ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC ĐÀ NẴNG
KHOA CÔNG NGHỆ THÔNG TIN

Môn học

Toán Ứng Dụng
Applied mathematics

Giảng Viên: Phạm Minh Tuấn

Giới thiệu

- Phạm Minh Tuấn
- E-mail: pmtuan@dut.udn.vn
- Tel: 0913230910
- Khoa Công nghệ thông tin –Trường ĐHBK – ĐHĐN

ITF

Contents

- Number Theory (Lý thuyết số)
  - Basic Number Theory
  - Primality Tests
  - Totient Function
- Combinatorics (Tổ hợp)
  - Basics of Combinatorics
  - Inclusion-Exclusion
- Geometry (Hình học)
- Game Theory (Lý thuyết trò chơi)

Bài 2:

Basic Number Theory (Con't)

Basic Number Theory (cont’t)

- Introduction
  - The concept of prime numbers is a very important concept in math. This article discusses the concept of prime numbers and related properties. It includes the following topics:
    - What are prime numbers and composite numbers?
    - Prime and factors
    - Sieve of Eratosthenes
    - Modification of Sieve of Eratosthenes for fast factorization

What are prime numbers?

What are prime numbers?

- Prime numbers are those numbers that are greater than 1 and have only two factors 1 and itself.
- Composite numbers are also numbers that are greater than 1 but they have at least one more divisor other than 1 and itself.
- For example,
  - 5 is a prime number because 5 is divisible only by 1 and 5 only.
  - However, 6 is a composite number as 6 is divisible by 1, 2, 3, and 6.
- There are various methods to check whether a number is prime.

Primes and factors

- A number a is called a **factor** or a **divisor** of a number b if a divides b. If a is a factor of b, we write a | b. For example, the factors of 24 are 1, 2, 3, 4, 6, 8, 12 and 24.
- A number n > 1 is a **prime** if its only positive factors are 1 and n. For example, 7, 19 and 41 are primes, but 35 is not a prime, because 5*7 = 35. For every number n > 1, there is a unique prime factorization

$$\boxed{n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},}$$

- where $p_1, p_2, \ldots, p_k$ are distinct primes and $\alpha_1, \alpha_2, \ldots, \alpha_k$ are positive numbers. For example, the prime factorization for 84 is $84 = 2^2 * 3^1 * 7^1$

Number of factors

- The **number of factors** of a number n is

$$\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1)$$

- because for each prime p_i, there are $\alpha_i + 1$ ways to choose how many times it appears in the factor.
- For example,
  - the number of factors of 84 is 3*2*2 = 12.The factors are 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 and 84.

Sum of factors

- The **sum of factors** of a number n is

$$\boxed{\sigma(n) = \prod_{i=1}^{k}(1 + p_i + \ldots + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1} - 1}{p_i - 1}}$$

- where the latter formula is based on the geometric progression formula.
- For example, the sum of factors of 84 is.

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224$$

Product of factors

- The **product of factors** of a number n is

$$\boxed{\mu(n) = n^{\tau(n)/2}}$$

- because we can form τ(n)/2 pairs from the factors, ach with product n.
- For example, the factors of 84 is
  - The factors are 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 and 84.
  - the factors of 84 produce the pairs 1×84, 2×42, 3×28, etc., and the product of the factors is $\mu(84)=8^{46}$=351298031616.

TEST YOUR UNDERSTANDING

- Given a number and check if a number is perfect or not. A number is said to be perfect if sum of all its factors excluding the number itself is equal to the number.
- **Input**:
  - First line consists of **T** test cases. Then **T** test cases follow .Each test case consists of a number **N**.
- **Output**:
  - Output in a single line 1 if a number is a perfect number else print 0.
- **Constraints**:
  - $1 \le T \le 300$
  - $1 \le N \le 10^5$.

Hint - Perfect number

- In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum).
- Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself) i.e. $\sigma(n) = 2n$

Number of primes

- It is easy to show that there is an infinite number of primes. If the number of primes would be finite, we could construct a set P = $\{p_1, p_2, \ldots, p_n\}$ that would contain all the primes. For example, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, and so on. However, using P, we could form a new prime

$$\boxed{p_1 p_2 \cdots p_n + 1}$$

- that is larger than all elements in P. This is a contradiction, and the number of primes has to be infinite.

Density of primes

- The density of primes means how often there are primes among the numbers.
- Let $\pi(n)$ denote the number of primes between 1 and n. For example, $\pi(10) = 4$, because there are 4 primes between 1 and 10: 2, 3, 5 and 7.
- It is possible to show that

$$\boxed{\pi(n) \approx \frac{n}{\ln n}}$$

- which means that primes are quite frequent.
- For example, the number of primes between 1 and 106 is $\pi(10^6) = 78498 \approx 10^6 / \ln 106 = 72382$.

Conjectures

- There are many conjectures involving primes. Most people think that the conjectures are true, but nobody has been able to prove them. For example, the following conjectures are famous:
  - **Goldbach's conjecture**: Each even integer $n > 2$ can be represented as a sum $n = a + b$ so that both a and b are primes.
  - **Twin prime conjecture**: There is an infinite number of pairs of the form $\{p, p + 2\}$, where both p and $p+2$ are primes.
  - **Legendre's conjecture**: There is always a prime between numbers n^2 and $(n+1)^2$, where n is any positive integer.

Naive approach

- Traverse all the numbers from 1 to N and count the number of divisors. If the number of divisors is equal to 2 then the given number is prime, else it is not.

```
void checkprime(int N){
        int count = 0;
        for( int i = 1;i <= N;++i )
                if( N % i == 0 )
                        count++;
        if(count == 2)
                cout << N << " is a prime number." << endl;
        else
                cout << N << " is not a prime number." << endl;
}
```

- The time complexity of this function is O(N)

Better approach

- If you have two positive numbers N and D, such that N is divisible by D and D is less than the square root of N.
  - (N/D) must be greater than the square root of N.
  - N is also divisible by (N/D). If there is a divisor of N that is less than the square root of N, then there will be a divisor of N that is greater than square root of N. You will have to traverse till the square root of N.
- Note: You are generating all the divisors of N and if the count of divisors is greater than 2, then the number is composite.

Better approach

- For example,
  - If N=50, $\sqrt{N}$=7 (floor value).
  - You will iterate from 1 to 7 and count the number of divisors of N.
  - The divisors of N are 1, 50; 2, 25; 5,10.
  - You have 6 divisors of 50, and therefore, it is not prime.

Better approach

```
void checkprime(int N) {
        int count = 0;
        for( int i = 1;i * i <=N;++i ) {
                if( N % i == 0) {
                        if( i * i == N )
                            count++;
                        else       // i < sqrt(N) and (N / i) > sqrt(N)
                            count += 2;
                 }
        }
        if(count == 2)
                cout << N << " is a prime number." << endl;
        else
                cout << N << " is not a prime number." << endl;
}
```

- The time complexity of this function is $O(\sqrt{N})$

Sieve of Eratosthenes

- *Sieve of Eratosthenes* is an algorithm to find all the prime numbers that are less than or equal to a given number N or to find out whether a number is a prime number.
- The basic idea behind the Sieve of Eratosthenes is that at each iteration one prime number is picked up and all its multiples are eliminated. After the elimination process is complete, all the unmarked numbers that remain are prime.

Sieve of Eratosthenes

- *Pseudo code*

```
1. Mark all the numbers as prime numbers except 1
2. Traverse over each prime numbers smaller than sqrt(N)
3. For each prime number, mark its multiples as composite numbers
4. Numbers, which are not the multiples of any number, will remain marked as prime number and others will change to composite numbers.
```

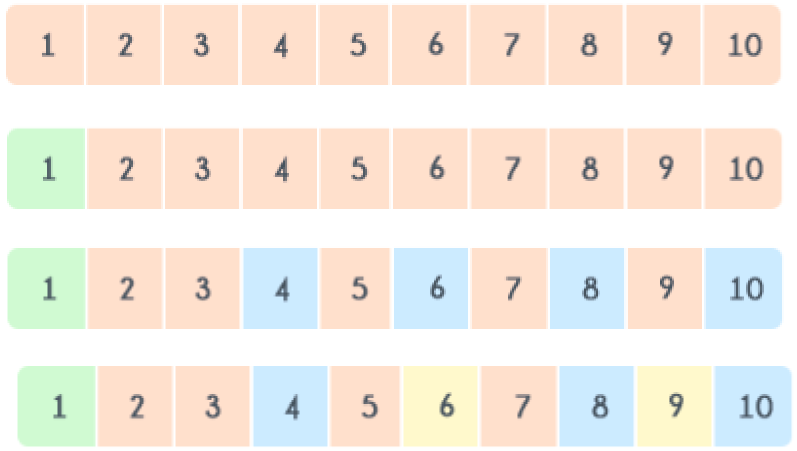
Sieve of Eratosthenes

```
void sieve(int N) {
        bool isPrime[N+1];
        for(int i = 0; i <= N;++i) {
                isPrime[i] = true;
        }
        isPrime[0] = false;
        isPrime[1] = false;
        for(int i = 2; i * i <= N; ++i) {
                if(isPrime[i] == true) {
                        for(int j = i * i; j <= N ;j += i)
                                isPrime[j] = false;
                }
        }
}
```

- This code will compute all the prime numbers that are smaller than or equal to N.

Sieve of Eratosthenes

- Let us compute prime numbers where N=10.
  - Mark all the numbers as prime
  - Mark 1 as composite
- In each iteration, check if a number is prime or not, if it is then mark all of its multiple as composite.



Mark 1 as composite

Mark multiples of 2 as composite

Mark multiples of 3 as composite

- The prime numbers are 2, 3, 5, and 7.

Sieve of Eratosthenes

- Time complexity
  - The inner loop that runs for each element is as follows:
    - If i = 2, inner loop runs N / 2 times
    - If i = 3, inner loop runs N / 3 times
    - If i = 5, inner loop runs N / 5 times
  - *Total complexity*
    - $N * (\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \ldots) = O(N\log\log N)$

Sieve of Eratosthenes for fast factorization

```
vector<int> factorize(int n) {
    vector<int> res;
    for (int i = 2; i * i <= n; ++i) {
        while (n % i == 0) {
            res.push_back(i);
            n /= i;
        }
    }
    if (n != 1) {
        res.push_back(n);
    }
    return res;
}
```

```
n= 50, i =2; res = {2}
n= 25, i =3; res = {2}
n= 25, i =4; res = {2}
n= 25, i =5; res = {2,5,5}
n= 1
```

At every step, the prime number of the least value, which divides the current N. This is the main idea of this modification.

The time complexity of this function is $O(\sqrt{N})$

Factorize N in *O(log(N))* time.

```
int minPrime[n + 1];
for (int i = 2; i * i <= n; ++i) {
    if (minPrime[i] == 0) {//If i is prime
        for (int j = i * i; j <= n; j += i) {
            if (minPrime[j] == 0) {
                minPrime[j] = i;
            }
        }
    }
}
for (int i = 2; i <= n; ++i) {
    if (minPrime[i] == 0) {
        minPrime[i] = i;
    }
}
```

```
vector<int> factorize(int n) {
    vector<int> res;
    while (n != 1) {
        res.push_back(minPrime[n]);
        n /= minPrime[n];
    }
    return res;
}
```

```
n = 50, minPrime[50] = 2; res = {2}
n = 25, minPrime[25] = 5; res = {2,5}
n = 5, minPrime[25] = 5; res= {2,5,5}
n = 1
```

The required factors are in *res* .

Factorize N in *O(log(N))* time.

- Conditions
  - You can implement this modification only if you are allowed to create an array of integers of size N.
- Note:
  - This approach is useful when you need to factorize not-very-large numbers but, need to calculate factorization many times.
  - It is not necessary to build a modified Sieve for every problem in which factorization is required. Moreover, you cannot build it for large numbers N like 10^9 or 10^{12}. Therefore, for such large numbers it is recommended that you factorize in $O(\sqrt{N})$ instead.

Sieve of Eratosthenes on the segment

- Sometimes you need to find all the primes that are in the range [L...R] and not in [1...N], where R is a large number.
- Conditions
  - You are allowed to create an array of integers with size (R−L+1).

Implemention

```
bool isPrime[r - l + 1]; //filled by true
for (long long i = 2; i * i <= r; ++i) {
    for (long long j = max(i * i, (l + (i - 1)) / i  * i); j <= r; j += i) {
        isPrime[j - l] = false;
    }
}
for (long long i = max(l, 2); i <= r; ++i) {
    if (isPrime[i - l]) {
        //then i is prime
    }
}
```

- The approximate complexity is O($\sqrt{N}$)

Suggestions

- It is recommended that you do not build a Sieve to check several numbers for primality. Use the following function instead, which works in O($\sqrt{N}$) for every number:

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

TEST YOUR UNDERSTANDING

- Given a number N, find number of primes in the range [1,N].
- **Input**:
  - The only line of input consists of a number *N*.
- **Output**:
  - Print the number of primes in the range [1,N].
- **Constraints**:
  - $1 \le N \le 10^6$

I T F

Code

```
#include <bits/stdc++.h>
using namespace std;
int sieve(int N) {
		int count = 0;
		bool isPrime[N+1];
		for(int i = 0; i <= N;++i) {
				isPrime[i] = true;
		}
		isPrime[0] = false;
		isPrime[1] = false;
		for(int i = 2; i * i <= N; ++i) {
				if(isPrime[i] == true) {
						for(int j = i * i; j <= N ;j += i)
						isPrime[j] = false;
				}
		}
		for (int i = 0; i <= N; ++i) if (isPrime[i]) count++;
    return count;
}
```

```
int main() {
		int N;
		cin >> N;
		cout << sieve(N);
}
```

Problems