Bitwise Sieve

Status Array:

- 0th bit of status[0] ⇒ indicates primality of 0
- 1st bit of status[0] ⇒ indicates primality of 1
- 2nd bit of status[0] ⇒ indicates primality of 2
- 0th bit of status[1] ⇒ indicates primality of 32
- 1st bit of status[1] ⇒ indicates primality of 33

Bitwise Operations:

- `i >> 5` is equivalent to $\frac{i}{2^5} = \frac{i}{32}$ which is much faster, it has brief description below[1]
- `j & 32` is equivalent to `i % 32`

```
#define mx 1000
int status[(mx/32)+2];
bool Check(int N,int pos){return (bool)(N & (1<<pos));}
int Set(int N,int pos){ return N=N | (1<<pos);}

void sieve()
{
    int i, j, sqrtN;
    sqrtN = int( sqrt( mx ) );
    for( i = 3; i <= sqrtN; i += 2 )
    {
    // in our code -> 0 means prime, 1 means not prime.
        if( Check(status[i>>5],i&31)==0)
        {
            for( j = i*i; j <= mx; j += (i<<1) )
            {
                status[j>>5]=Set(status[j>>5],j & 31)   ;
            }
        }
    }
}
bool isPrime(int n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    // in our code -> 0 means prime, 1 means not prime.
    return !Check(status[n >> 5], n&31);
}
int main() {
    sieve(); int n;
    n = 3; cout << n << ": " <<  (isPrime(n) ? " prime" : " NOT prime") << endl;
    n = 5; cout << n << ": " <<  (isPrime(n) ? " prime" : " NOT prime") << endl;
    n = 27; cout << n << ": " <<  (isPrime(n) ? " prime" : " NOT prime") << endl;
}
```

[1]Bitwise and in place of modulus operator:

$$x\%2^n \equiv x\&(2^{n-1})$$

$$x\%4 \equiv x\&3 \text{ , } x\%8 \equiv x\&7 \text{ and so on}$$

Though it's actually not accurate to say that `x % 2 == x & 1`. Simple counterexample: `x = -1`. In many languages, including Java, `-1 % 2 == -1`. That is, % is not necessarily the traditional mathematical definition of modulo. Java calls it the **"remainder operator"**, for example.