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.