Math

Primality Tests

A natural number N is said to be a prime number if it can be divided only by 1 and itself. Primality Testing is done to check if a number is a prime or not. The topic explains different algorithms available for primality testing.

Basic Method:

This is an approach that goes in a way to convert definition of prime numbers to code. It checks if any of the number less than a given number(N) divides the number or not. But on observing the factors of any number, this method can be limited to check only till $\sqrt{N}$. This is because, product of any two numbers greater than $\sqrt{N}$ can never be equal to N. A C++ function for basic method is shown below.

```
int PrimeTest(int N)
{
    for (int i = 2; i*i <= N; ++i)
    {
        if(N%i == 0)
        {
            return 0;
        }
    }
    return 1;
}
```

The function returns 1 if N is a prime number and 0 for a composite number. This function runs with a complexity of $O(\sqrt{n})$. That implies, this method can at most be used for numbers of range 10^{15} to 10^{16} to determine if it's a prime or not in reasonable amount of time.

One major application of prime numbers are that they are used in cryptography. One of the standard cryptosystem - RSA algorithm uses a prime number as key which is usually over 1024 bits to ensure greater security. When dealing with such large numbers, definitely doesn't make the above mentioned method any good. Also, should be noticed that it is not easy to work with such large numbers especially when the operations performed are `/` and `%` at the time of primality testing. Thus most primality testing algorithms that are developed can only determine if the given number is a "probable prime" or composite. Couple of widely used of these algorithms are explained below.

Sieve of Eratosthenes:

This is a simple algorithm useful in finding all the prime numbers up to a given number(N). The algorithm takes all the numbers from 2 to N all initially unmarked. It starts from 2. If the number is unmarked, mark all its multiples $\leq N$ as composites. The performance can be improved by doing the above operation only till $\sqrt{N}$ and all the numbers in range $[2, N]$ that remained unmarked are primes. The reason that we can stop after doing the iterations only till $\sqrt{N}$ is that, no composites $\leq N$ would have a prime factor greater than $\sqrt{N}$.

A pseudocode for this algorithm is as below

```
A[N] = {0}

for i from 2 to sqrt(N):
    if A[i] = 0:
        for j from 2 to N:
            if i*j > N:
                break
            A[i*j] = 1
```

In the final array, starting from 2, if for any index, value is 0, it is a prime, else is a composite.

Fermat Primality Testing:

This testing is based on Fermat's Little Theorem. The theorem states that, given a prime number P, and any number a (where $0 < a < p$), then $a^{p-1} \equiv 1 mod p$.

In Fermat Primality Testing, k random integers are selected as the value of X (where all k integers follow $0 < X < p$). If the statement of Fermat's Little Theorem is accepted for all these k values of X for a given number N, then N is said as a probable prime. Pseudocode for Fermat primality testing is as below.

```
function: FermatPrimalityTesting(int N):
    pick a random integer k      //not too less. not too high.
```

```
    LOOP: repeat k times:
        pick a random integer X in range (1,N-1)
        if(X^(N-1)%N != 1):
            return composite
    return probably prime
```


Miller-Rabin Primality Testing:

Similar to Fermat primality test, Miller-Rabin primality test could only determine if a number is a probable prime.

It is based on a basic principle where if $X^2 \equiv Y^2 mod N$, but $X! \equiv \pm Y mod N$, then N is composite.

The algorithm in simple steps can be written as,

Given a number $N (> 2)$ for which primality is to be tested,
Step 1: Find $N - 1 = 2^R . D$
Step 2: Choose A in range $[2, N - 2]$
Step 3: Compute $X_0 = A^D mod N$. If X_0 is ± 1, N can be prime.
Step 4: Compute $X_i = X_i - 1 mod N$. If $X_i = 1$, N is composite.
If $X_i = -1$, N is prime.
Step 5: Repeat **Step 4** for $R - 1$ times.
Step 6: If neither -1 or $+1$ appeared for X_i, N is composite.

Pseudocode for Miller-Rabin primality testing is given below.

```
function: MillerRabin_PrimalityTesting(int N):
    if N%2 = 0:
        return composite

    write N-1 as (2^R . D) where D is odd number
    pick a random integer k     //not too less. not too high.

    LOOP: repeat k times:
        pick a random integer A in range [2,N-2]
        X = A^D % N
        if X = 1 or X = N-1:
            continue LOOP
        for i from 1 to r-1:
            X = X^2 % N
            if X = 1:
                return composite
            if X = N-1:
                continue LOOP
```

```
        return composite
    return probably prime
```

There are other methods too like AKS primality test, Lucas primality test which predicts if a number could be prime number or not. A method called Elliptic curve primality testing proves if a given number is prime, unlike predicting in the above mentioned methods.

Contributed by: Vinay Kumar

Did you find this tutorial helpful? YES NO

TEST YOUR UNDERSTANDING

Prime number

Given an integer(N), write a code to check if it is prime or not.

Input Format:
First line has an integer T - number of test cases.
Each test case is in a new line with a single integer N.

Output Format:
Print "**prime**" if N is prime, "**composite**" if N is not a prime. Answer for each test case should be printed in a new line.

Constraints:

- $2 \leq T \leq 100$
- $1 \leq N \leq 10^{16}$

SAMPLE INPUT

```
5
13
9
27
325
23
```

SAMPLE OUTPUT

```
prime
composite
composite
composite
prime
```

7 LIVE EVENTS

Enter your code or Upload your code as file. Save C (gcc 4.8.2)

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

1:1

Provide custom input

COMPILE & TEST SUBMIT

Press Ctrl-space for autocomplete suggestions. POWERED BY code table

COMMENTS (6) SORT BY: Relevance

Join Discussion...

Cancel Post

Amulya Gaur 3 months ago

which method should be used here

1 vote • Reply • Message • Permalink

Akhilesh Soni 13 days ago

Normal sqrt(n) method

0 votes • Reply • Message • Permalink

Amulya Gaur 13 days ago

time out!

0 votes • Reply • Message • Permalink

Akhilesh Soni Edited 13 days ago

It got accepted for me

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;
int main()
{
ios::sync_with_stdio(false);
int t;
cin >> t;
while(t--)
{
ll n;
cin >> n;
ll i;
bool flag=false;
for(i=2;i<=sqrt(n);i++)
{
if(n%i==0)
{
flag=true;
break;
}
}
if(flag)
cout << "composite\n";
else
cout << "prime\n";
}
return 0;
}
```

1 vote • Reply • Message • Permalink

Manish PERIWAL 3 months ago

Plz help me, it is saying error from line 47

http://ideone.com/XRWtYp

0 votes • Reply • Message • Permalink

threat 2 months ago

When you calculate X^2(mod N) you get overflow, you can use this code

http://ideone.com/nba3Wc

0 votes • Reply • Message • Permalink

About Us
Hackathons
Talent Solutions
University Program
Developers Wiki
Blog
Press
Careers
Reach Us

7
LIVE EVENTS