Math

TAI, WEI HSUAN

August 15, 2025

Outline

- Prime Numbers
- Euclidean algorithm
- Modular
- Combinations

Prime Numbers

Given a positive integer N, can you check if it is prime?
Following is a simple algorithm:

```
1        bool is_prime(int n){
2            if(n==1)return false;
3            if(n==2)return true;
4            for(int i=2;i<n;++i){
5                if(n%i==0)return false;
6            }
7            return true;
8        }
```

Time complexity ?

If N isn't a prime number, we can show N as $a \times b (a \leq b)$, where a and b are positive integers. Moreover, a and b are not equal to 1 and N.
If we can find a or b, we can conclude that N is not a prime number. Observe that if a and b are both greater than $\sqrt{N}$, then $a \times b > N$. The same, if a and b are both less than $\sqrt{N}$, then $a \times b < N$.
Thus, we get a conclusion: $a \leq \sqrt{N} \leq b$. Therefore, we only need to check if N is divisible by any integer from 2 to $\sqrt{N}$.

Code

```
1         bool is_prime(int n){
2             if(n==1)return false;
3             if(n==2)return true;
4             for(int i=2;i*i<=n;++i){
5                 if(n%i==0)return false;
6             }
7             return true;
8         }
```

Sieve of Eratosthenes

The slide above shows how to check if a number is prime. But what if we want to find all prime numbers in a range $[1, N]$? The Sieve of Eratosthenes is a classic algorithm to find all prime numbers up to a given limit N efficiently. The main idea is: If p is a prime number, then all multiples of p (i.e., $2p, 3p, \ldots$) are not prime. We can use this property to mark non-prime numbers in a list.

Code for Sieve of Eratosthenes

```
1        void eratosthenes() {
2            for(int i=2;i<=N;i++)isprime[i]=true;
3            isprime[0]=isprime[1]=false;
4
5            for(int i=2;i<=N;i++)
6                if(isprime[i])
7                    for(int j=i*i;j<=N;j+=i)
8                        isprime[j]=false;
9        }
```

Time Complexity of Sieve of Eratosthenes

The Sieve of Eratosthenes works by iterating through each number up to N and marking its multiples as non-prime.
We can obtain this formula:

$$T(N) = \sum_{p \le N} \frac{N}{p} = N \times \sum_{p \le N} \frac{1}{p}$$

Where T is the total times of marking non-prime numbers, p is the prime number, and N is the limit.

Time Complexity of Sieve of Eratosthenes

With PNT(Prime Number Theorem), the number of primes up to N is approximately $\frac{N}{\ln N}$. To conduct the Sieve of Eratosthenes, we iterate through each number up to N and mark its multiples.
We can't directly get all the primes, but we can use **density** to estimate the number of primes. The density of primes around N is approximately $\frac{1}{\ln N}$, which means that the average gap between consecutive primes is about $\ln N$.

$$T(N) = \sum_{p \le N} \frac{N}{p} = N \times \sum_{p \le N} \frac{1}{p} = N \times \sum_{m=2}^{N} \frac{1_{m\_is\_prime}}{m} \approx N \sum_{m=2}^{N} \frac{1}{m} \cdot \frac{1}{\ln m}$$

Where T is the total times of marking non-prime numbers, p is the prime number, and N is the limit.

Time Complexity of Sieve of Eratosthenes

We can use integer to estimate the number of primes up to N:

$$T(N) = N \sum_{m=2}^{N} \frac{1}{m} \cdot \frac{1}{\ln m} \approx N \cdot \int_2^N \frac{1}{x} \cdot \frac{1}{\ln x}\, dx = N \cdot (\ln(\ln(N)) - \ln(\ln(2)))$$

Therefore, the time complexity of the Sieve of Eratosthenes is approximately $O(N \cdot \ln(\ln(N)))$.

Euclidean algorithm

Given two integers a, b, can you find their greatest common divisor (GCD) efficiently?
Following is a simple algorithm:

```
1        bool GCD(int a, int b){
2            if(a>b)swap(a, b);
3            for(int i=a;i>=1;--i)
4                if(a%i==0 and b%i==0)return i;
5        }
```

Time complexity ?

Euclidean algorithm

Given two integers a, b, can you find their greatest common divisor (GCD) efficiently?
Following is a simple algorithm:

```
1        int gcd(int a, int b){
2            return a==0 ? b : gcd(b, a%b);
3        }
```

Time complexity ?

How does it work?

The greatest common factor of two integers is equal to the greatest common factor of the smaller number and the remainder when the two numbers are divided.
For examples, we have $gcd(252, 105)$. $252 \div 105 = 2 \ldots 42$, so gcd(252, 105) must be equal to gcd(105, 42).
Iterating this process, we can find the GCD of two integers.

C++ built-in GCD

C++ provides a built-in function to calculate the GCD of two integers.

```
1         int a, b;
2         cout<<__gcd(a, b)<<'\n';
```

Modular

Basic modular arithmetics:

- $a \equiv b \pmod{m}$ represent $a\%m = b\%m$
- $(a+b)\%m = ((a\%m) + (b\%m))\%m$
- $(a-b)\%m = ((a\%m) - (b\%m))\%m$
- $(a \times b)\%m = ((a\%m) \times (b\%m))\%m$

Modular Inverse

In the slide above, we can see that modular arithmetic is similar to normal arithmetic. However, there is one thing that is different: division. In modular arithmetic, we cannot simply divide by a number. Instead, we need to find the modular inverse of that number.

$$(a \div b) \mod m \Rightarrow (a \times b^{-1}) \mod m$$

In other words,

$$a \times a^{-1} \equiv 1 \pmod{m}$$

Fermat's Little Theorem

Fermat's Little Theorem states that if p is a prime number and a is an integer not divisible by p, then:

$$a^{p-1} \equiv 1 \pmod{p}$$

This theorem can be used to find modular inverses efficiently.
In the previous slide, we mentioned that the definition of modular inverse is:

$$a \times a^{-1} \equiv 1 \pmod{m}$$

With Fermat's Little Theorem, we can find the modular inverse of a modulo p as:

$$a^{-1} \equiv a^{p-2} \pmod{p}$$

This means that to find the modular inverse of a modulo a prime p, we can simply compute $a^{p-2} \mod p$.

Fermat Prime Test

We utilize Fermat's Little Theorem to check if a number is prime. The basic idea is to randomly select a base a and check if $a^{n-1} \equiv 1 \pmod{n}$ holds true. If it doesn't, then n is composite. Therefore, if we test enough bases, we can conclude that n is prime with high probability. However, this method is not solid enough, as there are some composite numbers that can pass a lot of tests with a high probability.

The time complexity of this test is $O(1)$, depending on how many bases we test. The more bases we test, the higher the probability that we can conclude that n is prime.

Miller-Rabin Primality Test

There's a kind of special composite number that can pass the Fermat's Little Theorem test with a high probability. But they are not prime. These numbers are called Carmichael numbers. To avoid this problem, we can use the Miller-Rabin Primality Test, which is a more robust primality test.

Code for Miller-Rabin Primality Test

```
1   bool millerRabin(ll n, ll a) {
2       if (n < 2) return 0;
3       if ((a = a%n) == 0) return 1;
4       if (n & 1 ^ 1) return n == 2;
5
6       ll tmp = (n - 1) / ((n - 1) & (1 - n));
7       ll t = log2((n - 1) & (1 - n)), x = 1;
8       for (; tmp; tmp >>= 1, a = a*a%n)
9           if (tmp & 1) x = x * a % n;
10      if (x == 1 || x == n - 1) return 1;
11      while (--t)
12          if ((x = x*x%n) == n - 1) return 1;
13      return 0;
14  }
```

Combinations

Combinations are a way to select items from a larger set where the order does not matter.
Given n items, the number of ways to choose k items from n is denoted as $C(n,k)$ or $\binom{n}{k}$, and it is calculated using the formula:

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

How to calculate combinations?

If you directly calculate the factorials, it may lead to overflow or inefficiency for large n. Instead, we can use an idea of dynamic programming. It is known that:

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

With this formula, we can build a table to store the values of combinations.

Code for Combinations

```
1          int C(int n, int k){
2              if(k>n)return 0;
3              if(k==0 or k==n)return 1;
4              return C(n-1, k-1) + C(n-1, k);
5          }
```

However, this code has exponential time complexity. We can use dynamic programming to optimize it.

Calculate with Modular Inverse

If you insist on calculating combinations with large numbers, you can use modular arithmetic to avoid overflow.
We know that the divide operation in modular arithmetic is **Modular Inverse**.
Therefore, we can rewrite the combination formula as:

$$C(n,k) \equiv n! \times (k!)^{-1} \times ((n-k)!)^{-1} \mod m$$

Reference

- https://hackmd.io/@Ccucumber12/Hy8nj-fxt#/
- https://oi-wiki.org/math/