GCD,LCM

最大公约数（GCD）

代码很简单，证明感兴趣可以看看，也不是很难。

求解a,b的最大公约数使用的算法是欧几里得算法，算法是这样子的：`gcd(a,b) = gcd(b,a mod b)`。

复杂度为$O(log\ n)$

证明：

$$假设a > b$$
$$a可以表示成a = kb + r（a，b，k，r皆为正整数，r = a\ mod\ b）$$
$$假设d是a,b的一个公约数，记作d|a,d|b，即a和b都可以被d整除。$$
$$而r = a - kb，两边同时除以d得\frac{r}{d} = \frac{a}{d} - k\frac{b}{d}$$
$$由等式右边可知m = \frac{r}{d}为整数，因此d|r$$
$$因此d也是b,a\ mod\ b的公约数。$$
$$假设d是b,r的一个公约数，记作d|b,d|r，即b和r都可以被d整除。$$
$$而a = kb + r，两边同时除以d得\frac{a}{d} = k\frac{b}{d} + \frac{r}{d}$$
$$由等式右边可知m = \frac{a}{d}为整数，因此d|a$$
$$因此d也是a,b的公约数。$$
$$因(a,b)和(b,a\ mod\ b)的公约数相等，则其最大公约数也相等。$$

代码：

```
int gcd(int a,int b){
    return b==0?a:gcd(b,a%b);
}
```

最小公倍数（LCM）

先给出公式：`lcm(a,b)=a*b/gcd(a,b)`

证明：

$$设d = gcd(a,b), a = k_1 * d, b = k_2 * d$$
$$lcm(a,b) = k_1 * d * t_1 = k_2 * d * t_2$$
$$并且k_1,k_2互质，t_1,t_2互质，所以，k_1 = t_2, k_2 = t_1$$
$$代入lcm = k_1 * d * t_1 = k_1 * k_2 * d = a * b/d$$

代码：

```
int lcm(int a,int b){
    //先做除法防止a*b数据溢出，后面也会用到类似的技巧
    return a/gcd(a,b)*b;
}
```

质数与质数筛

质数

质数的定义：质数是指只能被1和本身整除的正整数，最小的质数是2.

质数的判断：判断n是否位质数只需要从2到n-1判断是都可以整除n，当且仅当所有数都不可整除n时，n为质数。其实只需要从2到$\sqrt{n}$判断即可，可以假设$n = k_1 * k_2(k_1, k_2为整数)$，很容易得到$k_1, k_2$必定有一个小于等于$\sqrt{n}$，所以只需要在$[2, \sqrt{n}]$中判断就好。该算法的复杂度为$O(\sqrt{n})$。

代码：

```
bool isPrime(int x){
    if(x<2)
        return 0;
    for(int i=2;i<=x/i;i++)
        if(x%i==0)
            return 0;
    return 1;
}
```

质数筛

质数筛是用来求一个区间内所有的质数的算法。

埃式筛

埃式筛的思想时从小到大用每一个数去筛掉它所有的倍数，最后没有被筛掉的一定是质数。

比如求15内的所有质数：

1. 筛掉2的所有倍数，剩余2，3，5，7，9，11，13，15
2. 筛掉3的所有倍数，剩余2，3，5，7，11，13，15
3. 筛掉5的所有倍数，剩余2，3，5，7，11，13。已经全部是质数

埃式筛的复杂度为$O(n\ loglog\ n)$,证明较为复杂，用到了很多复杂的数学知识，这里就不证明了。

代码：

```
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n){
    for (int i = 2; i <= n; i ++ ){
        if (st[i]) continue;
        primes[cnt ++ ] = i;
        for (int j = i + i; j <= n; j += i)
            st[j] = true;
    }
}
```

线性筛

线性筛的复杂度为$O(n)$这也就是叫做线性筛的原因。

观察下埃式筛法，可以发现还是有很多重复计算的地方的，一个合数会被它的所有的质因子筛掉一遍，就比如6，会分别被2和3筛掉一次。而线性筛则保证每个合数只被它的最小质因子筛掉一次，这也是线性筛的核心思想。具体的思路是，每次枚举一个 i 时，从小到大枚举已经筛出来的质数，将primes[j]*i筛掉，直到`primes[j]*i>n或i%primes[j]==0`。下面我们来分析下算法为什么是正确的，以及为什么时线性的，也就是分析为什么每个合数必定被筛掉，且只会被最小质因子筛掉。

当枚举到 i 且 i 没有被筛掉时，我们分两种情况来讨论

1. `i%primes[j]==0:`因为我们是从小到大枚举所有的质数，且当`i%primes[j]==0:`我们就会跳出循环，所以此时primes[j]一定为 i 的最小质因子，primes[j]也一定为primes[j]*i的最小质因子。
2. `i%primes[j]! =0:`也是因为我们是从小到大枚举的所有质因子，此时还没有跳出循环，说明primes[j]还没有枚举到 i 的最小质因子，也就是说primes[j]<i的所有质因子，所以primes[j]也一定是primes[j]*i的最小质因子。
3. 综上每个合数都只会被最小质因子给筛掉一次，所以它为线性复杂度。
4. 当枚举到 i 时如果 i 为合数，则在[2,i-1]的范围内必定存在 i 的最小质因子，我们就可以把它筛掉，所以所有的合数都会被筛掉。

代码：

```
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];          // st[x]存储x是否被筛掉

void get_primes(int n){
    for (int i = 2; i <= n; i ++ ){
        if (!st[i]) primes[cnt ++ ] = i;
        //循环条件为primes[j]*i<=n，写成下面的形式是为了防止爆int
        for(int j=0;primes[j]<=n/i;j++){
            st[primes[j]*i]=true;
            if(i%primes[j]==0) break;
        }
    }
}
```

下面用一道例题给大家看下质数筛是干什么的吧。

题目描述

如题，给定一个范围 n，有 q 个询问，每次输出第 k小的素数。n <= 1e8,1≤q≤1e6，保证查询的素数不大于 n。

输入格式

第一行包含两个正整数 n,q，分别表示查询的范围和查询的个数。

接下来 q 行每行一个正整数 k，表示查询第 k 小的素数。

输出格式

输出 q 行，每行一个正整数表示答案。

输入输出样例

输入

```
100 5
1
2
3
4
5
```

输出

```
2
3
5
7
11
```

这个其实就是素数筛的一个模板题了吧。要查询一个区间内第k小的质数，思路肯定是先将区间内的所有素数筛出来打表，对于每一次查询直接输出就好。n给的很大，用埃氏筛绝对TLE，需要选择线性筛去筛素数。

代码：

```
#include<iostream>

using namespace std;

```

```cpp
const int N=1e8;
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;
        }
    }
}

int main(){
    int n,p,k;
    cin>>n>>p;
    get_primes(n);

    while(p--){
        scanf("%d",&k);
        printf("%d\n",primes[k-1]);
    }

    return 0;
}
```

高精度算法

高精度算法定义

在C++内置数据类型中，最大的数据类型为`unsigned long long`，其范围是`[0,2^64-1)`，约1e19的数据范围。当数据更大时不能继续使用内置数据类型进行表示，我们称其为高精度数，高精度算法就是用来处理高精度数的一类算法。

高精度算法的思想

高精度算法的原理很简单，代码实现稍微有点复杂。通常用一个数组来存储高精度数的每一位，手动模拟四则运算。高精度数读入时通常以字符串的形式读入，存储时通常用数组逆序存储。

高精度数读入与存储

算法思想

先以字符串形式读入，再逆序存储到数组中，因为读入到字符串中，数字的每一位都是以字符的形式存储的，所以放入到数组中时应先`-'0'`，否则数组中存的将不是数字，而是当前字符对应的ASCII码。

代码

```cpp
string s;
vector<int> a;
cin>>s; //高精度数读入
for(int i=s.size()-1;i>=0;i--)
    a.push_back(s[i]-'0')   //将数字逆序存储
```

前导0的删除

什么是前导0

在高精度运算的过程中因为各种各样的原因，可能会在计算结果的最前面多出一些0，这些就称为前导0。如0112，00等等。单独的数字0不算前导0。

代码

```
//vector<int> C
while(C.size()>1&&C.bakc()==0)
    C.pop_back();
```

高精度加法

算法思想

先回想下加法的运算过程：首先要将两个数的最低位对齐，然后从低位到高位运算算，将对应位相加，相加结果超出10就进位。高精度加法其实就是模拟的这个过程。

代码

```
vector<int> add(<vector<int> &A,vector<int> &B>){
    for(int i=0,t=0;i<A.size()||i<B.size()||t;i++){ //t表示A[i],B[i],进位的加和
        if(i<A.size()) t+=A[i];
        if(i<B.size()) t+=B[i];
        C.push_back(t%10);
        t/=10;
    }
    while(C.size()>1&&C.back()==0)  //删除前导0
        C.pop_back();
    return C;
}
```

高精度减法

算法思想

减法运算时需要先考虑两数的大小，以`c=a-b`为例:

```
1. 当a>=b时计算a-b，
2. 当a<b时计算`-(b-a)`
3. 计算时从低位到高位计算，两数对应位相减，如果`a[i]<b[i]`则需要借位。
```

代码

```
//判断A>=B?
bool cmp(vector<int> &A,vector<int> &B){
    if(A.size()!=B.size())
        return A.size()>B.size();
    for(int i=A.size()-1;i>=0;i--)
        if(A[i]!=B[i])
            return A[i]>B[i];
    return true;
}
//A>=B时，计算C=A-B
//t同时表示运算结果和借位
vector<int> sub(vector<int> &A,vector<int> &B){
    vector<int> C;
    for(int i=0,t=0;i<A.size();i++){
        t=A[i]-t;   //处理借位,并将处理后的结果存入t中
        if(i<B.size())
            t-=B[i];    //做减法
        C.push_back((t+10)%10);     //存数
        t=t<0?1:0;  //判断是否借位
    }
    while(C.size()>1&&C.back()==0)//删除前导零
        C.pop_back();
    return C;
}
```

上面的减法函数有些地方可能不是很好理解，可以举个例子来看下。如计算1632-971

1. 逆序存入数组：A={2 , 3 , 6 , 1} , B={1 , 7 , 9}
2. 计算：
    1. `t=A[0]-t=2,t=t-B[0]=1,C[0]=1,t=0;`
    2. `t=A[1]-t=3,t=t-B[1]=-4,C[1]=6,t=1;`
    3. `t=A[2]-t=5,t=t-B[2]=-4,C[2]=6,t=1;`
    4. `t=A[3]-t=0,C[3]=0//此时t=B.size()故不做减法运算`
3. 删除前导0：

    此时的C={1 , 6 , 6 , 0}，最高位为0，这就是前导0的一种产生情况，需要将前导0删除。

高精乘低精

算法思想

与加法思路类似，将高精度数的每一位与低精度数做乘法，如果超出10则进位。处理过程比加法还要少一步，理解了高精度加法，这个代码理解起来就不是很困难了

代码

```
//C=A*b
vector<int> mul(vector<int> &A,int b){
    vector<int> C;
    for(int i=0,t=0;i<A.size()||t;i++){
        if(i<A.size())
            t+=A[i]*b;
        C.push_back(t%10);
        t/=10;
    }
    while(C.size()>1&&C.back()==0)
        C.pop_back();
    return C;
}
```

高精乘高精

算法思想

这个算法需要做点思考，先来看一个多项式乘法的例子

$$A=a_0+a_1x+a_2x^2$$

$$B=b_0+b_1x+b_2x^2$$

$$C=A*B=a_0b_0+(a_0b_1+a_1b_0)x+(a_0b_2+a_1b_1+a_2b_0)x^2+(a_1b_2+a_2b_1)x^3+a_2b_2x^4$$

可以发现交叉相乘的a,b的下标之和与对应的x的指数相等，高精乘高精的过程就是将两个数看作多项式处理的利用的是如下性质：

A与B相乘时，A[i]与B[j]相乘结果落在C[i+j]上，如123*456

$$A=3+2*10+1*10^2$$

$$B=6+5*10+4*10^2$$

$$C=18+(3*5+2*6)*10+(3*4+2*5+1*6)*10^2+(2*4+1*5)*10^3+4*10^4=56088$$

代码

```
//C=A*B
//高精乘高精用STL实现会很不方便，这里给出用数组实现的代码
int a[N],b[N],c[2N];
void mul(){
    string a1,b1;
    cin>>a1>>b1;
    int la=a1.size()
```

```
    int lb=b1.size();
    int len=la+lb;
    for(int i=la-1;i>=0;i--) a[la-i-1]=a1[i]-'0';
    for(int i=lb-1;i>=0;i--) b[lb-i-1]=b1[i]-'0';
    for(int i=0;i<la;i++)
        for(int j=0;j<lb;j++){
            c[i+j]+=a[i]*b[j];
            c[i+j+1]+=c[i+j]/10;
            c[i+j]%=10;
        }
    while(c[len-1]==0&&len>1)    //删除前导0
        len--;
}
```

高精度除法

算法思想

回想下除法的运算过程，除法时有两个计算结果，一个是商，一个是余数，计算时从高位到地位运算。因为有两个运算结果，但函数不能返回两个值，所以将余数用引用的方式传参。

代码

```
//C=A/b, r=A%b
vector<int> div(vector<int> &A,int b,int &r){
    vector<int> C;
    r=0;
    for(int i=A.size()-1;i>=0;i--){
        r=r*10+A[i];
        C.push_back(r/b);
        r%=b;
    }
    //注意，C数组中存的值是从高位到低位存储的，为了方便删除前导0，故需要将C数组反转一下
    reverse(C.begin(),C.end());
    while(C.size()>1&&C.back()==0)
        C.pop_back();
    return C;
}
```

一个完整的高精度题目的代码

减法的完整代码最为复杂，这里给出减法的完整代码，其余的代码与减法类似

题目描述

给定两个正整数（不含前导 0），计算它们的差，计算结果可能为负数。

输入格式

共两行，每行包含一个整数。

输出格式

共一行，包含所求的差。

数据范围

1≤整数长度≤1e5

输入样例：

```
32
11
```

输出样例：

```
2
```

代码

```cpp
#include<iostream>
#include<vector>
#include<string>

using namespace std;

//判断A>=B?
bool cmp(vector<int>&A,vector<int>B){
    if(A.size()!=B.size())
        return A.size()>B.size();
    for(int i=A.size()-1;i>=0;i--)
        if(A[i]!=B[i])
            return A[i]>B[i];
    return true;
}

//计算A-B
vector<int> sub(vector<int>&A,vector<int>&B){
    vector<int>C;
    for(int i=0,t=0;i<A.size();i++){
        t=A[i]-t;
        if(i<B.size())
            t-=B[i];
        C.push_back((t+10)%10);
        t=t<0?1:0;
    }
    while(C.size()>1&&C.back()==0)
        C.pop_back();
    return C;
}

int main(){
    string a,b;
    vector<int>A,B,C;
    cin>>a>>b;

    //逆序存储，注意类型间的转换
    for(int i=a.size()-1;i>=0;i--)
        A.push_back(a[i]-'0');
    for(int i=b.size()-1;i>=0;i--)
        B.push_back(b[i]-'0');

    //计算
    if(cmp(A,B))
        C=sub(A,B);
    else{
        cout<<'-';
        C=sub(B,A);
    }

    //从高位到低位以此输出
    for(int i=C.size()-1;i>=0;i--)
        printf("%d",C[i]);

```

```
    return 0;
}
```

快速幂

顾名思义，快速幂是一种快速求幂的算法，计算a^n用暴力算法需要O(n)的时间，用快速幂可以在O(log n)的时间内完成。

朴素快速幂

朴素快速幂是基于分治思想来实现的，先考虑怎么样可以比较快的计算出2^18，可以这样子来计算。

$$2^{18} = 2^9 * 2^9$$
$$2^9 = 2^4 * 2^4 * 2$$
$$2^4 = 2^2 * 2^2$$
$$2^2 = 2 * 2$$

可以通过四步，即O(log n)的时间计算出2^18的值计算过程为

1. 如果n==0则返回1
2. 如果n为奇数则

$$a^n = a^{\frac{n}{2}} * a^{\frac{n}{2}} * a$$

3. 如果n为偶数则

$$a^n = a^{\frac{n}{2}} * a^{\frac{n}{2}}$$

代码：

```
typedef long long ll;
//正确的代码
ll pow(ll a,ll n){
    if(n==0)
        return 1;
    ll tmp=pow(a,n/2);
    if(n%2==0)
        return tmp*tmp;
    else
        return tmp*tmp*a;
}

//错误的代码
ll pow(ll a,ll n){
    if(n==0)
        return 1;
    if(n%2==0)
        return pow(a,n/2)*pow(a,n/2);
    else
        return pow(a,n/2)*pow(a,n/2)*a;
}
```

注意：代码切不可写成第二段那种形式，可以来对比下两段代码，看起来差别并不是很大，其实第二段代码的复杂度为O(n)，原因是计算了两次pow(a,n/2)，二第一段代码只计算了一次。

位运算优化快速幂

其实朴素快速幂计算已经很快了，但是有种更好更简洁的 代码，常用的代码也是这个。

位运算优化快速幂是通过指数的二进制进行优化的，继续看一个例子，计算2^22

$$22 = 10110_{(2)}$$
$$2^{22} = 2^{16+4+2} = 2^{16} * 2^4 * 2^2$$

观察归纳下可以给出个一般的表达式

$$a^b=\prod_{i=0}^{log_2^b}k*a^{2^i}$$

$$k=b\text{的二进制的第}i\text{位}$$

$$a^{2^i}=a^{2^{i-1}}*a^{2^{i-1}}$$

所以就可以通过a自乘来获取a^{2^i}的值，循环处理b的二进制的每一位，$a*=a$，当前位是0时，则$continue$,当前位是1时,则$ans*=a$

代码：

```
//c=a^b
typedef long long ll;
ll pow(ll a, ll b){
    int c=1;
    while(b>){
        if(b&1)
            c*=a;
        a*=a;
        b>>=1;
    }
    return c;
}
```

快速幂取模

快速幂取模是计算a^b%p的值，先来了解几个模运算恒等式

$$(a+b)\ mod\ p=(a\ mod\ p+b\ mod\ p)\ mod\ p$$

$$(a-b)\ mod\ p=(a\ mod\ p-b\ mod\ p)\ mod\ p$$

$$(a*b)\ mod\ p=(a\ mod\ p*b\ mod\ p)\ mod\ p$$

$$a^b\ mod\ p=(a\ mod\ p)^b\ mod\ p$$

注意上述恒等式中没有除法的恒等式，对于除法上述的恒等式不成立

下面来推理以下a^b%p的计算

$$a^b=\prod_{i=0}^{log_2^b}k*a^{2^i}$$

$$a^b\ mod\ p=(\prod_{i=0}^{log_2^b}k*a^{2^i})\ mod\ p=(\prod_{i=0}^{log_2^b}k*a^{2^i}\ mod\ p)\ mod\ p$$

代码：

```
//c=a^b%p
typedef long long ll;
ll pow(ll a,ll b,ll p){
    ll c=1%p;
    while(b){
        if(b&1)
            c=c*a%p;
        a=a*a%p;
        b>>=1;
    }
    return c;
}
```