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.