算法基础（十六）： 数学基础 - 数论1 - 质数

质数（素数）

质数：一个大于`1`的自然数，除了`1`和它自身外，不能被其他自然数整除的数叫做质数；否则称为合数，所有小于`1`的数既不是质数也不是合数，**也就是说质数的因数（约数）只有`1`和它本身**

整除：若整数`b`除以非零整数`a`，商为整数，且余数为零，`b`为被除数，`a`为除数，即`a|b`（“`|`”是整除符号），读作“`a`整除`b`”或“`b`能被`a`整除”

自然数：`0 1 2 ....`

判断素数（式除法）

暴力解法：

```
bool is_prime(int n)
{
    //先判断是否大于1
    if(n < 2) return false;
    //再看它是否包含处理1和自己外的其他因数
    for(int i = 2; i < n; i ++)
    {
        //如果有一个数可以整除n, 表示n不是质数
        if(n % i == 0) return false;
    }
    return true;
}
```

算法的时间复杂度是`O(n)`，这个复杂度其实还是蛮高的

优化版本：

我们注意到一个数`n`

如果`d`能整除`n`，即`d | n`，那么显然`(n / d) | n`，即`n`除以`d`的商也可以整除`n`

比如`6`可以整除`12`，那么`12`除以`6`得到的`2`也可以整除`12`

我们可以发现一个数的约数总是成对的，那么我们在枚举的时候可以只枚举每对中的较小的那一个数即可，比如上面的数`12`，我们只用枚举`2`即可，`6`其实是多余的，当不存在大于`2`的较小的约数时当然也就不存在大于`2`的较大的约数，当然也就不存在大于`2`的约数了，也就是质数

因为较小的约数`d`一定满足`d <= (n / d)`，所以我们只用枚举所有满足这个条件的数即可，注意这里我们得加上等号，因为这一对数是可能相等的比如：`3 * 3 = 9`

于是我们就可以将算法的时间复杂度降到了`O(sqrt(n))`，这是一个质的飞跃

```
bool is_prime(int n)
{
    //先判断是否大于1
    if(n < 2) return false;
    //再看它是否包含处理1和自己外的其他因数
    for(int i = 2; i <= n / i; i ++)
    {
        //如果有一个数可以整除n，表示n不是质数
        if(n % i == 0) return false;
    }
    return true;
}
```

注意：

1. `i <= n / i`不要写成`i <= sqrt(n)`，因为`sqrt()`函数的复杂度大于`O(sqrt(n))`，实际的复杂度要高一些
2. `i <= n / i`不要写成`i * i <= n`，因为`int`的最大值是`2147483647`当`i`接近于这个数的时候`i * i`会溢出变成一个负数，出现奇怪的错误

分解质因数（式除法）

素因数（或称素因子）在数论里是指能整除给定正整数的素数。**根据算术基本定理，不考虑排列顺序的情况下，每个正整数都能够以唯一的方式表示成它的素因数的乘积。**两个没有共同素因子的正整数称为互素。因为`1`没有素因子，与任何正整数（包括`1`本身）都是互素。只有一个素因子的正整数为素数。

将一个正整数表示成素因数乘积的过程和得到的表示结果叫做素因数分解。显示素因数分解结果时，如果其中某个素因数出现了不止一次，可以用幂次的形式表示。例如`360`的素因数分解是：

$$360 = 2 * 2 * 2 * 3 * 3 * 5 = 2^3 * 3^2 * 5$$

其中的素因数`2、3、5`在`360`的素因数分解中的幂次分别是`3，2，1`

我们分解质因数就是为了得到一个数的因数以及因数的幂

暴力法：

```
void divide(int n)
{
    //从小到大枚举所有的数
    //当出现一个因数的时候就进入while循环
    //在while循环中将这个因数从n剔除掉，并改变n的大小
    //只有当n等于1的时候才会满足i > n这个条件，退出for循环，这也是结束的时刻
    //在其他时候n进入while循环剔除当前因子i之后得到的新的n必然会比i大
    //因为得到的新的n是由大于i的因数相乘得到的
    for(int i = 2; i <= n; i ++)
    {
        //如果i是n的约数
        if(n % i == 0)
        {
```

```
            //开始求i的次数
            int s = 0;

            while(n % i == 0)
            {
                n /= i;
                s ++;
            }
            printf("%d%d/n", i, s);
        }
    }
}
```

我们需要思考几个问题：

1. **为什么那个`while`循环可以求出这个约数的次数?**

我们以上面的`360`为例，`2`显然是`360`的约数，当`360`进入`while`循环之后，每循环一次，`360`就会除以一个`2`，表示从`360`中剔除一个因子`2`当将`2`剔除`3`次后，`360`变成`3 * 3 * 5 = 45`跳出`while`循环，也就求出了因子`2`的次数，同时我们可以看到，当`360`剔除所有的`2`之后变成`3 * 3 * 5 = 45`，`45`的质因数刚好就是`360`剩下的质因数，所以我们继续枚举就可以一步步将`360`的质因数求出来

2. **我们是从小到大枚举，为什么不会出现合数满足条件：`if(n % i == 0)`?**

当枚举到`i`的时候，由于`while`循环所以从`2 ~ i - 1`这个范围的质因子都已经从原来这个数`n`中剔除掉了，于是得到了一个不包含`2 ~ i - 1`这个范围的质因子的新数`n`

如果此时`n % i == 0`，而`n`又不包含`2 ~ i - 1`这个范围的质因子，所以`i`也就不包含`2 ~ i - 1`这个范围的质因子，因为`i`如果包含这个范围内的质因子，而`i`又可以整除`n`，就说明此时的`n`也包含这个范围内的质因子，矛盾，所以`i`也就不包含`2 ~ i - 1`这个范围的质因子，而任何一个整数都能够以唯一的方式表示成它的素因数的乘积，而`i`也就一种表示方式：`i = i^1`，`i`必然是素数

最开始`i`从素数`2`开始，满足初始条件从素数开始，所以后面的`i`也就全是素数了（类似于数学归纳思想）

直观上来讲，因为`n`在循环的过程中是会发生变化的，只要出现合数因子他就会一直除干净，所以`i`也就不可能出现合数了

3. **时间复杂度**

当`n`为质数的时候时间复杂度最坏是`O(n)`

4. **假如`n = k * k * k * l * l* l`为什么不会出现将`k`剔除完后`i > n`的情况，也就是说为啥改变`n`之后`i`的可取值范围也会减小，但是`i`仍然会取到所有的`n`的因数?**

这个问题也就是算法的退出情况

注意我们在剔除质因子的时候一定是从最小的质因子开始剔除的，也就是说`k < l`，当`i`取到`k`我们将其剔除完了之后剩下的`l * l * l`新的必然会大于当前的`i == k`，而且大于等于`l`，所以虽然`n`减小，`i`的可取值范围减小，但是并不会遗漏任何一个因子，**也就是说不会出现`n`缩小到小于某个因子从而导致`i`取不到这个因子的情况**（虽然像是一句废话。。）只有最后`n`继续剔除`l`一直到`n = 1`的时候才会退出`for`循环，当然此时也就找完了质因子，也就是算法结束的条件

优化版本：

算术基本定理：任何一个大于`1`的自然数`N`，如果`N`不为质数那么`N`可以唯一分解成有限个质数的乘积$N = p_1^{a_1} * p_2^{a_2} \dots * p_n^{a_n}$且最多只有一个大于$\sqrt{n}$的质因子，这里`P1<P2<P3.....<Pn`均为质数，其中指数`ai`是正整数，如果`N`是质数的话`N`就是自己本身`N = N ^ 1`

假设有两个大于`sqrt(N)`的质因子，显然乘一起大于`N`是矛盾的

代码如下：

```
void divide(int n)
{
    //从小到大先将小于等于sqrt(n)的质因子枚举完
    //对于不含大于sqrt(n)质因子的数，我们从前往后遍历小于sqrt(n)的质因子
    //最后n一定是等于1退出循环
    //只有当n包含大于sqrt(n)的质因子，我们从前往后将小于原来的sqrt(n)的质因子剔除完了最后留下的就是大于原来sqrt(n)的这个数本身
    //注意此处带等号，i是可以等于n / i的，比如：9 = 3 * 3
    for(int i = 2; i <= n / i; i ++)
    {
        //如果i是n的约数
        if(n % i == 0)
        {
            //开始求i的次数
            int s = 0;
            while(n % i == 0)
            {
                n /= i;
                s ++;
            }
            printf("%d %d\n", i, s);
        }
    }

    //如果没有大于原来sqrt(n)的的质因子，那么n最后一定会为1
    //如果不为1，那么n就是它最后的那个大于原来的sqrt(n)的质因子
    if(n > 1) printf("%d %d\n", n, 1);
    puts("");
}
```

从而将时间复杂度有效地降低到`O(sqrt(n))`，但是我们需要注意的是这里的时间复杂度是最多`O(sqrt(n))`，当`n = 2 ^ k`的时候我们进入`while`循环之后会直接就除干净了，复杂度是`O(k)`

需要注意一个问题，跟上面类似，当`n`不含大于`sqrt(n)`的质因子时，当`n`的值改变之后`i`可以取到的值的范围比上面的朴素版本减少更严重，是否会出现`i`取不到部分因子的情况？

比如`n = k...k * l...l * s..s`

我们注意到，当剔除掉`k`之后`n = l...l * s...s`，此时取不到部分因子指的是，`n`的因子`l`，`s`大于`sqrt(n)`，导致`i`不可能再递增到`l`和`s`了，会不会出现这种情况呢？

其实是会的，我们假设出现了这样一种情况，此时`n`**中只可能有一个因子大于**`sqrt(n)`**，也就是只可能是因子**`s`**，且**`s`**只可能有一个**，所以`l`此时仍然可以被取到，当剔除掉`l`之后留下`s`退出循环，进入`if(n > 1)`，输出最后一个因子，算法正确结束

综上我们总结：

1. **如果本身`n`就含有大于`sqrt(n)`的因子，`n = ll..kk.tt..ls`**

当我们剔除掉某个因子之后`n = kk..tt..ls`由于`s`大于原来的`sqrt(n)`，所以`s`前面的这些因子必然不会大于现在的`sqrt(n)`，因为一个整数只能有一个大于`sqrt(n)`的因子，在`n`缩小的过程中这个因子始终是`s`，于是`i`在递增过程中始终可以取到`s`前面的那些因子，最后剩下`s`退出循环，算法正确

2. **如果本身`n`不含有大于`sqrt(n)`的因子，`n = lll..lkkkk...ssss`**

如果在剔除某个因子之后`n = kkkk..ttt..sss`使得剩下的因子中含有大于新的`sqrt(n)`的因子，这时这个因子只有一个且只可能为最后一个，这时回到了第一种情况，最后剩下`s`退出循环，算法正确

3. **如果本身`n`不含有大于`sqrt(n)`的因子，`n = lll..lkkkk...ssss`**

如果在剔除某个因子之后`n = kkkk..ttt..sss`使得剩下的因子中始终不会出现大于新的`sqrt(n)`的因子，这时`i`会一直取到所有的因子，我们一直缩小`n`，剔除它的所有因子，`n = 1`之后退出循环，算法正确

另外关于为什么合数不会进入`if(n % i == 0)`解释和朴素版本类似，这里不再赘述

筛质数（朴素筛）

对于一组数字，比如，一共有`2 ~ p`

我们在这组数中先删掉`2`的倍数的数，再删掉`3`的倍数的数，再删掉`4`的倍数的数，再删掉`5`的倍数的数，一直删掉`p - 1`的倍数的数，由于剩下的数没有被删掉，也就不含从`2 - (p-1)`的因子，也就必然是质数

代码：

```
void get_prime(int n)
{
    //从2到最后的那个数进行枚举
    for(int i = 2; i <= n; i ++)
    {
        //如果这个数没有被前面的数筛过，说明就是一个质数
        if(!st[i]) prime[cnt ++] = i;

        //再将剩下的数中所有是i的倍数的数删掉
        for(int j = i + i; j <= n; j += i) st[j] = true;
    }
}
```

当进行到第`i`个的时候，前面`2 ~ i-1`个数已经筛完了，如果此时`i`没有被筛掉，那就说明`i`不含`2 ~ i-1`的因子，也就说明`i`是质数，然后用`i`继续筛后面的数

时间复杂度：

外层循环`n`次，每次外循环的时候，内层循环`n / i`次

也就是`n/2 + n/3 + n/4 + ... + n/n = n(1/2 + 1/3 + 1/4 +...1/n) = nlnn < nlogn(以2为底)`

时间复杂度是`O(nlnn)`

筛质数（埃氏筛）

我们注意到，当进行到`p`的时候，若`p`不含`2 ~ p-1`的因子，`p`当然是一个质数，但是如果`p`不含`2 ~ p-1`中的质因子，`p`当然也是一个质数，因为如果`p`是合数的话根据算术的基本定理`p`一定包含`2 ~ p-1`的质因子，由逆否命题，不含`2 ~ p-1`的质因子，所以`p`一定是质数

所以我们只用`2 ~ p-1`中的质数去筛即可，不用全部去筛

代码如下：

```
void get_prime(int n)
{
    //从2到最后的那个数进行枚举
    for(int i = 2; i <= n; i ++)
    {
        //如果这个数没有被前面的数筛过，说明就是一个质数
        if(!st[i])
        {
            prime[cnt ++] = i;
            //再将剩下的数中所有是i（质数）的倍数的数删掉
            for(int j = i + i; j <= n; j += i) st[j] = true;
        }

    }
}
```

当外层运行到`i`的时候，如果`i`没有被筛掉，说明`2 ~ i-1`没有`i`的质因数，说明`i`是质数，然后再用`i`去筛后面的数，依次递推下去（归纳递推）

最开始从`2`开始，`2`是质数（归纳奠基）

故算法正确

时间复杂度：

朴素情况下，外层循环`n`次，每次外循环的时候，内层循环`n / i`次时间复杂度是`n/2 + n/3 + n/4 + ... + n/n = n(1/2 + 1/3 + 1/4 +...1/n) = nlnn`

需要注意的是，只有质数才会进入内层的循环，我们只需要加上内部质数`i`的`n/i`

`1 ~ n`中大约有`n/lnn`个质数，简单来看可以直接用`n/lnn`去乘调和因子，也就是得到`O(n)`

真实的时间复杂度是`O(nloglogn)`，近似看作`O(n)`

筛质数（线性筛）

核心：`n`只会被最小质因子筛掉

这个算法是真的逆天

代码如下：

```
void get_prime(int n)
{
    for(int i = 2; i <= n; i ++)
    {
        //如果是一个质数，将其加入到质数表里面
        if(!st[i]) primes[cnt ++] = i;

        //从小到大枚举所有的质数
        //primes[j] <= n / i，算法不会筛掉大于n的合数，这样就没意义了
        for(int j = 0; primes[j] <= n / i; j ++)
        {
            st[primes[j] * i] = true;
            if(i % primes[j] == 0) break;
        }

    }
}
```

1. **为什么一个合数 `n` 只会被自己的最小质因子筛掉?**

也就是`st[primes[j] * i] = true;`总是用这个合数的最小质因子筛掉

我们注意到这样一行代码：`if(i % primes[j] == 0) break;`

当外层是`i`，然后内层开始枚举质数的时候有两种情况：

1. `i % primes[j] != 0`，`primes[j]`不能整除`i`，由于`pj`是从小到大枚举的，`pj`不能整除`i`，也就说明`pj`小于`i`的最小质因子，那么`pj`一定就是合数`pj * i`的最小质因子，从而利用最小质因子筛掉合数
2. 当枚举质数到`i % pj == 0`的时候，此时`pj`一定是`i`的最小质因子，因为我们是从小到大枚举的，`i`一定有一个最小质因子，当`i % pj == 0`的时候`pj`必然是`i`的最小质因子，那么`pj`同时也是`pj * i`的最小质因子，从而利用最小质因子筛掉合数

3. **为什么一个合数一定会被筛掉呢?**

假设这个合数是`l = p * i`，其中`p`是最小质因子，`i`是另一个因子，`p <= i <= l`

当外层循环到`i`的时候，开始从小到大枚举质数，所以就一定可以将其筛掉

3. **为什么这个合数只会被筛一次呢?**

因为当一个合数写作`p * i`的时候，其中`p`是最小质数，这种写法只有一次，最小质数只有一个，那么`i`也就只有一个，当循环到`i`的时候开始枚举就可以筛掉它，在后面的循环中，`i`递增，由于只有一个最小质因子，当枚举到`p`的时候`p * i`不会筛掉它，`p`往后枚举的时候也就不会再重新筛掉它

4. **为什么外层循环到`i`的时候`2 ~ i`中的合数一定会被筛掉?**

当`i = 4`的时候，会被`2 * 2`筛掉(归纳奠基)

假设外层循环到`i`的时候`2 ~ i`中的合数一定会被筛掉（归纳假设）

当到`i + 1`的时候（归纳递推）

如果`i + 1`是质数，它不会被筛掉

如果`i + 1`是合数，那么必然可以写成`p * j`这样的最小质数乘`j`形式，`p`和`j`都是小于`i + 1`的数，由于`2 ~ i`中的合数都被筛掉了，在前面的循环中必然外层循环到了`i = j`这个情况，并且遍历质数的时候也一定会遍历到`p`，所以`i + 1`也会被筛掉，所以`2 ~ i+1`中的合数也会全部被筛掉

5. **为什么`if(i % primes[j] == 0) break;`需要`break`呢?**

当我们枚举到`i % pj == 0`的时候前面筛合数的时候一定是用最小的质因子去筛，若此时我们继续枚举`pj`，虽然也可以筛掉`pj * i`这个合数，但是不能保证`pj`是最小合数了，一个合数有可能被筛多次，复杂度提高

当`n = 13`，`i = 6`的时候，遍历到`pj = 2`，当筛掉`12`之后，`2`已经是`6`的最小质因子，再往后筛的话，不会再是用最小质因子来筛，会出现重复筛的情况，所以此时退出循环

6. **时间复杂度为什么是`O(n)`**

虽然有两重循环，但是我们以这`n`个数来考虑，如果是合数则语句`st[primes[j] * i] = true`只会被执行一次，如果是质数，则`primes[cnt ++ ] = i`只会被执行一次，所以一共关键语句只执行了`n`次，所以时间复杂度是`O(n)`

7. **为什么在枚举质数的时候不用加上`j <= cnt`这个限定呢?**

`i`是质数时，`primes[]`的最后一个元素一定是`i`满足`for`循环的跳出条件`if(i % primes[j] == 0) break;`，`j`不会大于`cnt`
`i`是合数时，`i`一定可以写成`p * j`这种形式，其中`p`是最小质因子，且`p`在`2 ~ i`之间，而`2 ~ i`之间的质数都已经被筛出来了，所以`primes[]`中一定含有`i`的最小质因子`primes[j]`，也满足`for`循环的跳出条件`if(i % primes[j] == 0) break;`，`j`不会越界

上面说明了`j`始终不会大于`cnt`，关于`primes[j] <= n / i`这个条件，需要说明一下

当没有这个条件的时候，若`i`是质数则`for`循环也会退出，此时`j = cnt`，若`i`是合数则当枚举到`i`的最小质因子的时候`for`循环也会退出，所以这个条件不是用来避免无限递归的

这个条件的目的是少筛掉一些合数，注意到我们用最小质数去筛合数的时候`prime[j] * i`是有可能大于`n`，比如`n = 12`，`i = 11`，`prime[j] = 2`的时候再筛就没有意义了，因为我们只用计算小于等于`n`的质数的个数

我们在筛质数的时候当筛掉的合数小于等于`n`的时候，一定不会被这个条件漏掉（显然）