Binary Exponentiation

Binary exponentiation (also known as exponentiation by squaring) is a trick which allows to calculate a^n using only $O(\log n)$ multiplications (instead of $O(n)$ multiplications required by the naive approach).

It also has important applications in many tasks unrelated to arithmetic, since it can be used with any operations that have the property of **associativity**:

$$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$$

Most obviously this applies to modular multiplication, to multiplication of matrices and to other problems which we will discuss below.

Algorithm

Raising a to the power of n is expressed naively as multiplication by a done $n-1$ times: $a^n = a \cdot a \cdot \ldots \cdot a$. However, this approach is not practical for large a or n.

$a^{b+c} = a^b \cdot a^c$ and $a^{2b} = a^b \cdot a^b = (a^b)^2$.

The idea of binary exponentiation is, that we split the work using the binary representation of the exponent.

Let's write n in base 2, for example:

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

Since the number n has exactly $\lfloor \log_2 n \rfloor + 1$ digits in base 2, we only need to perform $O(\log n)$ multiplications, if we know the powers $a^1, a^2, a^4, a^8, \dots, a^{\lfloor \log n \rfloor}$.

So we only need to know a fast way to compute those. Luckily this is very easy, since an element in the sequence is just the square of the previous element.

$$\begin{aligned}
3^1 &= 3 \\
3^2 &= \left(3^1\right)^2 = 3^2 = 9 \\
3^4 &= \left(3^2\right)^2 = 9^2 = 81 \\
3^8 &= \left(3^4\right)^2 = 81^2 = 6561
\end{aligned}$$

So to get the final answer for 3^{13}, we only need to multiply three of them (skipping 3^2 because the corresponding bit in n is not set): $3^{13} = 6561 \cdot 81 \cdot 3 = 1594323$

The final complexity of this algorithm is $O(\log n)$: we have to compute $\log n$ powers of a, and then have to do at most $\log n$ multiplications to get the final answer from them.

The following recursive approach expresses the same idea:

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

Implementation

First the recursive approach, which is a direct translation of the recursive formula:

```
long long binpow(long long a, long long b) {
    if (b == 0)
        return 1;
    long long res = binpow(a, b / 2);
    if (b % 2)
        return res * res * a;
    else
        return res * res;
}
```

The second approach accomplishes the same task without recursion. It computes all the powers in a loop, and multiplies the ones with the corresponding set bit in n. Although the complexity of both approaches is identical, this approach will be faster in practice since we have the overhead of the recursive calls.

```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

Applications

Effective computation of large exponents modulo a number

Problem: Compute $x^n \bmod m$. This is a very common operation. For instance it is used in computing the modular multiplicative inverse.

Solution: Since we know that the module operator doesn't interfere with multiplications ($a \cdot b \equiv (a \bmod m) \cdot (b \bmod m) \pmod m$), we can directly use the same code, and just replace every multiplication with a modular multiplication:

```
long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

Note: If m is a prime number we can speed up a bit this algorithm by calculating $x^{n \bmod (m-1)}$ instead of x^n. This follows directly from Fermat's little theorem.

Euclidean algorithm for computing gcd

The algorithm is extremely simple:

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Implementation

```
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

Using the ternary operator in C++, we can write it as a one-liner.

```
int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}
```

And finally, here is a non-recursive implementation:

```
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

Correctness Proof

First, notice that in each iteration of the Euclidean algorithm the second argument strictly decreases, therefore (since the arguments are always non-negative) the algorithm will always terminate.

For the proof of correctness, we need to show that $\gcd(a,b) = \gcd(b, a \bmod b)$ for all $a \geq 0$, $b > 0$.

We will show that the value on the left side of the equation divides the value on the right side and vice versa. Obviously, this would mean that the left and right sides are equal, which will prove Euclid's algorithm.

Let $d = \gcd(a,b)$. Then by definition $d \mid a$ and $d \mid b$.

Now let's represent the remainder of the division of a by b as follows:

$$a \bmod b = a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor$$

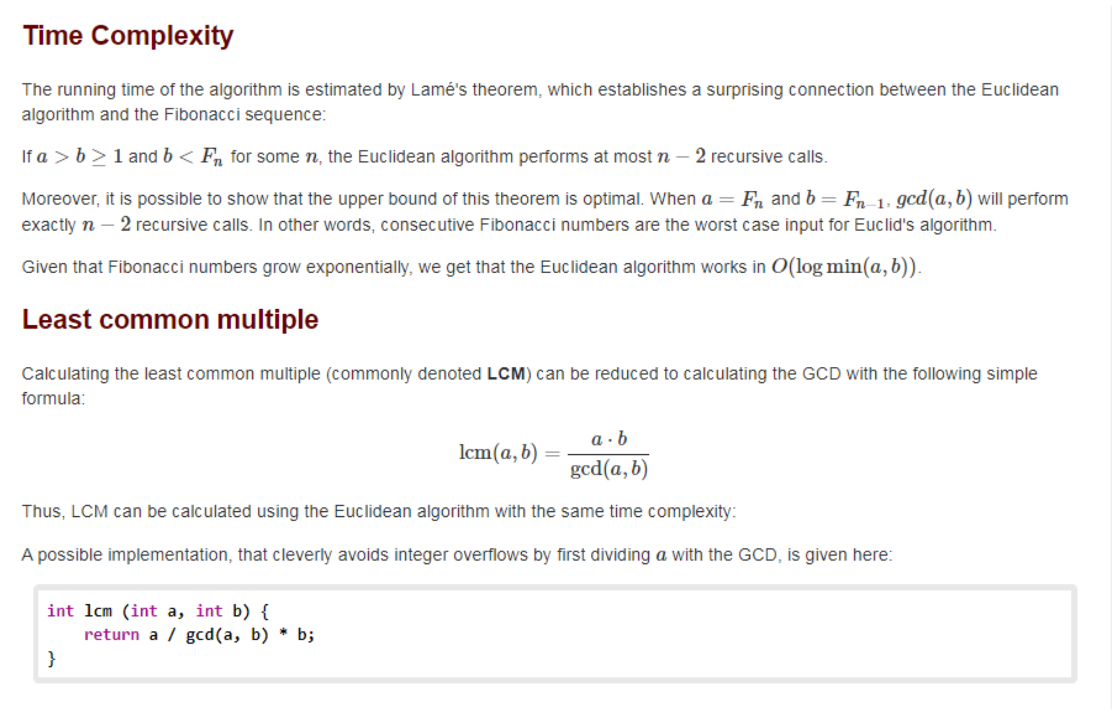
From this it follows that $d \mid (a \bmod b)$, which means we have the system of divisibilities:

$$\begin{cases} d \mid b, \\ d \mid (a \mod b) \end{cases}$$

Now we use the fact that for any three numbers p, q, r, if $p \mid q$ and $p \mid r$ then $p \mid \gcd(q,r)$. In our case, we get:

$$d = \gcd(a,b) \mid \gcd(b, a \mod b)$$

Thus we have shown that the left side of the original equation divides the right. The second half of the proof is similar.



Time Complexity

The running time of the algorithm is estimated by Lamé's theorem, which establishes a surprising connection between the Euclidean algorithm and the Fibonacci sequence:

If $a > b \geq 1$ and $b < F_n$ for some n, the Euclidean algorithm performs at most $n - 2$ recursive calls.

Moreover, it is possible to show that the upper bound of this theorem is optimal. When $a = F_n$ and $b = F_{n-1}$, $gcd(a, b)$ will perform exactly $n - 2$ recursive calls. In other words, consecutive Fibonacci numbers are the worst case input for Euclid's algorithm.

Given that Fibonacci numbers grow exponentially, we get that the Euclidean algorithm works in $O(\log \min(a, b))$.

Least common multiple

Calculating the least common multiple (commonly denoted **LCM**) can be reduced to calculating the GCD with the following simple formula:

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Thus, LCM can be calculated using the Euclidean algorithm with the same time complexity:

A possible implementation, that cleverly avoids integer overflows by first dividing a with the GCD, is given here:

```
int lcm (int a, int b) {
    return a / gcd(a, b) * b;
}
```

Extended Euclidean Algorithm

While the Euclidean algorithm calculates only the greatest common divisor (GCD) of two integers a and b, the extended version also finds a way to represent GCD in terms of a and b, i.e. coefficients x and y for which:

$$a \cdot x + b \cdot y = \gcd(a,b)$$

It's important to note, that we can always find such a representation, for

instance $\gcd(55,80)=5$ therefore we can represent 5 as a linear combination with the

terms 55 and 80: $55 \cdot 3 + 80 \cdot (-2) = 5$

A more general form of that problem is discussed in the article about Linear Diophantine Equations. It will build upon this algorithm.

Algorithm

We will denote the GCD of a and b with g in this section.

The changes to the original algorithm are very simple. If we recall the algorithm, we can see that the algorithm ends with $b = 0$ and $a = g$. For these parameters we can easily find coefficients, namely $g \cdot 1 + 0 \cdot 0 = g$.

Starting from these coefficients $(x, y) = (1, 0)$, we can go backwards up the recursive calls. All we need to do is to figure out how the coefficients x and y change during the transition from (a, b) to $(b, a \bmod b)$.

Let us assume we found the coefficients (x_1, y_1) for $(b, a \bmod b)$:

$$b \cdot x_1 + (a \bmod b) \cdot y_1 = g$$

and we want to find the pair (x, y) for (a, b):

$$a \cdot x + b \cdot y = g$$

We can represent $a \bmod b$ as:

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b$$

Substituting this expression in the coefficient equation of (x_1, y_1) gives:

$$g = b \cdot x_1 + (a \bmod b) \cdot y_1 = b \cdot x_1 + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) \cdot y_1$$

and after rearranging the terms:

$$g = a \cdot y_1 + b \cdot \left(x_1 - y_1 \cdot \left\lfloor \frac{a}{b} \right\rfloor\right)$$

We found the values of x and y:

$$\begin{cases} x = y_1 \\ y = x_1 - y_1 \cdot \left\lfloor \frac{a}{b} \right\rfloor \end{cases}$$

Implementation

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

The recursive function above returns the GCD and the values of coefficients to `x` and `y` (which are passed by reference to the function).

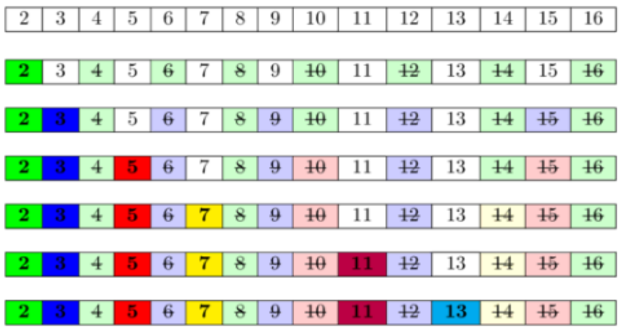
This implementation of extended Euclidean algorithm produces correct results for negative integers as well.

Sieve of Eratosthenes

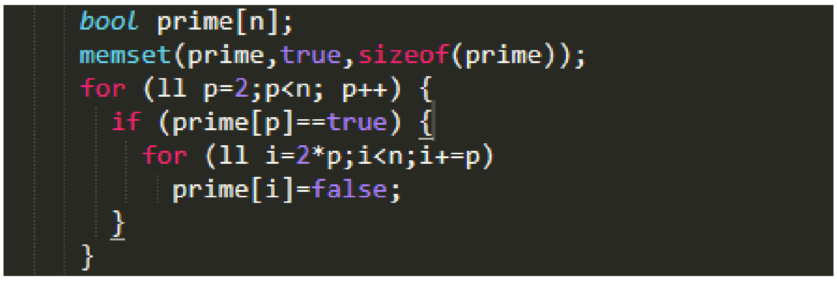
Sieve of Eratosthenes is an algorithm for finding all the prime numbers in a segment $[1; n]$ using $O(n \log \log n)$ operations.

The algorithm is very simple: at the beginning we write down all numbers between 2 and n. We mark all proper multiples of 2 (since 2 is the smallest prime number) as composite. A proper multiple of a number x, is a number greater than x and divisible by x. Then we find the next number that hasn't been marked as composite, in this case it is 3. Which means 3 is prime, and we mark all proper multiples of 3 as composite. The next unmarked number is 5, which is the next prime number, and we mark all proper multiples of it. And we continue this procedure until we processed all numbers in the row.

In the following image you can see a visualization of the algorithm for computing all prime numbers in the range $[1; 16]$. It can be seen, that quite often we mark numbers as composite multiple times.



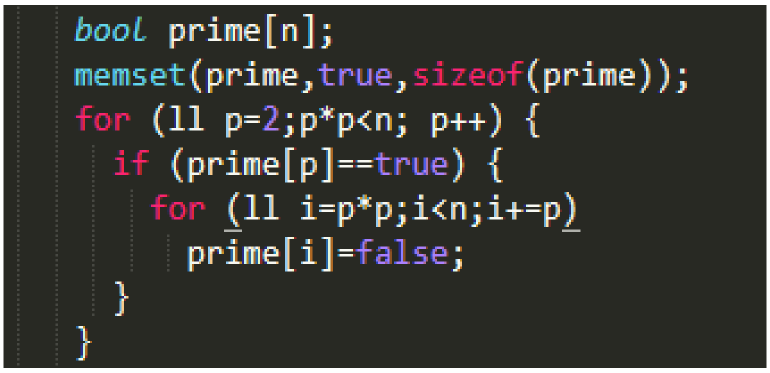
Implementation



```
bool prime[n];
memset(prime,true,sizeof(prime));
for (ll p=2;p<n; p++) {
    if (prime[p]==true) {
        for (ll i=2*p;i<n;i+=p)
            prime[i]=false;
    }
}
```

Sieving till root

Obviously, to find all the prime numbers until n, it will be enough just to perform the shifting only by the prime numbers, which do not exceed the root of n.

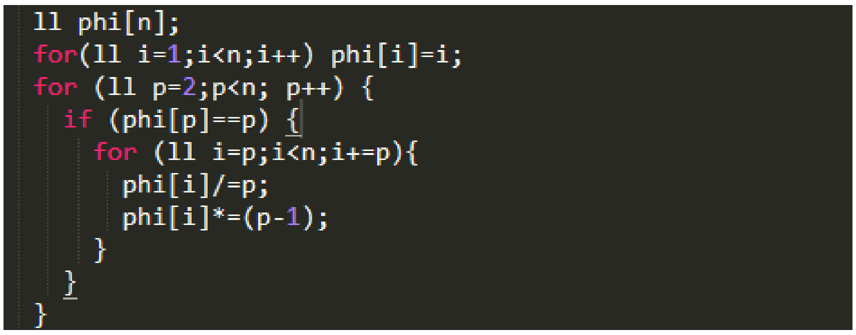


```
bool prime[n];
memset(prime,true,sizeof(prime));
for (ll p=2;p*p<n; p++) {
    if (prime[p]==true) {
        for (ll i=p*p;i<n;i+=p)
            prime[i]=false;
    }
}
```

Sieve to calculate totient function

(Should know the O(sqrt(n) method to calculate totient function)

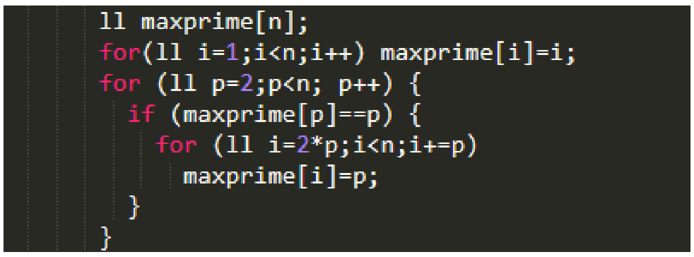
We will proceed similar to sieve of Eratosthenes. We find all prime numbers and for each prime number, we update the temporary results of all numbers that are divisible by that prime number.



```
ll phi[n];
for(ll i=1;i<n;i++) phi[i]=i;
for (ll p=2;p<n; p++) {
  if (phi[p]==p) {
    for (ll i=p;i<n;i+=p){
      phi[i]/=p;
      phi[i]*=(p-1);
    }
  }
}
```

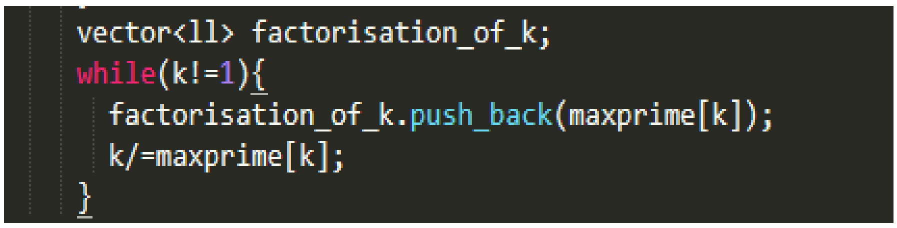
Finding prime factorisation using sieve

We can just store the largest prime number dividing a number for all numbers till n.



```
ll maxprime[n];
for(ll i=1;i<n;i++) maxprime[i]=i;
for (ll p=2;p<n; p++) {
  if (maxprime[p]==p) {
    for (ll i=2*p;i<n;i+=p)
      maxprime[i]=p;
  }
}
```

To factorise a given number k, we can do this:



```
vector<ll> factorisation_of_k;
while(k!=1){
  factorisation_of_k.push_back(maxprime[k]);
  k/=maxprime[k];
}
```

Practice Problems

1. Find the largest integer that can't be represent as x*7+y*11 where x,y>=0.
2. Show that gcd(2*a,2*b)=2*gcd(a,b). Also show that gcd(2*a,b)=gcd(a,b) if b is odd.
3. Two fraction a/b and p/q are given, find the largest fraction that divides both of them.
4. Primality test (hackerearth)
5. Gcd queries(codechef)
6. And 0 sum big (codeforces)
7. Crucial equation (SPOJ)
8. Get AC in one go (codechef)
9. Given 3 integers a,b,n , find gcd of a^n+b^n and a-b where $1 <= b < a <= 10^{12}$ and $1<= n <=10^{12}$.
10. Short Task(codeforces)
11. Neko does maths (codeforces)
12. Math (codeforces)
13. Divisors(codeforces)
14. Power Products (codeforces)
15. Infinite path(codeforces)
16. The holmes children(codeforces)