CS3391 Advanced Programming Lab (Week 2)

Yixiong Gao

September 15th, 2023

Primes

Prime Numbers

An integer p $(|p| \geq 1)$ is **prime** if its only divisors are ± 1 and $\pm p$ only.

Otherwise, it is a **composite** number.

e.g. 2,3,5,7 are prime numbers, and 4,6,8,9 are not.

Primality Test

Given a positive integer p, how to determine if it is a prime number?

A Naive Algorithm

Idea: Try to find a divisor of p different to 1 and p.

For all the integers i in $[2, p)$, determine if i is a divisor of p.

Using modulo operation, if `p % i == 0` holds, then i is a divisor of p.

```
bool isprime(int p) {
    for (int i = 2; i < p; ++i)
        if (p % i == 0) return false;
    return true;
}
```

The time complexity is $O(p)$. Can we find a faster one?

A Faster Algorithm

If integer a is a divisor of p, then $p = a \times b$ for some integer b.

Fact: $\min(a, b) \leq \sqrt{p}$

$\Rightarrow$ If each integer in $[2, \sqrt{p}]$ is not a divisor of p, then p is a prime number.

```
bool isprime(int p) {
    int lim = sqrt(p);
    for (int i = 2; i <= lim; ++i)
        if (p % i == 0) return false;
    return true;
}
```

The time complexity is $O(\sqrt{p})$.

It's usually fast enough in most problems in competitive programming.

A Faster Algorithm?

Many faster primality tests are probabilistic tests.

They use some numbers which are chosen at random from some sample space.

it is possible for a composite number to be reported as prime.

The probability of error can be reduced by repeating the test.

Easy to implement: Fermat Test and Miller-Rabin Test.

How to find all prime numbers up to any given limit n?

List of (positive) prime numbers less than 200: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199

A Naive algorithm

Runing primality test for each integer in $[1, n]$.

The time complexity is $O(n\sqrt{n})$.

Sieve of Eratosthenes

1. Make a list of $2, \dots, n$.
2. At first, nobody is marked. And let $p = 2$ as the minimum prime number.
3. Enumerate the multiples of p by counting in increments of p from $2p$ to n, and mark them in the list (these will be $2p, 3p, 4p, \dots$; the p itself should not be marked).
4. Find the smallest number p' in the list greater than p that is not marked. If there was no such number, stop. Otherwise, let $p = p'$ (which is the next prime), and go to step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

Implementation

```
bool mark[N];
for (int p = 2; p <= n; ++p) {
    if (mark[p]) continue;
    for (int q = p * 2; q <= n; q += p) mark[q] = true;
}
```

What is the time complexity of this algorithm?

Time Complexity of Sieve of Eratosthenes

For any prime p , the time complexity of the second loop is $O(n/p)$.

$$\sum_{p \le n, p\ is\ prime} \frac{n}{p} \le \sum_{p \le n} \frac{n}{p} \le \int_1^n \frac{n}{x}\, dx = n \ln n$$

So the time complexity is at least $O(n \ln n)$.

It's usually fast enough in most problems in competitive programming.

Optimization

For every composite number, it will be marked by its minimum divisor for the first time.

For a prime number p, any multiples of p less to $p \times p$ have already been marked.

So we can modify the second loop into:

```
for (int q = p * p; q <= n; q += p) mark[q] = true;
```

Beware of integer overflow.

What is the time complexity now?

One More Question

How many prime numbers are in a given range $[l, r]$?

Try to find an algorithm of time complexity $O(\max((r-l)\ln(r-l), \sqrt{r}\ln\sqrt{r}))$.

I'll provide an algorithm next tutorial session.

Recursion

Going Upstairs

There is a staircase consisting of n steps.

You are now in front of the first step, and you can step up one or two steps at a time.

How many different ways do you have to walk n steps exactly?

E.g. for $n = 3$ there are 3 different ways to going upstairs: $(1, 1, 1), (1, 2), (2, 1)$.

Solution

Assume that there are currently n steps left, then you have only two options:

take one step forward, or take two steps forward if you can.

So we can use recursion to complete the calculation, let that the return value of `f(n)` represents the number of solutions with n steps.

```
int f(int n) {
    if (n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

What's special about this answer?

Codeforces 305 B

Given n integers $a_1, a_2 \ldots, a_n$ and another two integers p and q.

Check if $a_1 + \frac{1}{a_2 + \frac{1}{\cdots + \frac{1}{a_n}}}$ is equal to $\frac{p}{q}$.

Constrains: $n \leq 90, 1 \leq a_i, p, q \leq 10^{18}$

Calculate the values on both sides directly?

No, there will be a loss of precision when makeing a division between floating-point numbers. And making division too many times can lead to significant accuracy errors.

Solution

$$a_1 + \frac{1}{a_2 + \frac{1}{\cdots + \frac{1}{a_n}}} = \frac{p}{q} \quad \Leftrightarrow \frac{1}{a_2 + \frac{1}{\cdots + \frac{1}{a_n}}} = \frac{p}{q} - a_1 = \frac{p - a_1 q}{q}$$

$$\Leftrightarrow a_2 + \frac{1}{a_3 + \frac{1}{\cdots + \frac{1}{a_n}}} = \frac{q}{p - a_1 q}$$

The current problem is in the same form as before.

so we can using a recursion that maintaining the numerator and denominator of the right side without any loss of precision.

Implementation

```
bool check(int i, int p, int q) {
    if (i == n) return a[n] * q == p;
    return check(i + 1, q, p - a[i] * q);
}
```

Note that this is just an example. In fact, you may need to implement a type yourself to support the operation of huge numbers.

Thanks for Listening!

Contact: yixiongao3-c@my.cityu.edu.hk