Sieve of Eratosthenes and Efficient exponentiation

By Muhammad Khan

Sieve of Eratosthenes

The sieve of Eratosthenes is an algorithm for finding all prime numbers in the range [1; n] in $O(n \log \log n)$ time and requiring only n bits of memory.

The algorithm works as follows:

1. Create a Boolean array of size n and mark each number in the array from 2 to n as prime (by setting the value at each index of the array to true).

2. Set p equal to 2 (the smallest prime number).

3. Mark each multiple of p starting from p^2 as composite (by setting the value at that index of the array to false).

4. Iterate through the rest of the array. For each prime number set p equal to this number. If p is greater than $\sqrt{n}$ then the algorithm is done. Otherwise repeat step 3. The next prime number at each point of the algorithm will be the next number in the array that is marked as prime.

Example

Find all prime numbers from 1 to 40

Create a Boolean array of size n and mark each number in the array from 2 to n as prime

	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

Example

Find all prime numbers from 1 to 40

Set all proper multiples of 2 (multiples of 2 that are greater than 2) as composite

	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

Example

Find all prime numbers from 1 to 40

The next prime number is 3 so mark all proper multiples of 3 as composite

	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

Example

Find all prime numbers from 1 to 40

The next prime number is 5 so mark all proper multiples of 5 as composite

	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

Example

Find all prime numbers from 1 to 40

The next prime number is 7. However 7 is greater than $\sqrt{40}$ so the algorithm is done. The remaining numbers that are unmarked are prime 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	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

Why the algorithm works

- A number is prime if it is not divisible by any prime number smaller than it. Because we iterate through the numbers in order, if we reach a number that has not yet been marked as composite, then this number is a prime number.
- For each prime number p, we mark each multiple of p as composite, starting from p^2. This is because each multiple of p less than p^2 has a prime factor less than p.
- If the current value of p is greater than $\sqrt{n}$, then the algorithm is done. This is because each composite number in the range [1; n] has a prime factor less than or equal to $\sqrt{n}$.

```
bool prime[n];
for i=2 to n:
    prime[i]=true;
for i=2 to n:
    if prime[i] == true:
        j = i*i;
        while j <= n:
            prime[j]=false;
            j = j+i;
```

Example implementation

```
int n;
bool prime[n+1];
for(int i=2; i<=n; i++)
    prime[i]=true;
for(int i=2; i*i<=n; i++){
    if(prime[i])
        for(int j=i*i; j<=n; j+=i)
            prime[j]=false;
}
```

Linear sieve

The linear sieve algorithm is an alternative algorithm to the sieve of Eratosthenes. It has a time complexity of $O(n)$. However, the downside is that it requires n bytes of memory whereas the sieve of Eratosthenes only requires n bits.

An advantage of using the linear sieve is that we can easily calculate the prime factorisation of any number in the range [2; n] after all the preprocessing is done.

How the algorithm works

We create an array lp which will contain the minimum prime factor of each number i in the range [2; n]. Initially, this array will contain zeroes at every index, indicating that these numbers are all prime.

Each time we encounter a prime number, we store this number in an array pr.

We then traverse through the array. At each index i we have two possibilities:

If $lp[i] = 0$ then i is a prime number.

If $lp[i] \neq 0$ then i is a composite number and its minimum prime factor is $lp[i]$.

We then update numbers in the array that are divisible by i in a way that each number is only updated once.

Example implementation

```
std::vector<int> lp(n+1), pr;
for(int i=2; i <= n; i++){
    if(lp[i]==0){
        lp[i]=i;
        pr.push_back(i);
    }
    for(int j=0; i * pr[j]<=n; j++){
        lp[i*pr[j]] = pr[j];
        if(pr[j]==lp[i])
            break;
    }
}
```

Why the algorithm works

Each number i has a unique representation in the form $i = lp[i] \cdot x$ where $lp[i]$ is the minimum prime factor of i and the number x doesn't have any prime factors less than $lp[i]$.

Our algorithm therefore goes through each prime number multiple p of i, and sets the lowest prime factor of this number to be p while $p \leq lp[x]$.

Binary exponentiation

Binary exponentiation is a method of calculating a^n using $O(\log n)$ multiplications.

By expressing n in its base 2 representation, we can calculate a^n by doing at most $\log_2 n$ multiplications. For example, $5^{11} = 5^{1011_2} = 5^8 \cdot 5^2 \cdot 5^1$

We can therefore calculate a^n efficiently if we know $a^1, a^2, a^4, \ldots, a^{\lfloor \log_2 n \rfloor}$. These numbers can be found by starting with a and repeatedly squaring it. We can then use bitwise operations to determine if the current power of a should be included in the representation of a^n.

The algorithm can be extended to calculate $a^n \; mod \; m$.

Example C++ implementation

```
long long binary(long long a, long long n){
    long long answer=1;
    while(n>0){
        if(n&1)
            answer*=a;
        a*=a;
        n>>=1;
    }
    return answer;
}
```

Example C++ implementation using mod

```
long long binary(long long a, long long n, long long m){
    a%=m;
    long long answer=1;
    while(n>0){
        if(n&1)
            answer=answer*a%m;
        a*=a;
        n>>=1;
    }
    return answer;
}
```

Practice problem

Given two integers a and b, find the last digit of a^b.

Input

Two integers, a $(0 \leq a \leq 20)$ and $b (0 \leq b \leq 10^{10})$

Output

One integer, d, the last digit of a^b.

Answer

Calculate $a^b \; mod \; 10$ using binary exponentiation

Resources

https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html

https://cp-algorithms.com/algebra/binary-exp.html

https://www.spoj.com/problems/LASTDIG/