Last update: June 8, 2022 Original

Binary Exponentiation by Factoring

Consider a problem of computing $ax^y \pmod{2^d}$, given integers a, x, y and $d \geq 3$, where x is odd.

The algorithm below allows to solve this problem with $O(d)$ additions and binary operations and a single multiplication by y.

Due to the structure of the multiplicative group modulo 2^d, any number x such that $x \equiv 1 \pmod 4$ can be represented as

$$x \equiv b^{L(x)} \pmod{2^d},$$

where $b \equiv 5 \pmod 8$. Without loss of generality we assume that $x \equiv 1 \pmod 4$, as we can reduce $x \equiv 3 \pmod 4$ to $x \equiv 1 \pmod 4$ by substituting $x \mapsto -x$ and $a \mapsto (-1)^y a$. In this notion, ax^y is represented as

$$ax^y \equiv ab^{yL(x)} \pmod{2^d}.$$

The core idea of the algorithm is to simplify the computation of $L(x)$ and $b^{yL(x)}$ using the fact that we're working modulo 2^d. For reasons that will be apparent later on, we'll be working with $4L(x)$ rather than $L(x)$, but taken modulo 2^d instead of 2^{d-2}.

In this article, we will cover the implementation for 32-bit integers. Let

- `mbin_log_32(r, x)` be a function that computes $r + 4L(x) \pmod{2^d}$;
- `mbin_exp_32(r, x)` be a function that computes $rb^{\frac{x}{4}} \pmod{2^d}$;
- `mbin_power_odd_32(a, x, y)` be a function that computes $ax^y \pmod{2^d}$.

Then `mbin_power_odd_32` is implemented as follows:

```
uint32_t mbin_power_odd_32(uint32_t rem, uint32_t base, uint32_t exp) {
    if (base & 2) {
        /* divider is considered negative */
        base = -base;
        /* check if result should be negative */
        if (exp & 1) {
            rem = -rem;
        }
```

```
    }
    return (mbin_exp_32(rem, mbin_log_32(0, base) * exp));
}
```

Computing 4L(x) from x

Let x be an odd number such that $x \equiv 1 \pmod 4$. It can be represented as

$$x \equiv (2^{a_1} + 1) \dots (2^{a_k} + 1) \pmod{2^d},$$

where $1 < a_1 < \dots < a_k < d$. Here $L(\cdot)$ is well-defined for each multiplier, as they're equal to 1 modulo 4. Hence,

$$4L(x) \equiv 4L(2^{a_1} + 1) + \dots + 4L(2^{a_k} + 1) \pmod{2^d}.$$

So, if we precompute $t_k = 4L(2^n + 1)$ for all $1 < k < d$, we will be able to compute $4L(x)$ for any number x.

For 32-bit integers, we can use the following table:

```
const uint32_t mbin_log_32_table[32] = {
    0x00000000, 0x00000000, 0xd3cfd984, 0x9ee62e18,
    0xe83d9070, 0xb59e81e0, 0xa17407c0, 0xce601f80,
    0xf4807f00, 0xe701fe00, 0xbe07fc00, 0xfc1ff800,
    0xf87ff000, 0xf1ffe000, 0xe7ffc000, 0xdfff8000,
    0xffff0000, 0xfffe0000, 0xfffc0000, 0xfff80000,
    0xfff00000, 0xffe00000, 0xffc00000, 0xff800000,
    0xff000000, 0xfe000000, 0xfc000000, 0xf8000000,
    0xf0000000, 0xe0000000, 0xc0000000, 0x80000000,
};
```

On practice, a slightly different approach is used than described above. Rather than finding the factorization for x, we will consequently multiply x with $2^n + 1$ until we turn it into 1 modulo 2^d. In this way, we will find the representation of x^{-1}, that is

$$x(2^{a_1} + 1) \dots (2^{a_k} + 1) \equiv 1 \pmod{2^d}.$$

To do this, we iterate over n such that $1 < n < d$. If the current x has n-th bit set, we multiply x with $2^n + 1$, which is conveniently done in C++ as `x = x + (x << n)`. This won't change bits lower than n, but will turn the n-th bit to zero, because x is odd.

With all this in mind, the function `mbin_log_32(r, x)` is implemented as follows:

```
uint32_t mbin_log_32(uint32_t r, uint32_t x) {
    uint8_t n;

    for (n = 2; n < 32; n++) {
        if (x & (1 << n)) {
            x = x + (x << n);
            r -= mbin_log_32_table[n];
        }
    }

    return r;
}
```

Note that $4L(x) = -4L(x^{-1})$, so instead of adding $4L(2^n + 1)$, we subtract it from r, which initially equates to 0.

Computing x from 4L(x)

Note that for $k \geq 1$ it holds that

$$(a2^k + 1)^2 = a^2 2^{2k} + a2^{k+1} + 1 = b2^{k+1} + 1,$$

from which (by repeated squaring) we can deduce that

$$(2^a + 1)^{2^b} \equiv 1 \pmod{2^{a+b}}.$$

Applying this result to $a = 2^n + 1$ and $b = d - k$ we deduce that the multiplicative order of $2^n + 1$ is a divisor of 2^{d-n}.

This, in turn, means that $L(2^n + 1)$ must be divisible by 2^n, as the order of b is 2^{d-2} and the order of b^y is 2^{d-2-v}, where 2^v is the highest power of 2 that divides y, so we need

$$2^{d-k} \equiv 0 \pmod{2^{d-2-v}},$$

thus v must be greater or equal than $k - 2$. This is a bit ugly and to mitigate this we said in the beginning that we multiply $L(x)$ by 4. Now if we know $4L(x)$, we can uniquely decomposing it into a sum of $4L(2^n + 1)$ by consequentially checking bits in $4L(x)$. If the n-th bit is set to 1, we will multiply the result with $2^n + 1$ and reduce the current $4L(x)$ by $4L(2^n + 1)$.

Thus, `mbin_exp_32` is implemented as follows:

```
uint32_t mbin_exp_32(uint32_t r, uint32_t x) {
    uint8_t n;
```

```
    for (n = 2; n < 32; n++) {
        if (x & (1 << n)) {
            r = r + (r << n);
            x -= mbin_log_32_table[n];
        }
    }

    return r;
}
```

Further optimizations

It is possible to halve the number of iterations if you note that $4L(2^{d-1}+1)=2^{d-1}$ and that for $2k \geq d$ it holds that

$$(2^n+1)^2 \equiv 2^{2n} + 2^{n+1} + 1 \equiv 2^{n+1}+1 \pmod{2^d},$$

which allows to deduce that $4L(2^n+1)=2^n$ for $2n \geq d$. So, you could simplify the algorithm by only going up to $\frac{d}{2}$ and then use the fact above to compute the remaining part with bitwise operations:

```
uint32_t mbin_log_32(uint32_t r, uint32_t x) {
    uint8_t n;

    for (n = 2; n != 16; n++) {
        if (x & (1 << n)) {
            x = x + (x << n);
            r -= mbin_log_32_table[n];
        }
    }

    r -= (x & 0xFFFF0000);

    return r;
}

uint32_t mbin_exp_32(uint32_t r, uint32_t x) {
    uint8_t n;

    for (n = 2; n != 16; n++) {
        if (x & (1 << n)) {
            r = r + (r << n);
            x -= mbin_log_32_table[n];
        }
    }

    r *= 1 - (x & 0xFFFF0000);
```

```
    return r;
}
```

Computing logarithm table

To compute log-table, one could modify the Pohlig–Hellman algorithm for the case when modulo is a power of 2.

Our main task here is to compute x such that $g^x \equiv y \pmod{2^d}$, where $g = 5$ and y is a number of kind $2^n + 1$.

Squaring both parts k times we arrive to

$$g^{2^k x} \equiv y^{2^k} \pmod{2^d}.$$

Note that the order of g is not greater than 2^d (in fact, than 2^{d-2}, but we will stick to 2^d for convenience), hence using $k = d - 1$ we will have either g^1 or g^0 on the left hand side which allows us to determine the smallest bit of x by comparing y^{2^k} to g. Now assume that $x = x_0 + 2^k x_1$, where x_0 is a known part and x_1 is not yet known. Then

$$g^{x_0 + 2^k x_1} \equiv y \pmod{2^d}.$$

Multiplying both parts with g^{-x_0}, we get

$$g^{2^k x_1} \equiv (g^{-x_0} y) \pmod{2^d}.$$

Now, squaring both sides $d - k - 1$ times we can obtain the next bit of x, eventually recovering all its bits.

References

- M30, Hans Petter Selasky, 2009

Contributors:

adamant-pwn (85.71%) hselasky (14.29%)