Finding Divisors,
Number of Divisors
and
Summation of Divisors

by

Jane Alam Jan

Finding Divisors

If we observe the property of the numbers, we will find some interesting facts. Suppose we have a number 36.

36 is divisible by 1, 2, 3, 4, 6, 9, 12, 18 and 36. It is clear that if *a* is divisible by *b*, then *a* is also divisible by *(a/b*). That means, 36 is divisible by 3, so, this property helps us to find that 36 will also be divisible by 12 *(36/3)*. Now we will make two lists, if *a* is divisible by *b*, then we will take *b* into the first list and *(a/b)* into the second list.

So, for 36, we will evaluate our idea.

36 is divisible by 1. So, it will be added in list 1, and (36/1) = 36 will be added in list 2.

```
List 1: 1

List 2: 36
```

36 is divisible by 2. So, it will be added in list 1, and (36/2) = 18 will be added in list 2.

```
List 1: 1 2

List 2: 36 18
```

36 is divisible by 3. So, it will be added in list 1, and (36/3) = 12 will be added in list 2.

```
List 1: 1 2 3

List 2: 36 18 12
```

36 is divisible by 4. So, it will be added in list 1, and (36/4) = 9 will be added in list 2.

```
List 1: 1 2 3 4

List 2: 36 18 12 9
```

36 is divisible by 6. So, it will be added in list 1, and (36/6) = 6 will be added in list 2.

```
List 1: 1 2 3 4 6

List 2: 36 18 12 9 6
```

Now, see that if we move further we will find the numbers again (some numbers from list 2, which will try to insert into list 1) So, it is clear that, if we have list 1, we can generate list 2 (from this position).

So, we can stop checking if we find *b*, for which *a/b* is less than or equal to *b*. If we try to divide by numbers greater than *b*, we would actually repeat the same steps. For 36, (check the lists) the next number that will be included to list 1 is 9, but 36/9 = 4, which is already in list 1.

Now we have to find *b* for which *a/b* is less than or equal to *b*. If we think mathematically,

$b >= a/b$

$or, b^2 >= a$

$or, b >= sqrt(a)$

$so, b_{minimum} = sqrt(a)$

So, we have already found an idea to find divisors of a number in complexity - **sqrt(N)**. And of course, we can count them to find the number of divisors and summing them to find the summation of divisors.

Finding Number of Divisors

Now, we want to find the number of divisors of an integer. It can be done easily by finding all the divisors and finally counting them as described above. But there is a simpler idea. If we know the prime factorization of a number, we can easily find the number of divisors.

For example, 18 has 6 divisors. If we prime factorize 18, we find that

$18 = 2^1 * 3^2$

We can say that any divisor of 36 should be in the form

$2^x * 3^y$ where $0 <= x <= 1$ and $0 <= y <= 2$

This can be proved easily, but if we think a bit we will find that the reason is obvious.

So, the divisors are,

$2^0 * 3^0 = 1$
$2^0 * 3^1 = 3$
$2^0 * 3^2 = 9$
$2^1 * 3^0 = 2$
$2^1 * 3^1 = 6$
$2^1 * 3^2 = 18$

That means we are taking all possible solutions. These form the number of divisors. Just see the pattern of the prime factors of the divisors. So, we can say that the number of divisors of 18 is

$(1 + 1) * (2 + 1) = 6$

So, if a number is factorized as

$2^x * 3^y$

Then the number of divisors is $(x+1) * (y+1)$. Again, the reason is

$2^0 * 3^0$
$2^0 * 3^1$
...
$2^0 * 3^{y-1}$
$2^0 * 3^y$

} $y + 1$ divisors

$2^1 * 3^0$
$2^1 * 3^1$
...
$2^1 * 3^{y-1}$
$2^1 * 3^y$

} $y + 1$ divisors

...

$2^x * 3^0$
$2^x * 3^1$
...
$2^x * 3^{y-1}$
$2^x * 3^y$

} y + 1 divisors

So, there are (x+1) sections each having (y+1) divisors, which means a total of (x+1) * (y+1) divisors.

Now, let's find the total number of divisors of 60.

$60 = 2^2 * 3 * 5$

So, similarly the divisors are

$2^0 * 3^0 * 5^0 = 1$
$2^0 * 3^0 * 5^1 = 5$
$2^0 * 3^1 * 5^0 = 3$
$2^0 * 3^1 * 5^1 = 15$
$2^1 * 3^0 * 5^0 = 2$
$2^1 * 3^0 * 5^1 = 10$
$2^1 * 3^1 * 5^0 = 6$
$2^1 * 3^1 * 5^1 = 30$
$2^2 * 3^0 * 5^0 = 4$
$2^2 * 3^0 * 5^1 = 20$
$2^2 * 3^1 * 5^0 = 12$
$2^2 * 3^1 * 5^1 = 60$

So, total number of divisors is 12. Let's check it in another way.

For each section of 2 (2^0, 2^1 or 2^2) the number of divisors are same. So, there are 3 sections for 2. Now for each section of 2, there are two sections for 3 (3^0, 3^1). Now for any combination of 2 and 3 there are two sections for 5 (5^0, 5^1). So, it can be represented as

(2^0, 2^1, 2^2) and (3^0, 3^1) and (5^0, 5^1)

So, at first take any power of 2, after that take any power of 3 and after that take any power of 5 from the above sections. So, we can take 2 in three ways, 3 in two ways and 5 in two ways. So, total ways (or total divisors) are

3 * 2 * 2
= 12

Similarly, suppose a number is factorized as $2^x * 3^y * 5^z$, so, the total number of divisors is

$(x + 1) * (y + 1) * (z + 1)$

Now, the general formula is

If the prime factorization of an integer is

$p_1^{x1} * p_2^{x2} * p_3^{x3} * ... * p_n^{xn}$

where, $p_1, p_2, ..., p_n$ are primes, then the number of divisors is

$(x_1 + 1) * (x_2 + 1) * (x_3 + 1) * ... * (x_n + 1)$

Finding Summation of Divisors

Suppose we want to find the summation of all the divisors of an integer. We can do it by finding all the divisors and then summing them. But we can make it better. The idea is as follows.

Let's say we want to find the summation of divisors of 60. From previous section, we know that the divisors are `1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30` and `60`; so the summation is `168`. The summation can be written as (summing up all the divisors in their factorized form),

$$2^0 * 3^0 * 5^0 + 2^0 * 3^0 * 5^1 + 2^0 * 3^1 * 5^0 + 2^0 * 3^1 * 5^1 +$$
$$2^1 * 3^0 * 5^0 + 2^1 * 3^0 * 5^1 + 2^1 * 3^1 * 5^0 + 2^1 * 3^1 * 5^1 +$$
$$2^2 * 3^0 * 5^0 + 2^2 * 3^0 * 5^1 + 2^2 * 3^1 * 5^0 + 2^2 * 3^1 * 5^1$$

It can be written as,

$$2^0 * (3^0 * 5^0 + 3^0 * 5^1 + 3^1 * 5^0 + 3^1 * 5^1) +$$
$$2^1 * (3^0 * 5^0 + 3^0 * 5^1 + 3^1 * 5^0 + 3^1 * 5^1) +$$
$$2^2 * (3^0 * 5^0 + 3^0 * 5^1 + 3^1 * 5^0 + 3^1 * 5^1)$$

can be written as

$$(2^0 + 2^1 + 2^2) * (3^0 * 5^0 + 3^0 * 5^1 + 3^1 * 5^0 + 3^1 * 5^1)$$

can be written as

$$(2^0 + 2^1 + 2^2) * (3^0 * (5^0 + 5^1) + 3^1 * (5^0 + 5^1))$$

can be written as

$$(2^0 + 2^1 + 2^2) * (3^0 + 3^1) * (5^0 + 5^1)$$

Now, we know that the summation of series

$$1 + r + r^2 + r^3 + r^4 + \ldots + r^n = \frac{r^{n+1} - 1}{r - 1}$$

So, the summation can be written as

$$\frac{2^3 - 1}{1} * \frac{3^2 - 1}{2} * \frac{5^2 - 1}{4} = 7 * 4 * 6 = 168$$

So, the result is same as we have calculated.

So, if the prime factorization of an integer is

$$p_1{}^{x_1} * p_2{}^{x_2} * \ldots * p_n{}^{x_n}$$

where, $p_1, p_2, \ldots, p_n$ are primes, then the summation of divisors is

$$\frac{p_1{}^{x_1+1} - 1}{p_1 - 1} * \frac{p_2{}^{x_2+1} - 1}{p_2 - 1} * \ldots * \frac{p_n{}^{x_n+1} - 1}{p_n - 1}$$

Discussion

After finding the prime factorization, these ideas are easy to code. Before get to coding, try to understand them fully.

Sieve of Eratosthenes

(Generating Primes)

by

Jane Alam Jan

Introduction

Before starting the problem, you have to understand how to generate primes. This document contains some ideas and observations that make a naïve algorithm to a better one.

The first task is to know what is a *prime*. Before advancing, we should understand the word '**divisible**'.

An integer '*a*' is said to be ***divisible*** by an integer '*b*' if we divide *a* by *b*, the reminder is zero. For example 20 is divisible by 5, 26 is divisible by 2, 30 is **not** divisible by 16. Or we can say that *a* is divisible by *b* if *a* can be written as *b* * *k* where *k* is an integer.

A *prime* number is an integer which is *divisible* by exactly two different integers. By definition, we see that 1 is not a prime. Since we need two different integers, but 1 is only divisible by 1. So, 5 is a *prime*, because it is divisible by two different integers - 1 and 5. 49 is **not** a *prime* because it is divisible by 1, 7 and 49 (three different integers).

The problem is - we have to generate *primes* from 1 to ***N***. But how to generate *primes*?

Idea 1 (divide by all possible numbers)

We have to check whether a number is a *prime* or not. A number *P* is said to be *prime* if it is not divisible by any integer from *2* to *P-1*.

```
int N = 5000;

bool isPrime( int i ) {
    for( int j = 2; j < i; j++ ) {
        if( i % j == 0 ) // i is divisible by j, so i is not a prime
            return false;
    }
    // No integer less than i, divides i, so, i is a prime
    return true;
}

int main() {
    for( int i = 2; i <= N; i++ ) {
        if( isPrime(i) == true )
            printf("%d ", i);
    }
    return 0;
}
```

This code will generate all the primes up to 5000. But if we see carefully, we will find that to check any number we have to divide it by all the numbers smaller to it. So, for 3, we will try to divide 3 by 2. For 7, we will try to divide it by 2, 3, 4, 5, 6. So, this method is slow, because so many divisions and checks are required.

Idea 2 (divide up to square root)

If we observe the property of the numbers, we will find some interesting facts. Suppose we have a number 36.

36 is divisible by 1, 2, 3, 4, 6, 9, 12, 18 and 36. It is clear that if *a* is divisible by *b*, then *a* is also divisible by *(a/b*). That means, 36 is divisible by 3, so, this property helps us to find that 36 will also be divisible by 12 *(36/3)*. Now we will make two lists, if *a* is divisible by *b*, then we will take *b* into the first list, *(a/b)* and into the second list.

So, for 36, we will evaluate our idea.

36 is divisible by 1. So, it will be added in list 1, and (36/1) = 36 will be added in list 2.

```
List 1: 1

List 2: 36
```

36 is divisible by 2. So, it will be added in list 1, and (36/2) = 18 will be added in list 2.

```
List 1: 1 2

List 2: 36 18
```

36 is divisible by 3. So, it will be added in list 1, and (36/3) = 12 will be added in list 2.

```
List 1: 1 2 3

List 2: 36 18 12
```

36 is divisible by 4. So, it will be added in list 1, and (36/4) = 9 will be added in list 2.

```
List 1: 1 2 3 4

List 2: 36 18 12 9
```

36 is divisible by 6. So, it will be added in list 1, and (36/6) = 6 will be added in list 2.

```
List 1: 1 2 3 4 6

List 2: 36 18 12 9 6
```

Now, see that if we move further we will find the numbers again (some numbers from list 2, which will try to insert into list 1) So, it is clear that, if we have list 1, we can generate list 2 (from this position).

So, we can stop checking if we find *b* for which *a/b* is less than or equal to *b*. Cause if we try to divide by numbers greater than *b* we will actually repeat the same steps. For 36, (check the lists) the next number that will be included to list 1 is 9, but 36/9 = 4, which is already in list 1.

Now we have to find *b* for which *a/b* is less than or equal to *b*. If we think mathematically,

$$b >= a/b$$

$$\text{or, } b^2 >= a$$

$$\text{or, } b >= \text{sqrt}(a)$$

$$\text{so, } b_{\textbf{minimum}} = \text{sqrt}(a)$$

So, we can update our code.

```
int N = 5000;

bool isPrime( int i ) {
    int sqrtI = int( sqrt( (double) i) );
    // dont write "for(int j = 2; j <= sqrt(i); j++)" because sqrt is a slow
    // function. So, dont calculate it all the time, calculate it only once
    for( int j = 2; j <= sqrtI; j++ ) {
        if( i % j == 0 ) // i is divisible by j, so i is not a prime
            return false;
    }
    // No integer less than i, divides i, so, i is a prime
    return true;
}

int main() {
    for( int i = 2; i <= N; i++ ) {
        if( isPrime(i) == true )
            printf("%d ", i);
    }
    return 0;
}
```

Idea 3 (no need to take even numbers)

Only 2 is the even prime, all other primes are odd, so, we can update our algorithm. Firstly we will not check even numbers (*condition 1*), and secondly we will not use even numbers to divide (*condition 2*). Since an odd number can be divisible by only odd numbers.

So, our new code will be like

```
int N = 5000;

bool isPrime( int i ) {
    int sqrtI = int( sqrt( (double) i) );
    for( int j = 3; j <= sqrtI; j += 2 ) { // j += 2 is given, condition (2)
        if( i % j == 0 ) // i is divisible by j, so i is not a prime
            return false;
    }
    return true;
}

int main() {
    printf("2 "); // 2 is the only even prime, so, print it
    for( int i = 3; i <= N; i += 2 ) { // i += 2 is given here, condition (1)
        if( isPrime(i) == true )
            printf("%d ", i);
    }
    return 0;
}
```

Idea 4 (Sieve of Eratosthenes)

We can easily say that, if a number is not divisible by any primes less than it, then it is *prime*. For example 7 is a prime because it is not divisible by 2, 3, or 5. Suppose a number is not *prime*, then there should be a prime which divides the number. We can say that 100 is not a prime and it is divisible by 20. We can see that 20 is divisible by 5, so 100 will also be divisible by 5. That means we can only check the previous primes to ensure whether a number is prime or not. And of course the square root bound also works here. Now this idea is quite hard to implement if we want to use the previous codes. How about we generate a new idea?

The idea is - we will discard all the multiples. That means if we find 2, we will discard 4, 6, 8, … Because we are sure that if 2 is a prime, then none of its multiple will be.

We should understand how it works, consider that we have the numbers

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

And initially consider that all are primes. We assume that a number – 'colored black' is prime.

At first we check 2 and color (discard) all multiples of 2.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Then we advance to 3, and find that it is also a prime. So, we will discard all the multiples of 3.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Now we advance to 4, but see that 4 is not a prime because it is already colored. So, we advance to 5 and discard its multiples.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

So, we continue until all cant color anymore.

So, to implement the idea, we will use an array (status), which will contain 0 or 1. If the *ith* index contains 0 then we will think that *i* is a prime, otherwise it is not.

```
int N = 5000;
int status[5001];
// status[i] = 0, if i is prime
// status[i] = 1, if i is not a prime
int main() {
    int i, j;
    // initially we think that all are primes, so change the status
    for( i = 2; i <= N; i++ )
    status[i] = 0;

    for( i = 2; i <= N; i++ ) {
        if( status[i] == 0 ) {
            // so, i is a prime, so, discard all the multiples
            // j = 2 * i is the first multiple, then j += i, will find the
            // next multiple
            for( j = 2 * i; j <= N; j += i )
                status[j] = 1; // status of the multiple is 1
        }
    }
    // print the primes
    for( i = 2; i <= N; i++ ) {
        if( status[i] == 0 ) {
            // so, i is prime
            printf("%d ", i);
        }
    }
    return 0;
}
```

Idea 5 (again, no even numbers)

Again, as described in idea - 3, 2 is the only even prime, so we can update our idea. So, we will start from 3. Since we are dealing with only odd numbers, so, we will not color the even numbers. And see that the first multiple of 3 is 6, then we have 9, 12, 15, 18, … So, for any odd number the first multiple is even, the second one is odd, the third one is even, and so on. So, we can easily avoid even numbers.

```
int N = 5000;
int status[5001];

// status[i] = 0, if i is prime
// status[i] = 1, if i is not a prime

int main() {
    int i, j;
    // initially we think that all are primes
    for( i = 2; i <= N; i++ )
    status[i] = 0;

    for( i = 3; i <= N; i += 2 ) {
        if( status[i] == 0 ) {
            // so, i is a prime, so, discard all the multiples
            // 3 * i is odd, since i is odd. And j += 2 * i, so, the next odd
            // number which is multiple of i will be found
            for( j = 3 * i; j <= N; j += 2 * i )
                status[j] = 1; // status of the multiple is 1
        }
    }
    // print the primes
    printf("2 ");
    for( i = 3; i <= N; i += 2 ) {
        if( status[i] == 0 ) {
            // so, i is prime
            printf("%d ", i);
        }
    }
    return 0;
}
```

Idea 6 (modified discarding technique)

Now let's revise the discarding technique for the last code. Initially we have the list

```
3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
```

Now at first we discard (or color) the multiples of 3. So, we have

```
3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
```

Now, the next number is 5 and the first number that will be colored is 15. But is it necessary to color 15? Observe that, 15 = 3 * 5, so, we are sure that 15 is colored already. So, we start from 25.

```
3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
```

Now the next number is 7. The first number that will be discarded is 21. But again 21 = 3 * 7, so, we are sure that 21 is already colored. The next number is 35 (advancing 21 + 2 * 7 = 35), again 35 = 5 * 7, so, it's also colored. So, then we have 49, and it is the first number that should be colored. So, we have

```
3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
```

So, if we have ***n*** as a prime, and we want to discard its multiples, then 3 * n is not a good choice, not even 5 * n, not even (n - 2) * n. The reason is all the numbers mentioned here are colored already. So, the first number that should be colored is $\mathbf{n^2}$. So, we have to check primes up to square root of **N** and discard their multiples (say, we want to find primes up to 51, is it necessary to check the multiples of 11, or higher?). So, the new code will be

```
int N = 5000, status[5001];
int main() {
    int i, j, sqrtN;
    for( i = 2; i <= N; i++ ) status[i] = 0;
    sqrtN = int( sqrt((double) N )); // have to check primes up to (sqrt(N))
    for( i = 3; i <= sqrtN; i += 2 ) {
        if( status[i] == 0 ) {
            // so, i is a prime, so, discard all the multiples
            // j = i * i, because it's the first number to be colored
            for( j = i * i; j <= N; j += i + i )
                status[j] = 1; // status of the multiple is 1
        }
    }
    // print the primes
    printf("2 ");
    for( i = 3; i <= N; i += 2 ) {
        if( status[i] == 0 ) printf("%d ", i);
    }
    return 0;
}
```

Idea 7 (saving memory)

We are not considering even numbers, but we have declared memory for even numbers. For example we do have `status[2], status[4], status[506]` etc, which are completely wasted. So, we can easily observe that half of the allocated memories are actually of no use. Just think that between 1 and 100 there are 50 odd numbers and 50 even numbers.

Now we know that any odd number can be represent as *2 * i + 1* form where *i* is an integer. Like 3 can be represented as *2 * 1 + 1*, 11 can be represented as *2 * 5 + 1*.

So, we can update our idea with the property that if status of *i* equals 0 then we will think that *2 * i + 1* is a prime, and then we will discard all multiples of *2 * i + 1*. When discarding the multiples, we will find the multiple in *2 * x + 1* form, and we will update the status of x. So, If we check the status of x, we are actually checking that *2 * x + 1* is a prime or not. So, for any integer *p*, we will check the status of *p/2*. Division is a bit costlier, so we can right shift *p* once instead of dividing it by 2 as we will have same results. So, the new code will be

```
int N = 5000, status[2501];
int main() {
    int i, j, sqrtN;
    for( i = 2; i <= N >> 1; i++ ) status[i] = 0;
    sqrtN = int( sqrt((double)N) ); // have to check primes up to (sqrt(N))
    for( i = 3; i <= sqrtN; i += 2 ) {
        if( status[i>>1] == 0 ) {
            // so, i is a prime, so, discard all the multiples
            // j = i * i, because it's the first number to be colored
            for( j = i * i; j <= N; j += i + i )
                status[j>>1] = 1; // status of the multiple is 1
        }
    }
    // print the primes
    printf("2 ");
    for( i = 3; i <= N; i += 2 ) {
        if( status[i>>1] == 0 )
            printf("%d ", i);
    }
    return 0;
}
```

Discussion

The basic idea of sieve is described here. The code here may have bugs. Before you believe the codes, try to understand the ideas and convince yourself. Try solving some problems related to prime numbers.

Prime Factorization

by

Jane Alam Jan

Prime Factorization

Introduction

Sometimes we need to prime factorize numbers. So, what is prime factorization? Actually prime factorization means finding the prime factors of a number. That means the primes that divide the number, should be listed and how many times a prime divides the number should also be listed. For example,

$40 = 2^3 * 5$

$120 = 2^3 * 3 * 5$

$147 = 3 * 7^2$

$121 = 11^2$

$4021920 = 2^5 * 3^3 * 5 * 7^2 * 19$

Now, how to generate the prime factorization of a number? So, obviously the first task is - we have to generate primes. You can read the paper for "**Sieve of Eratosthenes**" to know how to find primes. From now on, we will think that we have an array named **`prime[]`** which contains all the primes. So, `prime[0] = 2, prime [1] = 3, prime [2] = 5,...`

Idea 1

The first idea that comes into mind is, we can check all the primes and try to divide the number. If we can divide the number by a prime, we will try to divide as many times as we can. We will continue our work until the number becomes 1. For example, we want to prime factorize 40.

At first, we take an empty list; it will contain the primes we have used for factorization (that's how we can find the primes we have used). The steps are given below. Initially we have

```
40
List: (empty)
```

Step 1)

40 is divisible by 2 (the 1st prime), so, we divide 40 by 2 (and get 20) and add 2 in the list, we get

```
20
List: 2
```

Step 2)

20 is divisible by 2, so, we divide 20 by 2 (and get 10) and add 2 in the list, we get

```
10
List: 2 2
```

Step 3)

10 is divisible by 2, so, we divide 10 by 2 (and get 5) and add 2 in the list, we get

```
5
List: 2 2 2
```

Step 4)

5 is not divisible by 3 (the 2nd prime), so the list remains unchanged

Step 5)

5 is divisible by 5 (the 3rd prime), so, we divide 5 by 5 (and get 1) and add 5 in the list, we get

```
1
List: 2 2 2 5
```

Now if we go further it will be unnecessary, because 1 can't be factorized further. So, we are sure that $40 = 2 * 2 * 2 * 5 = 2^3 * 5$. This way we can prime factorize any number.

The following code implements the same idea as described.

```
int List[128];  // saves the List
int listSize;   // saves the size of List

void primeFactorize( int n ) {
    listSize = 0;   // Initially the List is empty, we denote that size = 0
    for( int i = 0; prime[i] <= n; i++ ) {
        if( n % prime[i] == 0 ) { // So, n is a multiple of the ith prime
            // So, we continue dividing n by prime[i] as long as possible
            while( n % prime[i] == 0 ) {
                n /= prime[i]; // we have divided n by prime[i]
                List[listSize] = prime[i]; // added the ith prime in the list
                listSize++; // added a prime, so, size should be increased
            }
        }
    }
}
int main() {
    primeFactorize( 40 );
    for( int i = 0; i < listSize; i++ ) // traverse the List array
        printf("%d ", List[i]);
    return 0;
}
```

If we analyze the code a bit, we will find that, the condition should be **`n != 1`**, but here, the condition is given as `prime[i] <= n`. Do they have any difference?

Idea 2

The idea described in the previous section is not quite good, since if a prime number is given, say the 10000th prime, we have to check all the primes up to it and then we can find the factorization. For example, if we want to find the prime factorization of 19 (the 9th prime), then the code will try to divide 19 by 2, then 3, 5, 7, 11, 13, 15, 17, and finally 19. So, only 19 divides 19, then we will find the prime factorization of 19. So, it takes 9 operations.

The better idea is check primes up to sqrt(`n`). This idea is described in the paper for "**Sieve of Eratosthenes**". As we have stated there - an integer `n`, which is not a prime, will have a divisor (a prime divisor, too) less than or equal to sqrt(`n`). So, this way can divide **`n`** by its prime divisor, thus reducing it. But if we follow this way, can there be any problems?

At first, we try to factorize 40. The integer part (or floor) of sqrt(40) = 6. So, we will take primes up to 6. Now see that the largest prime used for 40 is 5 (in **Idea 1**). So, we will have the same steps as in **Idea 1**.

Now we try to factorize 114. The integer part (or floor) of sqrt(114) = 10. So, we will take primes up to 10. Initially we have

```
114
List: (empty)
```

Step 1)

114 is divisible by 2, so, we divide 114 by 2 (and get 57) and add 2 in the list, we get

```
57
List: 2
```

Step 2)

57 is divisible by 3, so, we divide 57 by 3 (and get 19) and add 3 in the list, we get

```
19
List: 2 3
```

Step 3)

19 is not divisible by 5 or 7. We will stop here, because the next prime is 11, but we assumed that we will check primes up to 10. So, we are sure that 19 doesn't have any prime divisor less than or equal to 10. So, as we have mentioned already, 19 must be a prime. Thus we add 19 in the list.

```
1
List: 2 3 19
```

So, to find the prime factorization of a given number, this method will check primes up to sqrt(**n**), which is far better than the previous one. So, the necessary prime numbers to factorize **n** is the primes numbers less than or equal to sqrt(**n**). So, when we generate primes, we can only generate the important primes checking the limit of the test data. For example if the maximum number in the input is 1000000, we need to generate primes up to sqrt(1000000) = 1000. So, primes up to 1000 are required only!

The following code implements the idea, which we have just described.

```
int List[128];  // saves the List
int listSize;   // saves the size of List

void primeFactorize( int n ) {
    listSize = 0;   // Initially the List is empty, we denote that size = 0
    int sqrtN = int( sqrt(n) ); // find the sqrt of the number
    for( int i = 0; prime[i] <= sqrtN; i++ ) { // we check up to the sqrt
        if( n % prime[i] == 0 ) { // n is multiple of prime[i]
            // So, we continue dividing n by prime[i] as long as possible
            while( n % prime[i] == 0 ) {
                n /= prime[i]; // we have divided n by prime[i]
                List[listSize] = prime[i]; // added the ith prime in the list
                listSize++; // added a prime, so, size should be increased
            }
            // we can add some optimization by updating sqrtN here, since n
            // is decreased. think why it's important and how it can be added
        }
    }
    if( n > 1 ) {
        // n is greater than 1, so we are sure that this n is a prime
        List[listSize] = n; // added n (the prime) in the list
        listSize++; // increased the size of the list
    }
}
int main() {
    primeFactorize( 114 );
    for( int i = 0; i < listSize; i++ ) // traverse the List array
        printf("%d ", List[i]);
    return 0;
}
```

Discussion

How many primes do we generate? We need the primes to the sqrt(max number in the input). But there can be a problem. In the above algorithm we have used the condition **`prime[i] <= sqrtN`**. So, when **`prime[i]`** is **greater than** **`sqrtN`** then the loop will terminate, but if we generate primes exactly up to **`sqrtN`**, a problem may occur. For example, say we are about to prime factorize 29, and so, we have generated primes 2, 3, 5 only (up to integer part of sqrt(29) = 5). So, `prime[0] = 2, prime[1] = 3` and `prime[2] = 5`. Now in the loop when `i` becomes 2, so `prime[i] = 5` which is less than or equal to square root of 29 and thus the loop will move further. But in the next turn, `i` increases and becomes 3. But as we have generated only 3 primes (`prime[0 to 2]`), `prime[3]` (invalid value) will be compared to sqrt(29). So, when generating primes, be sure that at least one extra prime is generated for safety.

Big Integer Library

by

Jane Alam Jan

Introduction

Often we need to write codes which can handle big integer operations. Suppose we have to add two numbers which can have 1000 digits. Then none of the usual data types would help. Though it's not that tough to code, but it's boring and tiresome.

Here we are giving a Big Integer library which has most of the operations including some mathematical and conditional operations. Mainly we have focused on some important facts.

1) The size of the code is an issue since if the size is too big then it would take so much time just to write it in real contests.
2) Negative numbers should be supported.
3) Number of digits shouldn't be an issue. If we need more digits we should allocate them dynamically.
4) Some exceptions like division by zero should be reported properly such that related coding bugs can be found.
5) The code should produce the same result as the basic C++ mathematical operators. For example, we should output -1 for (-10 % 3) which is also the result found using C++.

So, we will give the code at first. After that we will describe it fully and its merits and drawbacks.

Code

```
// header files

#include <cstdio>
#include <string>
#include <algorithm>
#include <iostream>

using namespace std;

struct Bigint {
    // representations and structures
    string a; // to store the digits
    int sign; // sign = -1 for negative numbers, sign = 1 otherwise

    // constructors
    Bigint() {} // default constructor
    Bigint( string b ) { (*this) = b; } // constructor for string

    // some helpful methods
    int size() { // returns number of digits
        return a.size();
    }
    Bigint inverseSign() { // changes the sign
        sign *= -1;
        return (*this);
    }
    Bigint normalize( int newSign ) { // removes leading 0, fixes sign
        for( int i = a.size() - 1; i > 0 && a[i] == '0'; i-- )
            a.erase(a.begin() + i);
        sign = ( a.size() == 1 && a[0] == '0' ) ? 1 : newSign;
        return (*this);
    }

    // assignment operator
    void operator = ( string b ) { // assigns a string to Bigint
        a = b[0] == '-' ? b.substr(1) : b;
        reverse( a.begin(), a.end() );
        this->normalize( b[0] == '-' ? -1 : 1 );
    }

    // conditional operators
    bool operator < ( const Bigint &b ) const { // less than operator
        if( sign != b.sign ) return sign < b.sign;
        if( a.size() != b.a.size() )
            return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.size();
        for( int i = a.size() - 1; i >= 0; i-- ) if( a[i] != b.a[i] )
            return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
        return false;
    }
    bool operator == ( const Bigint &b ) const { // operator for equality
        return a == b.a && sign == b.sign;
    }
```

```
    // mathematical operators
    Bigint operator + ( Bigint b ) { // addition operator overloading
        if( sign != b.sign ) return (*this) - b.inverseSign();
        Bigint c;
        for(int i = 0, carry = 0; i<a.size() || i<b.size() || carry; i++ ) {
            carry+=(i<a.size() ? a[i]-48 : 0)+(i<b.a.size() ? b.a[i]-48 : 0);
            c.a += (carry % 10 + 48);
            carry /= 10;
        }
        return c.normalize(sign);
    }
    Bigint operator - ( Bigint b ) { // subtraction operator overloading
        if( sign != b.sign ) return (*this) + b.inverseSign();
        int s = sign; sign = b.sign = 1;
        if( (*this) < b ) return ((b - (*this)).inverseSign()).normalize(-s);
        Bigint c;
        for( int i = 0, borrow = 0; i < a.size(); i++ ) {
            borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
            c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
            borrow = borrow >= 0 ? 0 : 1;
        }
        return c.normalize(s);
    }
    Bigint operator * ( Bigint b ) { // multiplication operator overloading
        Bigint c("0");
        for( int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] - 48 ) {
            while(k--) c = c + b; // ith digit is k, so, we add k times
            b.a.insert(b.a.begin(), '0'); // multiplied by 10
        }
        return c.normalize(sign * b.sign);
    }
    Bigint operator / ( Bigint b ) { // division operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0"), d;
        for( int j = 0; j < a.size(); j++ ) d.a += "0";
        int dSign = sign * b.sign; b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b, d.a[i]++;
        }
        return d.normalize(dSign);
    }
    Bigint operator % ( Bigint b ) { // modulo operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0");
        b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b;
        }
        return c.normalize(sign);
    }
```

```
    // output method
    void print() {
        if( sign == -1 ) putchar('-');
        for( int i = a.size() - 1; i >= 0; i-- ) putchar(a[i]);
    }
};

int main() {
    Bigint a, b, c; // declared some Bigint variables

    ////////////////////////
    // taking Bigint input //
    ////////////////////////
    string input; // string to take input

    cin >> input; // take the Big integer as string
    a = input; // assign the string to Bigint a

    cin >> input; // take the Big integer as string
    b = input; // assign the string to Bigint b

    ///////////////////////////////////
    // Using mathematical operators //
    ///////////////////////////////////

    c = a + b; // adding a and b
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a - b; // subtracting b from a
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a * b; // multiplying a and b
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a / b; // dividing a by b
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a % b; // a modulo b
    c.print(); // printing the Bigint
    puts(""); // newline

    ////////////////////////////////
    // Using conditional operators //
    ////////////////////////////////

    if( a == b ) puts("equal"); // checking equality
    else puts("not equal");

    if( a < b ) puts("a is smaller than b"); // checking less than operator

    return 0;
}
```

Documentation

Here we will describe the full idea behind the code. Mostly some parts of the codes are compact just to make it relatively small. Before going the details of the operators, we first describe the representations and structures.

Representations and Structures

We used a structure **`Bigint`** for the big integer operations. In **`Bigint`** we used two variables.

`string a:` stores the big integer digits in decimal number system. It contains digits only for both positive and negative numbers. We used **`string`** because it can hold arbitrary size. So, the number of digits will not be an issue.

`int sign:` represents whether the big integer is negative or not. If the value of **`sign`** is -1 then it contains a negative integer. Otherwise if the value is 1 then the integer can be positive or zero.

When we add, multiply or subtract integers we start from the right most digit or we can say we start from the least significant digit. After that we iterate left for rest of the calculations. That's why we store the digits in reversed form in '**`a`**'. So, to add, subtract or multiply we can start from left.

Instead of using methods, we overloaded some of the mathematical and conditional operators to simplify the usability of the code.

Constructors

There are two constructors.

`Bigint():` is the default constructor.

`Bigint( string b ):` uses assignment operator **`=`** such that the **`Bigint`** contains the string.

Some Helpful Methods

There are three methods in this section.

`size():` returns the number of digits of the Big integer. Since we store the digits in '**`a`**', so the length of '**`a`**' is actually the number of digits of the **`Bigint`**.

`inverseSign():` actually changes the sign from 1 to -1, or from -1 to 1. It has many uses. For example it's helpful for addition between a positive and a negative number. In this case we can use subtraction changing one of the numbers **`sign`**.

`normalize( int newSign ):` normalizes a **`Bigint`**. It actually removes leading zeroes from the big integer as well as it fixes the **`sign`** of the big integer according to **`newSign`**. It's normally used for subtractions or multiplications. Say we subtract 99 from 100. Usual calculations will find 01 as the result. After normalization the result will be 1.

Assignment Operator

`void operator = ( string b ):` it helps to store string b in the **`Bigint`**. What it does it quite simple. If **`b`** is negative then we store **`b[1..end]`** to **`a`**. Otherwise we store whole **`b[0..end]`** to **`a`**. After that we find the correct **`sign`** and we return the normalized **`Bigint`**.

Conditional Operators

We actually implemented two operators. We can easily add conditional operators, but to make the code compact we ignored them.

`bool operator < ( const Bigint &b ) const:` returns true if the current **`Bigint`** is less than **`Bigint b`**. We have used **`const`** since we actually used the reference of **`b`**, so, if we accidentally change **`b`** it would be a problem, but **`const`** will prevent it. This function is defined such that STL **`sort()`** can be used easily.

The idea of this function is simple. First we check the both the **`signs`**. If they are different then we can find the result based on sign. Otherwise we check numbers of digits of them. If they are different, then if they are negative numbers then the **`Bigint`** with greater size will be smaller. Otherwise if they are positive numbers then the smaller sized **`Bigint`** is smaller.

Now if their numbers of digits are also same, then we iterate from right and based on the digit and sign we find the smaller number. If they are positive numbers then the number containing smaller digit is smaller. For negative numbers the case is opposite. If the right most digits are equal then we still iterate to left until we find unequal digits.

`bool operator == ( const Bigint &b ) const:` it checks the equality of two **`Bigints`**. It's quite simple. If both the signs and the containing digits are equal then they are considered as equal.

Only these two conditional operators are listed because we can derive the other operators using these two operators. For example **`(a > b)`** can be written as **`(b < a)`**, **`(a <= b)`** can be written as **`(a < b || a == b)`**, **`(a != b)`** can be written as **`(!(a == b))`**.

If we think that you need more conditional operators to make things easier, you can easily overload some operators as well.

Mathematical Operators

Five of the main mathematical operators are overloaded here such as **+, -, *, /** and **%**.

Bigint operator + (Bigint b): returns the **Bigint** after adding **Bigint b** with the **Bigint** which initiates the operator. Here we have used the basic addition technique which we are using from childhood. Since the digits are stored in reversed fashion we start from left with no carry. After that we add digits and form carry and continue the procedure until we have no digits left. Throughout the process we saved the resulting digits to a new **Bigint** and finally we return that **Bigint**.

Since here we are thinking of both positive and negative numbers, so, before adding we see their signs. If they are different then we change one of the sign and use subtraction. Otherwise we used the addition technique we have just described. We return the normalized **Bigint** for safety and perfection. The complexity for addition is **O(n)**.

Bigint operator - (Bigint b): returns the Bigint after subtracting **b** from the operator initiator **Bigint**. The subtraction idea is also our childhood idea. We start from the leftmost digit (since we saved them in reversed order). After that we calculate borrow and iterate right. Finally we return the new normalized **Bigint**.

Again if both the signs are different then we don't have to use subtraction. We change the sign of **b** and return the addition result with **b**. The complexity for subtraction is **O(n)**.

Bigint operator * (Bigint b): returns the multiplication of the **Bigints**. The idea we have used for multiplication is a bit different than we have used it our childhood. The main theme is same but the implementation is different. The idea is that suppose we want to multiply **a** and **b**. Now we take **c = 0**. After that we start from the rightmost digit of **a**. If this digit is **k** then we add **b** to **c**, **k** times. After that we multiply **b** with **10** and we take the second rightmost digit of **a**. If this digit is **p** then we add **b** to **c**, **p** times. We continue this procedure until no digit is left in **a**. Since in **Bigint** we stored digits in reversed fashion so we start from the leftmost digit and we continue this procedure to find multiplication result. After that we normalized the result with the correct sign.

For example suppose we want to multiply 123 and 459. Now at first we say the result is 0. Now we start from taking the rightmost digit of 123. Since the rightmost digit is 3, we add 459 to result 3 times obtaining 1377 as result. Now we multiply 459 with 10 to get 4590. Now the next rightmost digit of 123 is 2. So, we add 4590 two times with 1377 to get 10557. Now again we multiply 4590 with 10 to get 45900. The last digit of 123 is 1, so, we add 45900 with 10557 to get 56457 which is the correct result. The complexity for multiplication is $O(n^2)$.

`Bigint operator / ( Bigint b ):` performs division. Division by zero can occur, that's why if this case occurs we force a division by zero operation such that the code creates the same exception.

However the idea we have used for division is a little bit different from the idea we used in school arithmetic. Say we want to divide **`a`** by **`b`**. Let **`c`** be the reminder throughout the process and **`d`** is the division result. Initially **`c`** and **`d`** both are **`0`**. Each time we take a digit from left of **`a`**. We multiply **`c`** with **`10`** and then add the digit to form the new reminder. While **`c`** is greater than or equal to **`b`** we subtract **`b`** from **`c`** and say we can subtract **`k`** times. Then we multiply **`d`** with **`10`** and add **`k`**. Then after the full iteration, **`c`** contains the reminder and **`d`** contains the division result.

For example suppose we want to divide 4567 with 12. Initially **`c`** and **`d`** both are zero. Now multiply **`c`** with 10 and add 4 (from **`a`**). Now **`c`** is 4 which is less than 12. So, **`k`** is zero and thus **`d`** will be zero. After that **`c`** will be 45. Now **`b`** can be subtracted from **`c`** 3 times. So, **`k`** = 3. Now **`d`** will be 3 and **`c`** will be (45 – 12 – 12 – 12) which is 9. Now in next iteration **`c`** will be 96 (multiplied with 10 and added the 3rd digit of **`a`** from left). So, we can subtract **`b`** from **`c`** 8 times. So, **`k`** will be 8. So, **`d`** will be 38 (multiplied with 10 and added 8) and **`c`** will be 0. Finally after adding the last digit of **`a`**, **`c`** will be 7 which is less than 12. So, **`k`** will be zero and **`d`** will be 380 (multiplied with 10 and added 0). So, the division result is 380 and reminder is 7. The Complexity for division is **`O(n`**^{2}**`)`**.

`Bigint operator % ( Bigint b ):` returns the modulo result by **`Bigint b`**. The procedure is same as the division method. Actually when finding the division result we find the reminder, too. The complexity is same as division, thus **`O(n`**^{2}**`)`**.

Output Method

`void print():` prints the **`Bigint`**. Actually it prints the sign if needed, and the numbers from **`string a`** in reversed order. Since we have stored the numbers in reversed order, so, if we print them in reversed order we get the right result.

main() function

In main() function we have shown some examples of how to use the **`Bigint`**. Since the operators are overloaded, expressions like **`(a * b + (d - e))`** can be used where the precedence of operators will remain correct.

Drawbacks

1) Don't expect the code to be too efficient. There are faster ways to do multiplications. You may search for **Karatsuba Algorithm** for faster multiplications.
2) The base we have used is 10. So, we can use only 0 to 9. If we change the base of the numbers to store then the length of the numbers will be small. For example if the base is 10000 then 12345678 will need only two digits, where in decimal base it needs 8 digits. If the length is small then multiplications and divisions can be done faster.
3) Some parts are too compact. So, be careful when writing it.
4) We have not overloaded some operators like we haven't even added the assignment operator for integers. If you need it you may convert the integers to strings and then you can use the assignment operator.