**Primality Tests**

* **TUTORIAL**
* [**PROBLEMS**](https://www.hackerearth.com/practice/math/number-theory/primality-tests/practice-problems/)

A natural number NN is said to be a prime number if it can be divided only by 11 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(NN) divides the number or not. But on observing the factors of any number, this method can be limited to check only till √NN. This is because, product of any two numbers greater than √NN can never be equal to NN. 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 11 if NN is a prime number and 00 for a composite number. This function runs with a complexity of O(√n)O(n). That implies, this method can at most be used for numbers of range 10151015 to 10161016 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 10241024 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(NN). The algorithm takes all the numbers from 22 to NN all initially unmarked. It starts from 22. If the number is unmarked, mark all its multiples ≤N≤N as composites. The performance can be improved by doing the above operation only till √NN and all the numbers in range [2,N][2,N] that remained unmarked are primes. The reason that we can stop after doing the iterations only till √NN is that, no composites ≤N≤N would have a prime factor greater than √NN.

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 PP, and any number aa (where 0<a<p0<a<p), then ap−1≡1modpap−1≡1modp.

In Fermat Primality Testing, kk random integers are selected as the value of XX (where all kk integers follow 0<X<p0<X<p). If the statement of Fermat's Little Theorem is accepted for all these kk values of XX for a given number NN, then NN 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 X2≡Y2modNX2≡Y2modN, but X!≡±YmodNX!≡±YmodN, then NN is composite.

The algorithm in simple steps can be written as,

Given a number NN(>2>2) for which primality is to be tested,  
**Step 1:** Find N−1=2R.DN−1=2R.D  
**Step 2:** Choose AA in range [2,N−2][2,N−2]  
**Step 3:** Compute X0=ADmodNX0=ADmodN. If X0X0 is ±1±1, NN can be prime.  
**Step 4:** Compute Xi=Xi−1modNXi=Xi−1modN. If Xi=1Xi=1, NN is composite.  
If Xi=−1Xi=−1, NN is prime.  
**Step 5:** Repeat **Step 4** for R−1R−1 times.  
**Step 6:** If neither −1−1 or +1+1 appeared for XiXi, NN 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](https://www.hackerearth.com/@vinaykumar" \t "blank)*

**Did you find this tutorial helpful?**

 YES

 NO

**TEST YOUR UNDERSTANDING**

**Prime number**

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

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

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

**Constraints:**

* 2≤T≤1002≤T≤100
* 1≤N≤10161≤N≤1016

**SAMPLE INPUT**

5

13

9

27

325

23

**SAMPLE OUTPUT**

prime

composite

composite

composite

prime

<https://www.hackerearth.com/practice/math/number-theory/primality-tests/tutorial/>

def **\_try\_composite**(a, d, n, s):

if pow(a, d, n) == 1:

return False

for i in range(s):

if pow(a, 2\*\*i \* d, n) == n-1:

return False

return True # n is definitely composite

def **is\_prime**(n, \_precision\_for\_huge\_n=16):

if n in \_known\_primes or n in (0, 1):

return True

if any((n % p) == 0 for p in \_known\_primes):

return False

d, s = n - 1, 0

while not d % 2:

d, s = d >> 1, s + 1

# Returns exact according to http://primes.utm.edu/prove/prove2\_3.html

if n < 1373653:

return not any(\_try\_composite(a, d, n, s) for a in (2, 3))

if n < 25326001:

return not any(\_try\_composite(a, d, n, s) for a in (2, 3, 5))

if n < 118670087467:

if n == 3215031751:

return False

return not any(\_try\_composite(a, d, n, s) for a in (2, 3, 5, 7))

if n < 2152302898747:

return not any(\_try\_composite(a, d, n, s) for a in (2, 3, 5, 7, 11))

if n < 3474749660383:

return not any(\_try\_composite(a, d, n, s) for a in (2, 3, 5, 7, 11, 13))

if n < 341550071728321:

return not any(\_try\_composite(a, d, n, s) for a in (2, 3, 5, 7, 11, 13, 17))

# otherwise

return not any(\_try\_composite(a, d, n, s)

for a in \_known\_primes[:\_precision\_for\_huge\_n])

\_known\_primes = [2, 3]

\_known\_primes += [x for x in range(5, 1000, 2) if is\_prime(x)]

t = int(raw\_input())

while t > 0:

n = int(raw\_input())

if is\_prime(n):

print *"prime"*

else: print *"composite"*

t-=1

---TAMBIEN ACEPTADO---

static string isPrime(long n)

{

if (n < 2) return "composite";

else if (n == 2) return "prime";

else if (n % 2 == 0) return "composite";

long sqr = (long)Math.Sqrt(n);

for (long i = 3; i <= sqr; i += 2)

{

if (n % i == 0) return "composite";

}

return "prime";

}