⇒ Facts about prime no.'s

① '2' is the only even prime.

② Every prime can be written as $6n+1$ or $6n-1$, except 2 & 3, n being a natural number.

③ Goldbach Conjecture :- Every even integer greater than 2 can be expressed as the sum of two primes.

④ Wilson's Theorem :-

$$(p-1)!\,\%\,p \Rightarrow (p-1) \bmod p$$

⑤ 2 & 3 are only consecutive no.'s that are prime.

wilson's e.g

$p = 5$

$(5-1)! = 4!$

$4! = 24$

$24 \,\%\, 5 = 4$

and $(p-1)\,\%\, p = (5-1)\,\%\, 5 = 4.$

Hence, $(p-1)!\,\%\, p = (p-1)\,\%\, p$

⇒ A no. is prime if it is divisible by 1 & itself and has exactly 2 factors.

// code

```
count = 0;
for (i = 1; i<=n; i++){
    if (n % i == 0)
        count++;
}
if (count == 2)   print (Yes)
else              print (NO).
```

O(n)

// optimised code

factors

36 ⇒

| i | | n/i |
|---|---|---|
| | 1 x 36 | |
| | 2 x 18 | |
| | 3 x 12 | |
| | 4 x 9 | |
| | 6 x 6 | |
| | 9 x 4 | Repeated |
| | 12 x 3 | |
| | 18 x 2 | |
| | 36 x 1 | |

So, if I iterate for sqrt times of n, then for every factor i, the another factor will be n/i & hence, no need for running the loop for n times.

```
for (i=1 ; i<= sqrt(n); i++){
    if (n % i == 0){
        count ++;
        if ((n/i) != i)   // most imp' condition
        { count ++ ;}
    }
}
if (count == 2) { print (prime) ;}
else {print (not prime); } ;
```

if the other factor is not same then we have to take 2 times the one factor: i.e, i & n/i

$T \Rightarrow O(\sqrt{n} \times \log n)$

The sqrt(n) function takes logarithmic time to calculate value of $\sqrt{n}$,
∴ the time is $O(\sqrt{n} \log n)$.

But instead of sqrt(n), we write (i*i <=n) → it will take square root time.

i.e, $O(\sqrt{n})$

So always write (i*i <=n)

Q:- Print sum of factors of n.

```
sum = 0;
for (int i = 0; i*i <=n ; i++){
    if (n % i == 0){
        sum += i;
        if ((n/i) != i)}
            sum += (n/i);
    }
}
```

## Q: Product of three numbers

find three numbers a, b & c, such that a, b, c ≥ 2 & abc = n.

Constraints ⇒ $\boxed{2 \le n \le 10^9}$

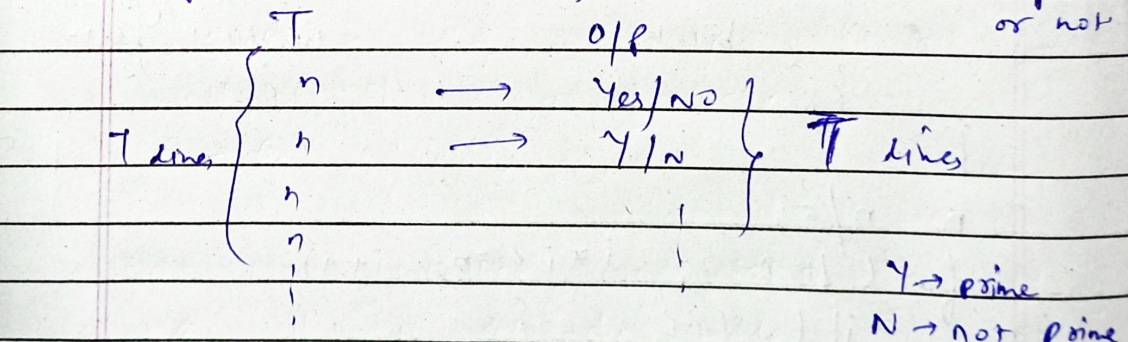
⇒ If we find the smallest factor, then we can able to b * & c, 'b' being the next smallest & c will be n/(b*a).

```
int n;
cin >> n;
int a=n, b=n, c=n;
for(i=2; i*i<=n; i++){
    if (n%i==0){
        a=i;                    // It's definitely the
        break;                  smallest factor
    }
}
n = n/a;
for(int i=2; i*i<=n; i++)
    if(n%i==0){
        if(i!=a){
            b= min(b,i);
        }
        if((n/i)!=i){
            if((n/i)!=a){
                b= min(b, n/i);
            }
        }
    }
}
```

```
c = n/b;
if(a!=b && b!=c && c!=a && a>1 && b>1 && c>1)
{
    cout <<" Yes\n";
    cout<<a<<" "<<b<<" "<<c<<endl;
}
else {
    cout << "No\n";
}
}..
```

⇒ Concept of Sieve

Suppose, in a problem, there are T test cases & every test have T lines of input & T lines of o/p. and we have to print Y/N depending upon whether n is prime or not




Let's take $T <= 10^6$
& $N <= 10^6$.

If we use our previous method.

```
bool checkPrime(int n){
    // code  → O(√n)
}

main(){
    while(t--){
        // code  → O(T·√n)
    }
}
```

T in the worst case
$T = 10^6$ & $n = 10^6$

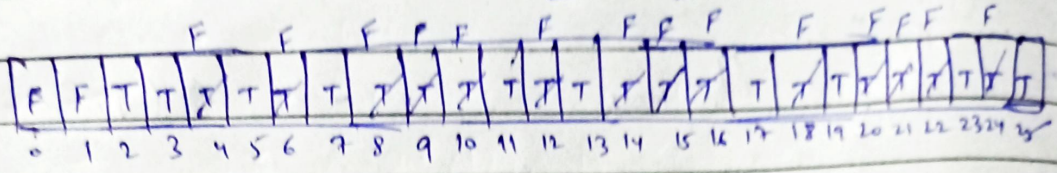
⇒ Time = $O(10^6 \cdot 10^3) \simeq 10^9$

In 1s, $\simeq 10^8$ operations can be computed, so, $10^9$ will take 10 sec, which will get a TLE.

// So, here, we will use Sieve concept. Sieve is like a box, which takes input as n and tell us that it is prime or not in very very less time than $\sqrt{n}$.

// Sieve can be a boolean array or integer array which contains T & F, T denotes prime no. & F denotes non-prime.

let's assume $n <= 26$ (for easier understanding)




fill every index as True ~~instead~~ except of 1, bez of 1 are not primes (as the smallest prime starts from 2).

Now, start from 2.
Since, 2 is a prime no, so all other multiples of it are obviously composite no.

So, we will make all the multiples of 2 as false.
$2 \rightarrow$ 4, 6, 8, 10, 12, - - -

now, take 3, & similarly make all the multiples of 3 as false.
$3 \rightarrow$ 6, 9, 12, 15, 21, - - -

~~Now~~, Now, whether we take 4? No!! bez, it is already made false by 2, and it is understandable that all the multiples of 4 are ultimately factors of 2, which had already been marked false.

Similarly, traverse the array, pick the prime no. & make its multiples false.

Now, the remaining 'true's are the prime no's.
∴, we will pass the value n, & in the sieve table, just check the index n, whether it's true of false & return the value correspondingly.

// code

```
int N = 10000000;
bool sieve [10000001];
void CreateSieve () {
    for (int i=2; i<=N; i++){
        sieve [i] = true;          // mark everything as true.
    }

    for (int i= 2; i<=N; i++){
        if (sieve [i] == true){
            // if the index is marked is marked as true, then only go & mark its multiples as false.

// The first multiple of any no. is obviously 2, & the next multiple is at i^th distance from i
            for(int j= 2*i; j<=N; j+=i){
                sieve [j] = false;
            }
        }
    }

// This code will take O(n^2) :
   let's optimize it.
```

→ 1st optimization:

Multiples of 2 → 4, 6, 8, 10, 12, ---

3 → (6), 9, (12), 15, (18), ---

These no's are already marked by 2, so it is really useless to again mark by 3.

So, instead of 6, we can start from 9.

5 ⇒ 10, 15, 20, 25, 30, ---

marked by 2 | marked by 3 | marked by 2.

We can start marking multiples of 5 from 25 instead of 10.

In general, for any no. (i), we can start marking its multiple from (i * i), bcz, before that, other multiples had been marked by some other no.

So, instead of starting from $(2 * i)^{th}$ multiple, we will start from $(i * i)^{th}$ multiple

→ 2nd optimization

here, we have $n <= 26$,
Now, for $n = 6$, its multiple would start from $6 \times 6 = 36 > 26$
for 7, ⇒ $7 \times 7 = 49 > 26$
for 8, ⇒ $8 \times 8 = 64 > 26$.

we can see, it's useless to count for those multiples which don't even lie in the sieve array.

Therefore, rather than running the outer loop for n times, we have to run it only square root times & our job is done.

```
// optimized code
void createSieve () {
    for( int i=2; i<=N; i++) {
        sieve [i] = true;
    }
    for( int i= 2; i*i <= N; i++){
        if (sieve [i] == true) {
            for (int j = i*i ; j<= N ; j+=i) {
                sieve [j] = false;
            }
        }
    }
}
```

Time of createSieve funct.

$= O( n \cdot \log(\log n))$

```
int main(){
    createSieve();   // Create the sieve array
                     before the test cases.
    int t; cin>>t;
    while(t--){
        int n; cin>>n;
        if(sieve[n] == true){
            cout<<"Yes\n";
        }
        else{
            cout<<"No\n";
        }
    }
}
```

Time complexity = $O(n \log(\log n)) + O(T)$

This is almost $n$ or less than $n$

which is very very less.

// Max size of array declared inside main

int / double ⇒ $10^6$

boolean ⇒ $10^7$

// Max size of array declared globally.

int / double ⇒ $10^7$

boolean ⇒ $10^8$