
*/

}}

namespace golitter {
namespace gcd {
int gcd(int a, int b) {
    return b ? gcd(b, b%a) : a;
}
}}

namespace golitter {
namespace inv {

// https://www.cnblogs.com/chy-2003/p/9656801.html
// 只能互质，且 x < mod才行
LL mod = 131;
// A / B % mod == A * inv(B, mod) % mod;
LL inv(LL a) {
    if(a == 0 || a == 1) return a;
    return (mod - mod/a) * inv(mod % a) % mod;
}
LL inv(LL x, LL mod) {
    if(x == 0 || x == 1) return x;
    return (mod - mod/x) * inv(mod % x, mod) % mod;
}

// 无限制逆元
```

```cpp
// 欧几里得拓展,x是a对于mod b的逆元 [O(logN)]
LL exgcd(LL a, LL b, LL &x, LL &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    LL d = exgcd(b, a % b, y, x);
    y = y - a / b * x;
    return d;
}

// 求解最小正整数解
LL inv(LL b, LL mod) {
    LL x, y;
    LL gcd = exgcd(b, mod, x, y);
    if (gcd != 1)
        return -1;
    else {
        LL tb = mod / gcd;
        LL minX = (x % tb + tb) % tb;
        return minX;
    }
}

}}

namespace golitter {
namespace prime {

// 判断质数
bool isPrime(int x) {
    if(x < 2) return false;
    for(int i = 2; i <= x / i; ++i) {
        if(x % i == 0) return false;
    }
    return true;
}

// 试除法分解质因数
void divide(int x) {
    for(int i = 2; i <= x / i; ++i) {
        if(x % i == 0) {
            int s = 0;
            while(x % i == 0) x /= i, s++;
            cout<<i<<" "<<s<<endl;
        }
    }
    if(x > 1) cout<<x<<" "<<1<<endl;
}

// 线性筛
int primes[N],cnt; bool st[N];
void get_primes(int n) {
    for(int i = 2; i <= n; ++i) {
        if(!st[i]) primes[cnt++] = i;
        for(int j = 0; primes[j] <= n / i; ++j) {
            st[ primes[j] * i] = true;
```

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

// 试除法求约数
vector<int> get_divisors(int x) {
    vector<int> res;
    for(int i = 1; i <= x / i; ++i) {
        if(x % i == 0) {
            res.push_back(i);
            if(i != x / i) res.push_back(x / i);
        }
    }
    sort(res.begin(), res.end());
    return res;
}

/**

如果 N = p1^c1 * p2^c2 * ... *pk^ck
约数个数: (c1 + 1) * (c2 + 1) * ... * (ck + 1)
约数之和: (p1^0 + p1^1 + ... + p1^c1) * ... * (pk^0 + pk^1 + ... + pk^ck)

*/

}}

namespace golitter {
namespace qmi {
int m,k,p; // 求 m ^ k mod p
int qmi(int m, int k, int p) {
    int res = 1 % p, t = m;
    while(k) {
        if(k & 1) res = res * t % p;
        t = t * t % p;
        k >>= 1;
    }
    return res;
}


}}

namespace golitter {
namespace permutation_and_combination {
int C[N][N]; // C[a][b] 表示从a个苹果中选取b个的方案数
const int MOD = 1e9 + 33;
void comb(int n) {
    for(int i = 0; i <= n; ++i) {
        for(int j = 0; j <= i; ++j) {
            if(!j) C[i][j] = 1;
            else C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % MOD;
        }
    }
}

/**
```

```
 *
 * 在计数时，必须注意没有重复，没有遗漏。为了使重叠部分不被重复计算，人们研究出一种新的计数方法，这种方法的基本思想是：先不考虑重叠的情况，把包含于某内容中的所有对象的数目先计算出来，然后再把计数时重复计算的数目排斥出去，使得计算的结果既无遗漏又无重复，这种计数的方法称为容斥原理。
 * A∪B∪C = A+B+C - A∩B - B∩C - C∩A + A∩B∩C
 * + - + - + - + - + - + - + - + - + - + - + - ### 注意
*/


}}
```