Last update: June 8, 2022 Original

Montgomery Multiplication

Many algorithms in number theory, like prime testing or integer factorization, and in cryptography, like RSA, require lots of operations modulo a large number. A multiplications like $xy \bmod n$ is quite slow to compute with the typical algorithms, since it requires a division to know how many times n has to be subtracted from the product. And division is a really expensive operation, especially with big numbers.

The **Montgomery (modular) multiplication** is a method that allows computing such multiplications faster. Instead of dividing the product and subtracting n multiple times, it adds multiples of n to cancel out the lower bits and then just discards the lower bits.

Montgomery representation

However the Montgomery multiplication doesn't come for free. The algorithm works only in the **Montgomery space**. And we need to transform our numbers into that space, before we can start multiplying.

For the space we need a positive integer $r \geq n$ coprime to n, i.e. $\gcd(n, r) = 1$. In practice we always choose r to be 2^m for a positive integer m, since multiplications, divisions and modulo r operations can then be efficiently implemented using shifts and other bit operations. n will be an odd number in pretty much all applications, since it is not hard to factorize an even number. So every power of 2 will be coprime to n.

The representative $\bar{x}$ of a number x in the Montgomery space is defined as:

$$\bar{x} := x \cdot r \bmod n$$

Notice, the transformation is actually such a multiplication that we want to optimize. So this is still an expensive operation. However you only need to transform a number once into the space. As soon as you are in the Montgomery space, you can perform as many operations as you want efficiently. And at the end you transform the final result back. So as long as you are doing lots of operations modulo n, this will be no problem.

Inside the Montgomery space you can still perform most operations as usual. You can add two elements ($x \cdot r + y \cdot r \equiv (x + y) \cdot r \bmod n$), subtract, check for equality, and even compute the greatest common divisor of a number with n (since $\gcd(n, r) = 1$). All with the usual algorithms.

However this is not the case for multiplication.

We expect the result to be:

$$\bar{x} * \bar{y} = \overline{x \cdot y} = (x \cdot y) \cdot r \bmod n.$$

But the normal multiplication will give us:

$$\bar{x} \cdot \bar{y} = (x \cdot y) \cdot r \cdot r \bmod n.$$

Therefore the multiplication in the Montgomery space is defined as:

$$\bar{x} * \bar{y} := \bar{x} \cdot \bar{y} \cdot r^{-1} \bmod n.$$

Montgomery reduction

The multiplication of two numbers in the Montgomery space requires an efficient computation of $x \cdot r^{-1} \bmod n$. This operation is called the **Montgomery reduction**, and is also known as the algorithm **REDC**.

Because $\gcd(n, r) = 1$, we know that there are two numbers r^{-1} and n' with $0 < r^{-1}, n' < n$ with

$$r \cdot r^{-1} + n \cdot n' = 1.$$

Both r^{-1} and n' can be computed using the Extended Euclidean algorithm.

Using this identity we can write $x \cdot r^{-1}$ as:

$$\begin{aligned} x \cdot r^{-1} &= x \cdot r \cdot r^{-1}/r = x \cdot (-n \cdot n' + 1)/r \\ &= (-x \cdot n \cdot n' + x)/r \equiv (-x \cdot n \cdot n' + l \cdot r \cdot n + x)/r \bmod n \\ &\equiv ((-x \cdot n' + l \cdot r) \cdot n + x)/r \bmod n \end{aligned}$$

The equivalences hold for any arbitrary integer l. This means, that we can add or subtract an arbitrary multiple of r to $x \cdot n'$, or in other words, we can compute $q := x \cdot n'$ modulo r.

This gives us the following algorithm to compute $x \cdot r^{-1} \bmod n$:

```
function reduce(x):
    q = (x mod r) * n' mod r
    a = (x - q * n) / r
    if a < 0:
        a += n
    return a
```

Since $x < n \cdot n < r \cdot n$ (even if x is the product of a multiplication) and $q \cdot n < r \cdot n$ we know that $-n < (x - q \cdot n)/r < n$. Therefore the final modulo operation is implemented using a single check and one addition.

As we see, we can perform the Montgomery reduction without any heavy modulo operations. If we choose r as a power of 2, the modulo operations and divisions in the algorithm can be computed using bitmasking and shifting.

A second application of the Montgomery reduction is to transfer a number back from the Montgomery space into the normal space.

Fast inverse trick

For computing the inverse $n' := n^{-1} \bmod r$ efficiently, we can use the following trick (which is inspired from the Newton's method):

$$a \cdot x \equiv 1 \bmod 2^k \implies a \cdot x \cdot (2 - a \cdot x) \equiv 1 \bmod 2^{2k}$$

This can easily be proven. If we have $a \cdot x = 1 + m \cdot 2^k$, then we have:

$$\begin{aligned} a \cdot x \cdot (2 - a \cdot x) &= 2 \cdot a \cdot x - (a \cdot x)^2 \\ &= 2 \cdot (1 + m \cdot 2^k) - (1 + m \cdot 2^k)^2 \\ &= 2 + 2 \cdot m \cdot 2^k - 1 - 2 \cdot m \cdot 2^k - m^2 \cdot 2^{2k} \\ &= 1 - m^2 \cdot 2^{2k} \\ &\equiv 1 \bmod 2^{2k}. \end{aligned}$$

This means we can start with $x = 1$ as the inverse of a modulo 2^1, apply the trick a few times and in each iteration we double the number of correct bits of x.

Implementation

Using the GCC compiler we can compute $x \cdot y \bmod n$ still efficiently, when all three numbers are 64 bit integer, since the compiler supports 128 bit integer with the types `__int128` and `__uint128`.

```
long long result = (__int128)x * y % n;
```

However there is no type for 256 bit integer. Therefore we will here show an implementation for a 128 bit multiplication.

```
using u64 = uint64_t;
using u128 = __uint128_t;
using i128 = __int128_t;

struct u256 {
    u128 high, low;

    static u256 mult(u128 x, u128 y) {
        u64 a = x >> 64, b = x;
        u64 c = y >> 64, d = y;
        // (a*2^64 + b) * (c*2^64 + d) =
        // (a*c) * 2^128 + (a*d + b*c)*2^64 + (b*d)
        u128 ac = (u128)a * c;
        u128 ad = (u128)a * d;
        u128 bc = (u128)b * c;
        u128 bd = (u128)b * d;
        u128 carry = (u128)(u64)ad + (u128)(u64)bc + (bd >> 64u);
        u128 high = ac + (ad >> 64u) + (bc >> 64u) + (carry >> 64u);
```

```
        u128 low = (ad << 64u) + (bc << 64u) + bd;
        return {high, low};
    }
};

struct Montgomery {
    Montgomery(u128 n) : mod(n), inv(1) {
        for (int i = 0; i < 7; i++)
            inv *= 2 - n * inv;
    }

    u128 init(u128 x) {
        x %= mod;
        for (int i = 0; i < 128; i++) {
            x <<= 1;
            if (x >= mod)
                x -= mod;
        }
        return x;
    }

    u128 reduce(u256 x) {
        u128 q = x.low * inv;
        i128 a = x.high - u256::mult(q, mod).high;
        if (a < 0)
            a += mod;
        return a;
    }

    u128 mult(u128 a, u128 b) {
        return reduce(u256::mult(a, b));
    }

    u128 mod, inv;
};
```

Fast transformation

The current method of transforming a number into Montgomery space is pretty slow. There are faster ways.

You can notice the following relation:

$$\bar{x} := x \cdot r \bmod n = x \cdot r^2/r = x * r^2$$

Transforming a number into the space is just a multiplication inside the space of the number with r^2. Therefore we can precompute $r^2 \bmod n$ and just perform a multiplication instead of shifting the number 128 times.

In the following code we initialize `r2` with `-n % n`, which is equivalent to $r - n \equiv r \bmod n$, shift it 4 times to get $r \cdot 2^4 \bmod n$. This number can be interpreted as 2^4 in Montgomery space. If we square it 5 times, we get $(2^4)^{2^5} = (2^4)^{32} = 2^{128} = r$ in Montgomery space, which is exactly $r^2 \bmod n$.

```
struct Montgomery {
    Montgomery(u128 n) : mod(n), inv(1), r2(-n % n) {
        for (int i = 0; i < 7; i++)
```

```
            inv *= 2 - n * inv;

        for (int i = 0; i < 4; i++) {
            r2 <<= 1;
            if (r2 >= mod)
                r2 -= mod;
        }
        for (int i = 0; i < 5; i++)
            r2 = mul(r2, r2);
    }

    u128 init(u128 x) {
        return mult(x, r2);
    }

    u128 mod, inv, r2;
};
```

Contributors:

jakobkogler (96.81%) adamant-pwn (2.28%) kyomukyomupurin (0.46%) Prakash-Jha-Dev (0.46%)