从998244353开始的数论函数筛法教程

本文内容源于hehelego/spinach的笔记/讲稿/校内研学报告,部分内容参考了网络资料(包括但不限于wikipedia词条,OI集训队论文集,神仙们的blog和<concrete mathematics>)

如果您认为文章中的某些内容有侵权行为,请立刻联系我

本文作者正在学文化课,不能保证及时更正文章内出现的错误,抱歉.

0.记号与约定

- 文中出现的小写字母,如不做特殊说明,均为正整数.
- 默认出现在除数位置的数字不为0.
- 不区分质数与合数.
- 定义函数[p],其中p为一命题,p为真时[p]=1,p为假时[p]=0.
- 定义函数$\lfloor x\rfloor = \max\{y\in\mathbb{Z}\mid y\le x\}$即向下取整函数.
- 定义函数$\lceil x\rceil = \min\{y\in\mathbb{Z}\mid y\ge x\}$即向上取整函数.
- 默认$\sum_{i=a}^{b} f(i) = \sum_{i=\lceil a\rceil}^{\lfloor b\rfloor} f(i)$出现在积分,求和上/下限的数分别向下/上取整.
- 当$a>b$时定义$\sum_{i=a}^{b} f(i)=0$
- 当$a>b$时定义$\prod_{i=a}^{b} f(i)=1$
- 定义$prime$为全体素数构成的集合,$prime_i$为第i个素数(2为第1个).
- 定义$mp(n)=min\{p\in prime\mid\ p\mid n\}$即最小质因子.

1.素数筛法

1.1 筛法简介.

Sieve of Eratosthenes是一个古老的算法,它基于这样一个思想$n\in prime\Rightarrow \forall p\in prime,p<n\ p\nmid n$,这一命题的正确性比较显然,不做证明.Eratosthenes筛算法通过不断找出素数,筛出合数(排除含有因子p且大于p的数,即p的倍数,这些数都是合数)从而得到不超过给定上限的所有素数.需要注意的是,如果对于每一个数字q都尝试用$p\in prime,p\le\sqrt{q}$进行测试的话,算法复杂度会退化为$O(n\sqrt{n})$.

1.2 筛法的实现与优化.

基于以上想法,我们不难给出这一算法的实现.

```
int vis[N],prime[N],cnt;
void sieve(){
    vis[1]=1;
    for(int i=2;i<N;i++) if(!vis[i]){
        prime[cnt++]=i;
        for(int j=2;i*j<N;j++) vis[i*j]=1;
    }
}
```

这一实现已经非常高效了,但是仍有优化余地.考虑形如$2p, p \in prime$的数,已经被2排除为质数的可能后,还会被p再次排除,我们使用p进行筛除时应当避免被更小的质数筛除过的数,换言之,应该筛除$mp(n) \geq p$的合数,这种n满足$p^2 \leq n$,因为最小非1因子(即为最小质因子)不小于p那么必须再乘一个不小于p的数构成合数,显然是不小于p^2的.

基于这一想法,我们对实现进行优化,得到如下的代码片段(需要说明的是,这一代码并不对于任意大的n可用,可能出现乘法溢出等问题).

```
int vis[N],prime[N],cnt;
void sieve(){
    vis[1]=1;
    for(int i=2;i*i<N;i++) if(!vis[i])
        for(int j=i*i;j<N;j+=i) vis[j]=1;
    for(int i=2;i<N;i++) if(!vis[i]) prime[cnt++]=i;
}
```

然而即使加了这一优化,也无法做到每个合数只被筛除一次.不过我们知道一个数n的不同质因子个数是远低于$O(logn)$量级的.算法复杂度并不会高于$O(nlogn)$.

另外,这一算法的空间复杂度也过高,如求解10^{11}内的质数,将会需要100GB的以上的空间,无法存储在内存中而需要进行磁盘的IO,cache也由于大步长访问难以命中,严重影响性能.针对空间问题,我们发现prime数组是无用的,省去,vis数组元素的取值仅为0,1,可以使用位图代替数组(如c++中的STL容器bitset,java中的bitmap等),大约空间使用降低到原来的$\frac{1}{64*4}$.

1.3 筛法的复杂度分析.

空间复杂度分析略过,考虑算法的时间复杂度,主要花销在于遍历质数p的倍数,在n以内有$\lfloor \frac{n}{p} \rfloor$,不难列出.

$$T(n) = \sum_{p \in prime} \lfloor \frac{n}{p} \rfloor \leq n \sum_{\substack{p \in prime \\ p \leq n}} \frac{1}{p} < n \sum_{i=1}^{n} \frac{1}{i} = nH(n) = O(nlogn)$$

不过把素数倒数和放大到调和级数太松了,通过查阅资料我们发现$\sum_{\substack{p \in prime \\ p \leq n}} \frac{1}{p}$大约是$O(loglogn)$量级,故

$T(n) = O(nloglogn)$这是一个足够好的上界.

另一方面,根据$\pi(x) = O(\frac{n}{logn})$,任取一$n$以内的正整数,它为素数的概率是$O(\frac{1}{logn})$量级的.我们可以用期望来估算$E(T(n)) \leq n \sum_{i=1}^{n} \frac{1}{ilogn} = O(nloglogn)$,得到了相同的结果.

1.4 线性筛

线性筛即时间复杂度为线性的筛法,由欧拉提出,也称Euler's sieve.这一算法可以看作是对于Sieve of Eratosthenes的优化也可以看作是由Eratosthenes筛不同的原理推导出的算法.这一算法的核心是"寻找最小质因子".

遍历[2...n],维护已经发现的素数列表和标记数组(被标记的不是素数).若遇到未被标记的数字,则发现了新质数,插入素数列表尾部.对于q,用质数$p_1 = 2, p_2 = 3 \dots p_m$,其中$p_m \mid q$标记$qp_i$,且$mp(qp_i) = p_i$.我们先基于这一描述给出实现.

```
int vis[N],prime[N],cnt;
void sieve(){
    for(int i=2;i<N;i++){
        if(!vis[i]) prime[cnt++]=i;
        for(int j=0;j<cnt&&i*prime[j]<N;j++){
            vis[i*prime[j]]=1;
            if(i%prime[j]==0) break;
        }
    }
}
```

可以证明,这一算法中,每个合数x被$mp(x)$筛除,且仅被筛除一次.下面给出证明. 考虑$i = q$时,其中q的标准分解形式为$\prod_{i=1}^{n}(prime_{a_i})^{b_i}$,被筛除的数为$m = q\,prime_j$其中$j \le a_1$.显然$mp(m) \le prime_j$ 当$j < a_1$时$prime_j \nmid q$,若$mp(m) = mp(q\,prime_j) = prime_k < prime_j$,可以得到$prime_j\,q = prime_k\,q'$其中$q' > q$.将$prime_j\,q = prime_k\,q'$放到模$prime_k$意义下,有$prime_j\,q \equiv 0 \pmod{prime_k}$,由于$gcd(prime_j, prime_k) = 1$故$\exists x < prime_k$使得$x\,prime_j \equiv 1 \pmod{prime_k}$,等式两边同时乘上$x$,得到$q \equiv 0 \pmod{prime_k}$从而有$prime_k \mid q$,但是$prime_k < prime_j < prime_{a_1} = mp(q)$我们发现$q$的最小质因子并非$prime_{a_1}$产生了矛盾. 当$j = a_1$时$mp(q) = prime_{a_1} = prime_j$,考虑$m = q\,prime_j$的标准分解形式,$m = prime_{a_1}^{b_1+1}\prod_{i=2}^{n}(prime_{a_i})^{b_i}$,显然有$mp(m) = mp(q) = prime_{a_1} = prime_j$ 综上,每个数只会被最小质因子筛除,而筛除时是$i\,prime_j$这样一个配对形式,所以$m = \frac{m}{mp(m)}\,mp(m)$被筛除时有$prime_j = mp(m)$进而$i = \frac{m}{prime_j} = \frac{m}{mp(m)}$是唯一确定的正整数.这说明了每个合数$x$会被$mp(x)$筛除,且只会被筛除一次.所以该算法的时间复杂度显然是线性的.

线性筛法看起来非常高效且充满了美感,但是我们仍要指出,这一算法的实现中涉及了取模运算,是非常耗时的操作,当数据规模较小(大约10^8以内)时,这一实现表现不及Eratosthenes筛.并且由于算法过程的需要,我们必须保存素数列表空间消耗无法进一步优化.可以说再寻找素数这一工作上,euler's sieve被前辈Eratosthenes筛吊打了.不过由于这一算法能够找到$[1..n]$内所有数的最小质因子,它能够帮助我们完成一些特殊的积性函数求值任务,这是Eratosthenes筛难以做到的.

2.数论函数求和方法

2.0 一些定义

- 数论函数,一个函数的定义域为正整数的子集,则称它为数论函数
- 数论积性函数,一个数论函数$f(n)$若满足$\forall a, b\ (a, b) = 1 \Rightarrow f(ab) = f(a)f(b)$
- 完全数论积性函数,一个数论函数$f(n)$满足$\forall a, b \quad f(ab) = f(a)f(b)$
- 定义两个数论函数$f(n), g(n)$的dirichlet卷积,为一个函数$h(n) = \sum_{d|n} f(d)g(\frac{n}{d})$
- 定义$e(n) = [n = 1]$称单位函数
- 定义$1(n) = 1$称常函数

- 定义$id_k(n) = n^k$
- 定义$\sigma_k = id * 1$
- 定义$\mu(n)$为满足$1 * \mu = e$的数论函数,可以证明,μ是唯一存在的,且为一个积性函数
- 定义$\varphi = \mu * id$,(根据容斥原理)可以证明$\varphi(n) = \sum_{i=1}^{n} [gcd(i, n) = 1]$,这是一个积性函数

2.1 线性筛法

考虑如果在$O(n)$时间内求出所有的$\varphi(i)$其中$i \le n$,我们先不加证明地给出以下性质.

$$\varphi(1) = 1$$
$$\varphi(p^k) = p^{k-1}(p-1) = \quad p \in prime$$
$$\varphi(n) = \varphi(\prod_{i=1}^{m} p_i^{c_i}) = n\prod_{i=1}^{m}(1 - \frac{1}{p_i})$$

性质3告诉我们这样一个事实,若$m = pq$,其中$p \mid q$则$\varphi(m) = \varphi(q)\, q$.考虑将它整合到线性筛法中.对于$q$,筛除$prime_j\, q \quad prime_j \mid q$时,有$\varphi(prime_j\, q) = \varphi(prime_j)\varphi(q) = (p-1)\varphi(q)$.当筛除$mp(q)q$时$\varphi(mp(q)q) = mp(q)\varphi(q)$.得到了一个可行的算法.给出实现如下.

```
int vis[N],prime[N],cnt,phi[N];
void sieve(){
    phi[1]=1;
    for(int i=2;i<N;i++){
        if(!vis[i]) phi[prime[cnt++]=i]=i-1;
        for(int j=0;j<cnt&&1LL*i*prime[j]<N;j++){
            vis[i*prime[j]]=1;
            if(i%prime[j]==0){
                phi[i*prime[j]]=phi[i]*prime[j];
                break;
            }
            phi[i*prime[j]]=phi[i]*(prime[j]-1);
        }
    }
}
```

这一方法也可以推广到一些满足$f(p^k)$为一个关于p, k的低次(多项式次数deg相比于求解规模n可以忽略)多项式的情况,给出两个例子.

$O(n)$时间内求出所有求$\sigma(n) = \sum_{d|n} d$和$id_k(n) = n^k$,其中$k = O(n)$

对于$id_k(n)$是一个完全积性函数筛除合数时简单相乘即可,并在$n \in prime$时使用快速幂技巧(一个经典的倍增算法)进行计算.发现时间复杂度$T(n) \le O(n) + \pi(n)O(logk) = O(n) + O(\frac{nlogk}{logn}) = O(n) + O(n) = O(n)$仍然是线性的.

对于$\sigma(n)$函数,显然是积性函数,且$\sigma(p^k) = \sum_{i=0}^{k} p^i$.故$\sigma(p^{k+1}) = \sum_{i=0}^{k+1} p^i = p\sigma(p^{k+1}) + 1$ 对于$m = i\, prime_j$,其中$prime_j \nmid i$的情况,$\sigma(m) = \sigma(i)\sigma(prime_j) = 2\sigma(i)$.对于$m = i\, mp(i)$的情况,设$i = mp^a(i)\, q$,其中$a = max\{b \mid mp^b(i) \mid i\}$即标准分解形式中最小质因子的指数,显然$gcd(mp(i), q) = 1$.故$\sigma(m) = \sigma(i\, mp^{a+1}(i)) = \sigma(mp^{a+1}(i)\, q) = \sigma(q)\sigma(mp^{a+1}(i)) = \sigma(q)[mp(i)\, \sigma(mp^a(i)) + 1]$筛法求素数的过程中,维护$\sigma(mp^k(i))$其中$k = max\{a \mid mp^a(i) \mid i\}$即可在解决问题.

更进一步地,对于给定的两个可以线性求解的积性函数$f(n), g(n)$,我们可以线性求解$h(n) = (f * g)(n)$.

2.2 整除分块

这是一个用于求$\sum_{i=1}^n \sigma_0(i)=\sum_{i=1}^n \lfloor \frac{n}{i} \rfloor$的技巧,可以在$O(\sqrt{n})$时间内完成求解. 对于一个给定的$n$,$f_n(d)=\lfloor \frac{n}{d} \rfloor$显然关于$d$单调不增,最大为$n$,最小为1似乎有$n$种可能取值.不过一个好消息是$d>\frac{n}{2} \Rightarrow f_n(d)<\frac{n}{\frac{n}{2}}=2 \Rightarrow f_n(d)=1$这说明有一半以上的$f_n(d)$是相同的.更进一步的,$f_n(d)=\lfloor \frac{n}{d} \rfloor$的可能取值是$O(\sqrt{n})$种.我们下面给出证明.

对于$d \leq \sqrt{n}$,这种d是$O(\sqrt{n})$的,即使每一个$f_n(d)$都不同,取值也是$O(\sqrt{n})$的.对于$d>\sqrt{n}$,$\lfloor \frac{n}{d} \rfloor \leq \frac{n}{\sqrt{n}} \leq \sqrt{n}$,于是可能的取值仍然是$O(\sqrt{n})$的.综合两部分,$\lfloor \frac{n}{d} \rfloor$对于给定的n,只有$O(\sqrt{n})$种取值.

这一证明的指导思想是$ab \leq n \Rightarrow min(a,b) \leq \sqrt{n}$.值得一提的是:这一上界是非常紧的,难以得到更紧且更优美的上界.

由于$\lfloor \frac{n}{d} \rfloor$的单调性,考虑使得$\lfloor \frac{n}{d} \rfloor \neq \lfloor \frac{n}{d-1} \rfloor$的$d$,记为$d_1,d_2 \ldots d_m$,它们会将区间$[1,n]$分成$O(\sqrt{n})$个区间.如果我们能对于一个确定的$d$,$O(1)$时间内求出$q=max\{p \mid \lfloor \frac{n}{p} \rfloor = \lfloor \frac{n}{d} \rfloor\}$即可在$O(\sqrt{n})$时间内计算$\sum_{i=1}^n \lfloor \frac{n}{i} \rfloor$.通过手动找规律猜想结合证明,我们得到$max\{p \mid \lfloor \frac{n}{p} \rfloor = d\} = \lfloor \frac{n}{d} \rfloor$.我们跳过对此的性质的严谨证明.下面直接给出求解$\sum_{i=1}^n \lfloor \frac{n}{i} \rfloor$的算法的实现.需要注意的是,由于直接使用c++ int类型(32bit有符号整数),这一实现在n较大时会发生整数溢出.

```cpp
int solve(int n){
    int res=0,l=1,r=0,q=0;
    while(l<=n){
        q=n/l; r=n/q;
        res=res+q*(r-l+1);
        l=r+1;
    }
    return res;
}
```

有了这一算法的启发,求解形如$\sum_{i=1}^n f(i)\lfloor \frac{n}{i} \rfloor$和$\sum_{i=1}^n f(\lfloor \frac{n}{i} \rfloor)p(i)$的问题时,也可以根据$\lfloor \frac{n}{d} \rfloor$的结果对于$d$将答案切分成几个不相交的块进行计算.这一技巧被国内算法竞赛选手称为"整除分块".

2.3 dirichlet卷积

定义两个数论函数$f(n),g(n)$的dirichlet卷积卷积为一个数论函数$h(n)$,满足$h(n)=\sum_{d|n} f(d)g(\frac{n}{d})$.记为$h=f*g$.下面我们不加证明地给出一些dirichlet卷积的性质.

- 交换律:$f*g=g*f$
- 结合律:$(f*g)*h=f*(g*h)$
- 对加法的分配律:$f*(g+h)=f*g+f*h$
- 存在单位元:$e(n)=[n=1]$
- 两个积性函数的卷积仍然是积性函数.

  dirichlet卷积是一个非常有用的工具,下面举一个例子来说明.

以低于$O(n)$的时间求$S(n)=\sum_{i=1}^{n}\mu(i)$.根据$\mu$的定义$\mu*1=e$.我们下面尝试使用卷积构造$S(n)$的递推式.

$$1=\sum_{i=1}^{n}e(i)=\sum_{i=1}^{n}(\mu*1)(i)=\sum_{i=1}^{n}\sum_{d|i}1(d)\mu(\frac{i}{d})$$

$$\sum_{i=1}^{n}\sum_{d|i}\mu(\frac{i}{d})=\sum_{i=1}^{n}\sum_{d=1}^{n}[d\mid i]\mu(\frac{i}{d})=\sum_{d=1}^{n}\sum_{i=1}^{n}[d\mid i]\mu(\frac{i}{d})$$

$$=\sum_{d=1}^{n}\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\mu(\frac{id}{d})=\sum_{d=1}^{n}\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\mu(i)=\sum_{d=1}^{n}S(\lfloor\frac{n}{d}\rfloor)$$

$$1=\sum_{d=1}^{n}S(\lfloor\frac{n}{d}\rfloor)=S(n)+\sum_{d=2}^{n}S(\lfloor\frac{n}{d}\rfloor)$$

$$S(n)=1-\sum_{d=2}^{n}S(\lfloor\frac{n}{d}\rfloor)$$

利用这个递推式,进行记忆化搜索,就可以在低于线性的时间内解决这一问题了.这里给出实现.

```
typedef long long Int;
const Int mod=998244353LL;
std::unordered_map<Int,Int> tbl;
Int sum(Int n){
    if(n<1) return 0;
    if(tbl.count(n)) return tbl[n];
    Int ret=1,l=2,r=0,q=0;
    while(l<=n){
        q=n/l; r=n/q;
        ret=ret-sum(q)*(r-l+1)%mod;
        ret=(ret%mod+mod)%mod;
        l=r+1;
    }
    return tbl[n]=ret;
}
```

下面我们来分析以下算法的复杂度.这里先给出一个关于向下取整函数的性质.$\lfloor\frac{a}{bc}\rfloor=\lfloor\frac{\lfloor\frac{a}{b}\rfloor}{c}\rfloor$利用带余除法很好证明,这里跳过.这一性质告诉我们如果利用这一递推式进行计算,涉及到的不同$S(m)$是和$\lfloor\frac{n}{a}\rfloor$的取值种类一样为$O(\sqrt{n})$个.给出这一结论的证明.

首先$n=\lfloor\frac{n}{1}\rfloor$,$S(n)=1-\sum_{d=2}^{n}S(\lfloor\frac{n}{d}\rfloor)$,只涉及到形如$S(\lfloor\frac{n}{a}\rfloor)$的项.

对于其中的每个$S(\lfloor\frac{n}{d}\rfloor)$再次展开,$S(\lfloor\frac{n}{d}\rfloor)=1-\sum_{p=2}^{\lfloor\frac{n}{d}\rfloor}S(\lfloor\frac{\lfloor\frac{n}{d}\rfloor}{p}\rfloor)=1-\sum_{p=2}^{\lfloor\frac{n}{d}\rfloor}S(\lfloor\frac{n}{dp}\rfloor)$.仍然是形如$S(\lfloor\frac{n}{a}\rfloor)$的项.继续展开显然只会涉及有限项,同理可以说明,所以结论正确.

接下来可以估计时间复杂度了,类似于证明$\lfloor\frac{n}{d}\rfloor$的取值只有$O(\sqrt{n})$种的过程,我们考虑按照$d$和$\sqrt{n}$的大小关系分类进行计算.由于采用了记忆化搜索,我们直接估计转移的次数(进入sum函数的次数)即可.具体地,对于$S(n)$,需要先求出$S(\lfloor\frac{n}{i}\rfloor)\quad i\le n$,共$O(\sqrt{n})$个转移.下面给出复杂度计算.

$$T(n)=\sum_{i=1}^{\sqrt{n}}\sqrt{i}+\sum_{i=1}^{\sqrt{n}}\sqrt{\frac{n}{i}}\quad \text{使用积分代替求和进行估计}$$

$$\int_1^{\sqrt{n}}\sqrt{x}dx+\int_1^{\sqrt{n}}\sqrt{\frac{n}{x}}dx=\frac{2}{3}x^{\frac{3}{2}}|_1^{\sqrt{n}}+(2\sqrt{n}x^{\frac{1}{2}})|_1^{\sqrt{n}}$$

$$=O(n^{\frac{3}{4}})-O(1)+O(n^{\frac{3}{4}})-O(1)=O(n^{\frac{3}{4}})$$

显然积分估计的结果和求和的结果是远低于$O(n^{\frac{3}{4}})$的,所以$T(n)=O(n^{\frac{3}{4}})$,这个上界是较紧的.

回顾推导过程,我们并没有利用$\mu(n),1(n)$各自的特殊性质,只是对$e(n)=(\mu*1)(n)$进行求和便得到了递推式,所以这一方法可以轻松地推广到任意两个数论函数$f(n),g(n)$上.令$S(n)=\sum_{i=1}^{n}g(i)$,给出$S(n)$的递推式.

$$\sum_{i=1}^{n}(f*g)(i)=\sum_{i=1}^{n}\sum_{d|i}f(d)g(\frac{i}{d})=\sum_{d=1}^{n}f(d)\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}g(i)=\sum_{d=1}^{n}f(d)S(\lfloor\frac{n}{d}\rfloor)$$

$$f(1)S(n)=\sum_{i=1}^{n}(f*g)(i)-\sum_{d=2}^{n}f(d)S(\lfloor\frac{n}{d}\rfloor)$$

套用刚刚的记忆化搜索,若求解$\sum_{i=1}^{n}(f*g)(i),\sum_{i=1}^{n}f(i)$在全部$\lfloor\frac{n}{d}\rfloor$处的取值的总复杂度不超过$O(n^{\frac{3}{4}})$则可以$O(n^{\frac{3}{4}})$内求解$S(n)$.此外若可以$O(n)$时间内求出$S(1),S(2)\dots S(n)$.还可以对$S(n)$进行预处理,从而把复杂度从$O(n^{\frac{3}{4}})$优化到$O(n^{\frac{2}{3}})$,具体地.则使用$O(n^{\frac{2}{3}})$时间预处理$S(1),S(2)\dots S(\lfloor n^{\frac{2}{3}}\rfloor)$,后再进行记忆化搜索即可.利用积分估计复杂度.

$$T(n)=O(n^{\frac{2}{3}})+\int_1^{n^{\frac{1}{3}}}\sqrt{\frac{n}{x}}dx=O(n^{\frac{2}{3}})+O(n^{\frac{1}{2}+\frac{1}{2}\times\frac{1}{3}})=O(n^{\frac{2}{3}})$$

这个利用dirichlet卷积构造递推式进行记忆化搜索的技巧,在国内算法竞赛圈内被称为"杜教筛",由杜瑜皓引入国内.

2.4 莫比乌斯反演

莫比乌斯反演是这样一个技巧$f(n)=\sum_{d|n}\mu(\frac{n}{d})g(d)$,其中$g(n)=\sum_{d|n}f(d)$.

上述等式很容易借助dirichlet卷积证明,$f=(\mu*1)*f=\mu*(1*f)$.也可以根据$\mu(\prod_{i=1}^{n}prime_{a_i})=(-1)^n$和$\sum_{i=1}^{n}(-1)^i\binom{n}{i}=[n=1]$进行证明,或者基于多重集容斥原理证明,由于莫比乌斯反演和本文主题关联不大,故略去对它们的详细介绍.

这里给出一个使用莫比乌斯反演和卷积的例题.求$\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}gcd(i_1,i_2,i_3,\dots i_n)$其中$m=\Theta(n)$即

n,m同阶,gcd为表示最大公约数,这一题目来自于陈卓裕,可以在LOJ 6491提交.

由于最大公约数函数并没有很好的性质,我们考虑转求和为计数.考虑gcd的每一种取值d会出现多少次,即有多少组$(i_1,i_2\dots i_n)$满足$(i_1,i_2\dots i_n)=d$.并将$\sum_{d|n}\mu(d)=e(n)$带入.给出完整推导过程.

$$\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}gcd(i_1,i_2,i_3,\dots i_n)=\sum_{d=1}^{m}d\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}[gcd(i_1,i_2,i_3,\dots i_n)=d]$$

$$=\sum_{d=1}^{m}d\sum_{i_1=1}^{\lfloor\frac{m}{d}\rfloor}\sum_{i_2=1}^{\lfloor\frac{m}{d}\rfloor}\dots\sum_{i_n=1}^{\lfloor\frac{m}{d}\rfloor}[gcd(i_1,i_2,i_3,\dots i_n)=1]=\sum_{d=1}^{m}df(n,\lfloor\frac{m}{d}\rfloor)$$

$$f(n,m)=\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}[gcd(i_1,i_2\dots i_n)=1]=\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}\sum_{dgcd(i_1,i_2\dots i_n)}\mu(d)$$

$$=\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}\sum_{d|i_1\&p|i_2\dots p|i_n}\mu(d)=\sum_{d=1}^{m}\mu(d)\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}[d|i_1][d|i_2]\dots[d|i_n]$$

$$=\sum_{d=1}^{m}\mu(d)(\lfloor\frac{m}{d}\rfloor)^n$$

$$\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m}gcd(i_1,i_2,i_3,\dots i_n)=\sum_{d=1}^{m}df(n,\lfloor\frac{m}{d}\rfloor)$$

$$=\sum_{d=1}^{m}d\sum_{p=1}^{\lfloor\frac{m}{d}\rfloor}\mu(p)(\lfloor\frac{m}{pd}\rfloor)^n=\sum_{T=1}^{m}\sum_{d|T}d\mu(\frac{T}{d})(\lfloor\frac{m}{T}\rfloor)^n=\sum_{T=1}^{m}(\lfloor\frac{m}{T}\rfloor)^n\varphi(T)$$

套用整除分块和杜教筛,再预处理$\varphi(n)$的前$m^{\frac{2}{3}}$项,即可$O(n^{\frac{2}{3}})$解决问题.值得注意的是,这里需要求解$S(n)=\sum_{i=1}^{n}\varphi(i)$在$\lfloor\frac{m}{d}\rfloor$其中$d=1,2,3\dots m$,共$O(\sqrt{m})$处的取值,而不仅仅是$S(m)$,这里复杂度是如何保证不退化的呢?

考虑记忆化搜索的过程,我们实际上遍历了$S(\lfloor\frac{m}{d}\rfloor)$其中$d=1,2,3\dots m$,并且将它们都一起求出了,所以杜教筛这一技巧实际上是在$O(n^{\frac{2}{3}})$内求解了$S(n)$,在所有形如$S(\lfloor\frac{n}{d}\rfloor)$共$O(\sqrt{n})$处的取值,因此总求解复杂度仍然是$O(n^{\frac{2}{3}})$的.

2.5 min_25筛

2.5.1 min_25筛简介

这是一个用于求解数论积性函数前缀和的方法,其复杂度并不优美,但实现简单且实际运行效率极高.它由日本算法竞赛选手min_25发明,2016年左右被引入中国,国内算法竞赛选手称其为min_25筛.

2.5.2 min_25筛适用条件

考虑求$S(n)=\sum_{i=1}^{n}f(i)$.其中$f(n)$是满足如下条件的数论函数.

- $f(p),p\in prime$,即在质数处的取值,是一个关于p的低次多项式$G(p)$(满足$deg(G(p))n=O(n)$时可以认为多项式$G(p)$是低次的).
- $f(p^c),p\in prime$,即在质数的幂处的取值,是可以$O(1)$求出,或者可以由$f(p^{c-1})$在$O(1)$内推出.
- $f(n)$是积性函数(更一般的n的不同质因子对于$f(n)$的贡献独立即可,即满足$gcd(a,b)=1\Rightarrow f(ab)=g(f(a),f(b))$,其中$g(a,b)$是是一个可以在$a,b$给定的情况下快速求值的函数).

2.5.3 min_25筛推导

首先考虑枚举i的最小质因子$mp(i)=p$以及它在i的标准分解形式种的指数$c=max\{b\geq 0\mid p^b\mid i\}$,将$p^c$从$i$种提取出来,显然剩余部分为1或者满足$mp(\frac{i}{p^c})>p$.

$$\sum_{i=1}^{n} f(i) = f(1) + \sum_{\substack{2 \le p^c \le n \\ p \in prime}} f(p^c)(1 + \sum_{\substack{2 \le x \le \lfloor \frac{n}{p^c} \rfloor \\ minprime_x > p}} f(x))$$

若x为合数则有.$mp(x) \le \sqrt{x} \le \sqrt{n}$;若$mp(x) > \sqrt{x}$则说明$x$为质数.于是我们按照$p$和$\sqrt{n}$的大小关系分类.

$$f(1) + \sum_{\substack{2 \le p^c \le \sqrt{n} \\ p \in prime}} f(p^c)(1 + \sum_{\substack{mp(x) > p \\ 2 \le x \le \lfloor \frac{n}{p^c} \rfloor}} f(x)) + \sum_{\substack{p \in prime \\ \sqrt{n} < p \le n}} f(p)$$

为了进一步化简进行计算,我们引入辅助函数$g_{n,m}$与h_n定义如下:

- $g_{n,m}$为n以内,最小质因子大于m的贡献.$g_{n,m} = \sum_{\substack{mp(x) > m \\ 2 \le x \le n}} f(x)$
- h_n为n以内素数对于$S(n)$的贡献.$h_n = \sum_{\substack{p \in prime \\ 2 < p \le n}} f(p)$

  根据g的定义由$\sum_{i=1}^{n} f(i) = S(n) = g_{n,1} + f(1)$尝试寻找$g$的递推式.

$$\begin{aligned} g_{n,m} &= \sum_{\substack{mp(x) > m \\ 2 \le x \le n}} f(x) \\ &= \sum_{\substack{p^c \le n \\ p \in prime \\ m < p \le \sqrt{n}}} f(p^c)(1 + \sum_{\substack{mp(x) > p \\ 2 \le x \le \lfloor \frac{n}{p^c} \rfloor}} f(x)) + \sum_{\substack{p \in prime \\ \sqrt{n} < p \le n}} f(p) \\ &= \sum_{\substack{p^c \le n \\ p \in prime \\ m < p \le \sqrt{n}}} f(p^c)(1 + g_{\lfloor \frac{n}{p^c} \rfloor, p}) + h_n - h_{\sqrt{n}} \end{aligned}$$

我们发现如果要根据这一递推式求解g,那么需要求出两类h_n.

一类是$h_{\sqrt{x}}$,显然$\sqrt{x} \le \sqrt{n}$可以$O(\sqrt{n})$预处理(之前的线性筛部分已经分析过若对于特定质数p,可以$O(logn)$求出$f(p)$,且$f(n)$为积性函数,则可以$O(n)$时间求出$f(i)\ i = 1, 2, 3, \dots n$).

另一类是形如$h_{\lfloor \frac{n}{d} \rfloor}$的$h_m$,这是由于$\lfloor \frac{\lfloor \frac{n}{i} \rfloor}{j} \rfloor = \lfloor \frac{n}{ij} \rfloor$,考虑$g_{n,m}$的递推式,不难发现求解$g_{n,1}$只会涉及形如$h_{\lfloor \frac{n}{d} \rfloor}$的,$h_m$,只有$O(\sqrt{n})$种.

如果能快速求出计算$g_{n,1}$种需要的所有h_m,那么利用递推式就能轻松求出g了.值得一提的是,递归求解g并不需要记忆化结果,即使使用了记忆化搜索也不能优化复杂度,将在后续对min_25筛复杂度分析的部分说明.

现在来考虑求解h_n

$$h_n=\sum_{\substack{p\in prime\\ 2<p\le n}} f(p)$$

$f(p)=G(p)$是一个低次多项式.$f(p)=\sum_{i=0}^{k} c_i p^i$.带入$h_n$的定义中,整理得到如下式子.

$$h_n=\sum_{\substack{p\in prime\\ 2<p\le n}} f(p)=\sum_{\substack{p\in prime\\ 2<p\le n}}\sum_{i=0}^{k} c_i p^i=\sum_{i=0}^{k} c_i\sum_{\substack{p\in prime\\ 2<p\le n}} p^i$$

观察结果,求h_n被转为求素数k次幂和.

定义辅助函数$L_{n,m}$.

$$L_{n,m}=\sum_{i=2}^{n}[(i\in prime)\,or\,(minprime_i>prime_m)]\ i^k$$

即$[1,n]$内的质数k次幂与,满足$mp(x)>prime_m$的数的k次幂和.或者换一个角度,$L_{n,m}$是Eratosthenes筛第m轮筛后没有被筛除的数的k次幂和.根据这个定义,有$L_{n,\sqrt{n}}=\sum_{p\in prime,p\le n} p^k$,因为任意一个$n$以内的合数$x$满足$mp(x)\le\sqrt{n}$.

考虑一个不断用素数筛除不满足约束的数的过程,考虑最小质因子为p_m的数$x=p_m\cdot y\le n\Rightarrow y\le\lfloor\frac{n}{p_m}\rfloor$,可以从$L_{n,m-1}$中扣除$p_m^k\,L_{\lfloor\frac{n}{p_m}\rfloor,m-1}$从而得到$L_{n,m}$.但是$p_m^k\,L_{\lfloor\frac{n}{p_m}\rfloor,m-1}$包含了形如$x=q\cdot p_m,x\le n$其中$mp(q)<p_m$的$x$的贡献,它们应该在之前被更小的质数排除.

再考虑这种q,必定是$q=p_i,p_i<p_m,p_i\in prime$的,不然$q$已经被筛掉了.所以补上一个$pre_{m-1}$,这样找到的$x=p_m\cdot y$就是我们本轮需要筛除的数了.

我们发现这样做扣除且仅扣除了所有满足$x\notin prime,x\le n,mp(x)=p_m$的$x$对于$L_{n,m-1}$的贡献.得到了$L_{n,m}$.形式化的说.

$$L_{n,m}=L_{n,m-1}-p_m^k(L_{\lfloor\frac{n}{p_m}\rfloor,m-1}-pre_{m-1})$$
$$pre_m=\sum_{i=1}^{m}(prime_i)^k$$

如果$p_m^2>n$了那么不需要计算了,直接令$L_{n,m}=L_{n,m-1}$理由如下.

设最小质因子不小于p_j的最小合数为x,显然$mp(x)\ge p_j$故$x\ge p_j^2>$n,这说明$L_{n,m-1}$中不包含这种$mp(x)=p_j\ x\notin prime$ 的x的贡献,即$L_{n,,m}=L_{n,m-1}$.另外$L_{n,1}=\sum_{i=2}^{n} i^k$

于是,结合利用递推式计算$g_{n,1}$,类筛法计算$L_{n,m}$,得到了这样一个完整的用于求解$S(n)$的算法.

1. 预处理$\sqrt{n}$以内的质数与$pre_x=\sum_{i=1}^{x}(prime_i)^k$.
2. 找到$g_{n,1}$需要的h_m.共$O(\sqrt{n})$项,初始令$h'(m)=L_{m,1}=\sum_{i=2}^{m} i^k$
3. 从小到达枚举不超过$\sqrt{n}$的质数$prime_j$,并从小到大枚举d对应的$\lfloor\frac{n}{d}\rfloor$的取值$m$,从$m=\lfloor\frac{n}{1}\rfloor=n$开始到$prime_j^2>m=\lfloor\frac{n}{d}\rfloor$为止,从$h'_m$中扣除$prime_j^k(h'_{\lfloor\frac{m}{prime_j}\rfloor}-pre_{m-1})$使得$h'_m=L_{m,j}$
4. 最后$\sum_{i=1}^{n} f(i)=g_{n,1}+f(1)$.根据$g$的递推式进入递归求解,这里不需要记忆化.

2.5.4 min_25筛复杂度分析

首先是类筛法求h,这一部分分析较为容易但是涉及到了对数积分,我们利用Wolfram Alpha做近似求解.

考虑一个需要求解的h_m,需要使用所有满足$prime_i \le \sqrt{m}$的$prime_i$进行筛除,所以计算一个h_m的时间为$\pi(\sqrt{m})$,其中$\pi(x)$是不超过x的素数个数.类似于杜教筛复杂度分析,我们考虑按照$m = \lfloor \frac{n}{d} \rfloor$和$\sqrt{n}$的大小关系分类进行计算.

$$\pi(n) = \sum_{x \le n} [x \in prime] = O(\frac{n}{log\, n})$$

$$T(n) = \sum_{i=1}^{\sqrt{n}} \pi(\sqrt{i}) + \sum_{i=1}^{\sqrt{n}} \pi(\sqrt{\frac{n}{i}})$$

$$\int_0^{\sqrt{n}} \frac{x}{log\, x} dx + \int_0^{\sqrt{n}} \frac{\sqrt{\frac{n}{x}}}{\log \sqrt{\frac{n}{x}}} = O(\frac{n^{\frac{3}{4}}}{log\, n})$$

而另一部分递归求解$g_{n,1}$的复杂度就难以分析了,我们查阅资料时发现国内算法竞赛选手朱震霆研究过这个问题.根据他的研究成果,计算$g_{n,1}$的复杂度是$O(n^{1-\epsilon})$的,是否记忆化g并不影响这部分的复杂度,其中ϵ为一大于0的小常数,并且发现当n较大(达到10^{13}以上)时,这个复杂度接近于线性增长,但n较小时,可以认为这一部分复杂度是$O(\frac{n^{\frac{3}{4}}}{logn})$的.

稍作总结,这是一个复杂度和线性筛法差不多,但是数据规模小时相较于线性筛法有极大优势的算法.它的理论价值并不高,但可以解决大多数算法竞赛中的数论函数求和问题.

- 类筛法求h_n的复杂度为$O(\frac{n^{\frac{3}{4}}}{log\, n})$,$n$越大这个上界越紧.
- 计算g这部分是的$O(n^{1-\epsilon})$近线性复杂度,不过$n \le 10^{13}$时可以认为是$O(\frac{n^{\frac{3}{4}}}{log\, n})$.
- 在$n \le 10^{13}$时,可以认为$T(n) = O(\frac{n^{\frac{3}{4}}}{logn})$.

2.5.5 min_25筛的实现.

这里给出一个可以使用min_25LOJ 6053

求$\sum_{i=1}^{n} f(i) \bmod (10^9 + 7)$,其中$n \le 10^{10}$,其中$f(n)$满足是一个数论积性函数,满足以下条件.

1. $f(1) = 1$
2. $f(p^c) = p \oplus c$ (p 为质数，$\oplus$ 表示按位异或).

首先$f(n)$在质数处取值是1次多项式;$n \le 10^{10} < 2^{64}$,所以求$f(p^c)$时,$p, c < 64$可以认为计算$p \oplus c$是$O(1)$的;$f(n)$是积性的,故可以使用min_25筛求$f(n)$的前缀和.我们直接给出c++实现.

```
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;
typedef unsigned long long UInt;
typedef long long Int;
```

```
const int N=300000+10;
const Int mod=(Int)(1e9)+7LL;
Int qpow(Int a,Int p){
    if(p==0) return 1;
    Int r=qpow(a,p>>1);
    r=r*r%mod;
    return (p&1)?(r*a%mod):r;
}
Int inv2;
UInt n;
int vis[N],prime[N],cnt;
Int pre0[N],pre1[N];
UInt qwq[N];
double SQRTN=0;
void init(){
    for(int i=2;i<N;i++){
        pre0[i]=pre0[i-1];
        pre1[i]=pre1[i-1];
        qwq[i]=qwq[i-1];
        if(!vis[i]){
            prime[++cnt]=i;
            pre0[i]++;
            pre1[i]=(pre1[i]+i)%mod;
        }
        for(int j=1;j<=cnt&&prime[j]*i<N;j++){
            vis[i*prime[j]]=1;
            if(i%prime[j]==0) break;
        }
    }
}
Int pre0A[N],pre1A[N];
Int pre0B[N],pre1B[N];
inline Int& at(UInt x,int y){
    if(x<=SQRTN) return y?pre1A[x]:pre0A[x];
    return y?pre1B[n/x]:pre0B[n/x];
}
inline Int geth(UInt x){
    Int ret=2*(x>=2);
    if(x<N) ret+=( (pre1[x]-pre0[x])%mod );
    else ret+=( (at(x,1)-at(x,0))%mod );
    return (ret%mod+mod)%mod;
}
Int g(UInt n,int m){
    m++; if(1ULL*prime[m]>n) return 0;
    Int s=0;
    while(m<=cnt&&1ULL*prime[m]*prime[m]<=n){
        UInt p=prime[m],pc=p,c=1;
        while(pc<=n){
            s=(s+((p^c)%mod)*(1+g(n/pc,m))%mod)%mod;
            c++;pc=pc*p;
        }
        m++;
    }
```

```
    Int tmp=((geth(n)-geth(prime[m-1]))%mod+mod)%mod;
    return (s+tmp)%mod;
}
inline Int s1(Int x){ x%=mod; return x*(x+1)%mod*inv2%mod; }
inline Int sub(Int a,Int b){
    a=(a%mod+mod)%mod;
    b=((-b)%mod+mod)%mod;
    return (a+b)%mod;
}
inline Int mul(Int a,Int b){
    a=(a%mod+mod)%mod;
    b=(b%mod+mod)%mod;
    return a*b%mod;
}
Int solve(){
    UInt l=1,q=0;
    while(l<=n){
        q=n/l; l=n/q+1;
        at(q,0)=(q-1);
        at(q,1)=(s1(q)+mod-1)%mod;
    }
    for(int i=1;i<=cnt&&1ULL*prime[i]*prime[i]<=n;i++){
        l=1;q=0;Int p=prime[i];
        while(l<=n){
            q=n/l; l=n/q+1;
            if(p*p>q) break;
            at(q,0)=sub(at(q,0),sub(at(q/p,0),pre0[p-1]));
            at(q,1)=sub(at(q,1),mul(p,sub(at(q/p,1),pre1[p-1])));
        }
    }
    Int ret=1+g(n,0);
    return ret%mod;
}

int main(){
    cin>>n; SQRTN=sqrt(n);
    inv2=qpow(2,mod-2);
    init();
    cout<<solve()<<endl;
    return 0;
}
```

这个程序有几个值得注意的地方,$10^{10}>2^{32}$,n^2超过了c++ long long类型(64位有符号整数最大为$2^{63}-1$)的最大值所以判断$m\le\sqrt{n}$时应当使用$n/m\ge m$而不是$m^2\le n$.而对于判断$prime_j$是否超过q可以直接用$prime_j^2>q$,因为这里用到的质数都是不超过$\sqrt{n}$的,不会发生溢出.这里$f(p)=p-1$是1次多项式,自然地$L_{n,1}=\sum_{i=2}^{n}i=\frac{n(n+1)}{2}-1$,对于较高次的情况,可以证明$S_k(n)=\sum_{i=1}^{n}i^k$是一个$k+1$次多项式,我们可以用拉格朗日插值法$O(k)$求出$S_k(n)=\sum_{i=0}^{k+1}a_ik^i$中的各项系数,进而求出$L_{n,1}$.此外,由于2在模$10^9+7$意义下存在逆元,我们使用费马小定理$0<a<p,a^{-1}\equiv a^{p-2}\pmod{p}$在$O(logp)$时间内可以求出2的逆元,显然$O(logp)$不超过$O(\sqrt{n})$并不会影响min_25筛的复杂度.

此外,这里并没有使用hash表(c++ STL中的unordered_map)存储h_n,而是使用了一个小技巧.对于h_m,若$m \le \sqrt{n}$,将它存储于A_m处,若$m > \sqrt{n}$则存储于$B_{\lfloor \frac{n}{m} \rfloor}$处,其中$A, B$是两个长度超过$\sqrt{n}$的数组.可以证明,这样存储不会发生冲突.从渐进复杂度角度看,hash表和静态数组都是可以$O(1)$存取的,但是实现中,hash表的存取涉及到很多的运算与判断,性能和静态数组差距很大(或者说,同样是$O(1)$,但常数不一样大).需要注意的是用数组代替hash表是一种常数优化而非复杂度优化.

3.参考文献

- wikipedia:Mertens' theorems
- wikipedia:Dirichlet convolution
- 复杂度分析:积性函数的狄利克雷卷积 by 李白天
- 《IOI2018 中国国家候选队论文集》:《一些特殊的数论函数求和问题》by 朱震霆
- 《2016 年信息学奥林匹克 中国国家队候选队员论文集》:《积性函数求和的几种方法》by 任之洲