Computing binomial coefficients modulo integers quickly

Simon Lindholm

2020-06-01

Abstract

We compare different techniques for modular multiplications, and apply this to compute binomial coefficients quickly.

This paper is written with the intention of becoming part of a series about SIMD within competitive programming, based around different example problems. However, it may be of interest outside of the competitive programming sphere as well.

Contents

1 Introduction

Modular multiplication comes up frequently within competitive programming, and it is sometimes useful to be able to do it quickly, e.g. to get away with bad time complexities. Using binomial coefficient computation as a running example, we show how to improve performance using data-parallel loops, SIMD and different algorithms for modular multiplication. We will first deal with the easy case of computing them mod 2^{32}, and then modulo arbitrary integers in the range $[1, 10^9]$.

As usual for competitive programming, we shall have little regard to portability and assume GCC and x86_64, and use a small file header to keep code size down (and in the case of `rep` also reduce typo-induced bugs):

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i,f,t) for (int i = (f); i < (t); i++)
typedef int64_t ll;
typedef uint32_t u32;
typedef uint64_t u64;
typedef __uint128_t u128;
```

All benchmarking performed will be highly unscientific, with measurements being done on an old laptop running an i5-4200U Haswell processor, and times eyeballed medians of relatively high-variance measurements. That said, the overall conclusions should still be valid.

Source code is available at GitHub.

2 Modulo powers of two

(Russian training camp) Given N, K $(0 \leq K \leq N \leq 10^{18})$, compute

$$\binom{N}{K} = \frac{N!}{K!(N-K)!} \pmod{2^{32}}.$$

Time limit: 1 second.

This problem seems to call for some extension of Lucas' theorem to prime powers... but let's ignore that and instead focus our attention on how to compute factorials quickly.

We can't compute $N!$, $K!$ and $(N-K)!$ straight off modulo 2^{32} and then do modular division, because they will be divisible by large powers of two, and division by zero is impossible. Instead, let's take each factorial and write it as $x! = x_{\text{odd}} 2^{x_{\text{even}}}$. Then

$$\frac{N!}{K!(N-K)!} = \frac{N_{\text{odd}}}{K_{\text{odd}}(N-K)_{\text{odd}}} \cdot 2^{N_{\text{even}} - K_{\text{even}} - (N-K)_{\text{even}}},$$

and the x_{odd} parts can be computed mod 2^{32} instead of requiring arbitrary precision integers. We can get formulas for x_{odd} and x_{even} by expanding $x!$ as follows:

$$\begin{array}{cccccccccccccc}
x! & = & 1 & \cdot 2 & \cdot 3 & \cdot 4 & \cdot 5 & \cdot 6 & \cdot 7 & \cdot 8 & \cdot 9 & \cdot 10 & \cdot \ldots \\
 & = & \mathbf{1} & & \mathbf{\cdot 3} & & \mathbf{\cdot 5} & & \mathbf{\cdot 7} & & \mathbf{\cdot 9} & & \cdot \ldots \\
 & & & \cdot 2 \cdot \mathbf{1} & & \cdot 2 & & \cdot 2 \cdot \mathbf{3} & & \cdot 2 & & \cdot 2 \cdot \mathbf{5} & \cdot \ldots \\
 & & & & & \cdot 2 \cdot \mathbf{1} & & & & \cdot 2 & & & \cdot \ldots \\
 & & & & & & & & & \cdot 2 \cdot \mathbf{1} & & & \cdot \ldots \\
 & & \cdot \ldots & & & & & & & & & &
\end{array}$$

x_{even} gets an easy formula: $\sum_{k=1}^{\infty} \lfloor x/2^k \rfloor$. For x_{odd}, we get a decomposition into a number of odd double factorials $k!! = 1 \cdot 3 \cdot 5 \cdot \ldots \cdot k$.

As it turns out, $1 \cdot 3 \cdot \ldots \cdot (2^{32} - 1) = 1 \pmod{2^{32}}$, so $k!! = (k \mod 2^{32})!! \pmod{2^{32}}$.

Now, we could try to compute all the double factorials with bruteforce, but it would be a bit too costly – there are up to $3 \cdot \log_2 10^{18}$ of them. But we can be a bit smarter, by reusing parts of the products. Let's partition the numbers $1, 3, \ldots, 2^{32} - 1$ into intervals that affect the same double factorials. Then we get an amortized runtime of 2^{31} multiplications, which should be totally doable in 1 second. (Note that arithmetic mod 2^{32} is fast, because it can make use of integer overflow.) Here is how it looks in code:

```
u32 modpow(u32 a, ll e) {
  if (e == 0) return 1;
  u32 x = modpow(a * a, e >> 1);
  return e & 1 ? x * a : x;
}

u32 solve(ll N, ll K) {
  ll trailingZeroes = 0;
  map<u32, int> ivs;
  int accMult = 0;
  rep(i,0,3) {
    ll x = (i == 0 ? N : i == 1 ? K : N-K);
    int mult = (i == 0 ? 1 : -1);
    while (x > 0) {
      // Include the product (1 * 3 * ... * (x % 2^32))
      // in the answer, 'mult' times.
      ivs[(u32)x + 1] += mult;
      accMult += mult;
      x /= 2;
      trailingZeroes += x * mult;
    }
  }

  u32 cur = 0, res = 1, resdiv = 1;
  for (auto pa : ivs) {
    u32 lim = pa.first, ilim = lim / 2;
    // The odd numbers in the range [last lim, lim) get
    // included 'accMult' times in the answer -- 'pa.second'
    // times for the interval ending at 'lim', and also for
    // all larger intervals. We divide the loop counter by 2
    // to avoid potential overflows.
    u32 prod = 1;
    for (; cur < ilim; cur++)
      prod *= cur * 2 + 1;
    if (accMult > 0) res *= modpow(prod, accMult);
    else resdiv *= modpow(prod, -accMult);
    accMult -= pa.second;
  }

  res *= modpow(2, trailingZeroes);
  res *= modpow(resdiv, (1LL << 31) - 1);
  return res;
}
```

Running this locally on worst-case input (e.g. $n = 2^{59} - 2$, k randomly keyboard mashed, to trigger a loop over the full range $[1, 2^{32} - 1]$) results in a time of 2.601s. Not great...

Let's add some auto-vectorization (SSE 4.1, to match the judge's hardware):

```
#pragma GCC optimize ("O3")
#pragma GCC target ("sse4.1")
```

New time: 2.141s. That's better, but not good. What's going on? If we experiment locally by changing `sse4.1` to `avx2` it gives a time of 1.065s. This is exactly what we would expect if the code was auto-vectorized and the vector width doubled with AVX2. A hypothesis we can make from this is that the code *was* auto-vectorized into operating on 4 u32's at once, but that the loop is completely latency-bound, and the added latency of `pmulld` (`_mm_mullo_epi32`) over `imul` (normal multiplication) compensated for the increased throughput. (Agner's instruction tables seem to agree with this hypothesis – imul is listed as having 3 cycles of latency on Skylake, while pmulld has 10.) If that's the case, we should be able to make it throughput-bound and increase performance by keeping a bunch of simultaneous loop counters. Let's try it:

```
const int PAR = 8;
u32 subprod[PAR] = {1,1,1,1,1,1,1,1};
u32 cur2 = cur * 2 + 1;
for (; cur + PAR < ilim; cur += PAR, cur2 += PAR*2) {
  rep(i,0,PAR) subprod[i] *= cur2 + 2*i;
}
u32 prod = 1;
rep(i,0,PAR) prod *= subprod[i];
for (; cur < ilim; cur++) {
  prod *= cur * 2 + 1;
}
```

0.434s! That's better, and enough to get Accepted on the problem. With an `avx2` target pragma it goes down further to 0.225s. In fact, even with auto-vectorization disabled this trick helps performance, giving 0.853s.

For fun we can also try writing the loop manually using intrinsics:

```
#pragma GCC optimize ("unroll-loops")
#include <immintrin.h>
typedef __m128i M;
...

const int PAR = 8;
M madd = _mm_set1_epi32(PAR * 8);
M msubprod[PAR], mcur2[PAR];
rep(i,0,PAR) {
  msubprod[i] = _mm_set1_epi32(1);
  mcur2[i] = _mm_setr_epi32(
      2*cur + 8*i + 1, 2*cur + 8*i + 3,
      2*cur + 8*i + 5, 2*cur + 8*i + 7);
}
for (; cur + PAR * 4 < ilim; cur += PAR * 4) {
  rep(i,0,PAR) {
    msubprod[i] = _mm_mullo_epi32(msubprod[i], mcur2[i]);
    mcur2[i] = _mm_add_epi32(mcur2[i], madd);
  }
}
u32 prod = 1;
union {
  M mprod;
  u32 parts[4];
} u;
u.mprod = _mm_set1_epi32(1);
rep(i,0,PAR) u.mprod = _mm_mullo_epi32(u.mprod, msubprod[i]);
rep(i,0,4) prod *= u.parts[i];
for (; cur < ilim; cur++) {
  prod *= cur * 2 + 1;
}
```

This gives a runtime of 0.442s; roughly the same as auto-vectorization.

3 Modulo arbitrary numbers

Let's make life more difficult on ourselves.

Given N, K, M ($0 \leq K \leq N \leq 10^{18}$, $1 \leq M \leq 10^9$), compute

$$\binom{N}{K} = \frac{N!}{K!(N-K)!} \pmod{M}.$$

Time limit: 1 second.

Our approach to this will be similar to that of powers of two. We can start by factoring $M = \prod p^\alpha$, and solve the problem modulo each prime power, piecing the results together using the Chinese remainder theorem. For each prime power p^α, we decompose $N!$, $K!$ and $(N-K)!$ into powers of p and products of ranges $[1,k]$ with numbers divisible by p excluded. The product of a whole interval $[1,p^\alpha]$ is 1 if $p=2, \alpha \geq 3$, else -1,[1] so we can look only at ranges with upper bound less than p^α. Here's an initial attempt:

```
// Extended Euclidean algorithm and CRT from KACTL
ll euclid(ll a, ll b, ll &x, ll &y) {
  if (b) { ll d = euclid(b, a % b, y, x);
    return y -= a/b * x, d; }
  return x = 1, y = 0, a;
}

ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  ll x, y, g = euclid(m, n, x, y);
  assert((a - b) % g == 0); // else no solution
  x = (b - a) % n * x % n / g * m + a;
  return x < 0 ? x + m*n/g : x;
}

template<class F>
void factor(ll n, F f) {
  for (ll p = 2; p*p <= n; p++) {
    int a = 0;
    while (n % p == 0) n /= p, a++;
    if (a > 0) f(p, a);
  }
  if (n > 1) f(n, 1);
}

ll modpow(ll a, ll e, ll m) {
  if (e == 0) return 1;
  ll x = modpow(a * a % m, e >> 1, m);
  return e & 1 ? x * a % m : x;
```

[1] This isn't really necessary to the implementation, since we could just as well compute the product of a whole interval using a brute-force loop. But it's a fun excuse to do math.

If $\alpha = 1$, Wilson's theorem says that $(p-1)! = -1 \pmod{p}$, and has an easy proof: rearrange the product $1 \cdot 2 \cdot \ldots \cdot (p-1)$ to group x together with x^{-1}; this will get rid of all factors that are not their own inverse. Solving $x = x^{-1}$, or equivalently $x^2 = 1$, yields just two solutions $x = \pm 1$, and so $(p-1)! = 1 \cdot (-1) = -1 \pmod{p}$. An easy way of seeing that there aren't more solutions to $x^2 = 1$ is that a degree two polynomial can only have two roots.

For $p = 2$ we can compute the results for $\alpha = 2, 3$ by hand: $\alpha = 2$ gives a product $1 \cdot 3 = -1$, and $\alpha = 3$ gives $1 \cdot 3 \cdot 5 \cdot 7 = 1$. For larger α we can use the Wilson argument together with induction on the fact that $x^2 = 1$ has four solutions 1, -1, $2^{\alpha-1} - 1$, $2^{\alpha-1} + 1$.

For $p > 2$ we can solve $x^2 = 1 \pmod{p^\alpha}$ using Hensel lifting: a root x to the polynomial mod p lifts to exactly one corresponding root mod higher powers of p since the derivative $2x$ is non-zero for $p > 2$. Hence, ± 1 are the only roots to $x^2 = 1 \pmod{p^\alpha}$, and we can apply the Wilson argument.

For an easier argument we could also factor $x^2 - 1$ as $(x+1)(x-1)$ and look at what $x \pm 1$ being divisible by a power of p means for $x \mp 1$.

```
}

// N choose K modulo p^a
ll solve(ll N, ll K, int p, int a) {
  int mod = 1, acc = 0;
  rep(i,0,a) mod *= p;
  ll trailingZeroes = 0, res = 1, resdiv = 1;
  map<int, int> ivs;
  rep(i,0,3) {
    ll x = (i == 0 ? N : i == 1 ? K : N-K);
    int mult = (i == 0 ? 1 : -1);
    while (x > 0) {
      // Include the product (1 * 2 * ... * x) in the answer,
      // ’mult’ times, with numbers divisible by p excluded.
      ll lim = x + 1;
      ivs[(int)(lim % mod)] += mult;
      if (lim / mod % 2 == 1 && (mod == 4 || p > 2))
        res *= -1;
      acc += mult;
      x /= p;
      trailingZeroes += x * mult;
    }
  }

  int cur = 1;
  for (auto pa : ivs) {
    int lim = pa.first;
    ll prod = 1;
    for (; cur < lim; cur++) {
      if (cur % p != 0)
        prod = prod * cur % mod;
    }
    if (acc > 0) res = res * modpow(prod, acc, mod) % mod;
    else resdiv = resdiv * modpow(prod, -acc, mod) % mod;
    acc -= pa.second;
  }

  res = res * modpow(p, trailingZeroes, mod) % mod;
  res = res * modpow(resdiv, mod - mod/p - 1, mod) % mod;
  return res;
}

// N choose K modulo M
ll solve(ll N, ll K, ll M) {
  ll res = 0, prod = 1;
  factor(M, [&](ll p, int a) {
    ll r = solve(N, K, (int)p, a), m = 1;
    rep(i,0,a) m *= p;
    res = crt(res, prod, r, m);
    prod *= m;
  });
  return res;
}

void test() {
  vector<vector<int>> binom(100, vector<int>(100, -1));
  rep(m,1,100) rep(n,0,100) rep(k,0,n+1) {
    if (k == 0 || k == n) binom[n][k] = 1 % m;
    else binom[n][k] = (binom[n-1][k-1] + binom[n-1][k]) % m;
    assert(solve(n, k, m) == binom[n][k]);
  }
}
```

Runtime is an abysmal 20.839s on worst-case input $n = 999999935$, $m = 999999937$, but it works. Let us focus on the innermost loop:

```
for (; cur < lim; cur++) {
  if (cur % p != 0)
    prod = prod * cur % mod;
}
```

There are two things that are slow: the divisibility check by `p`, and the modular multiplication. The former would be easy to eliminate by turning the loop into two nested loops, with the innermost looping from 1 to $p-1$, however, in the interest of keeping the code short we will start by using another trick:

```
u32 invp2(u32 x) { // x^-1 mod 2^32
  u32 xinv = 1;
  rep(i,0,5) xinv = xinv * (2 - x * xinv);
  return xinv;
}
...

u32 pinv = invp2(p);
u32 plim = 0xFFFFFFFF / p;
...

if (p == 2) {
  u32 prod2 = ... product mod 2^32 as before
  prod = prod2 % mod;
} else {
  for (; cur < lim; cur++) {
    if ((u32)cur * pinv > plim)
      prod = prod * cur % mod;
  }
}
```

If `cur` is divisible by p, dividing will result in a number in the range $[0, (2^{32}-1)/p]$. We can compute this result by multiplying by the modular inverse of p $\pmod{2^{32}}$. On the other hand, if `cur` is not divisible by p, multiplying by the modular inverse of p cannot result in a number in that range, since that multiplication is invertible and already has a pre-image (given by multiplication by p). Hence, `cur % p == 0` can be replaced by `(u32)cur * pinv <= plim`, multiplication being a much cheaper operation than modulo.

We do need to be careful to treat $p = 2$ specially, since 2 is not invertible mod 2^{32}. However, this special-casing will be needed later on anyway, when we start looking into optimizing the modular multiplication. (By using 64-bit integers, it is possible to write a divisibility check similar to the above that works for both even and odd numbers, see [1].)

Performance-wise, this doesn't make much of a difference – we go from 20.8 seconds to 19.3. It seems likely that we are latency-bound. To avoid this latency bottleneck, let's do as before and use multiple accumulators:

```
const int PAR = 2;
ll subprod[PAR] = {1,1};
while (cur + PAR <= lim) {
  rep(i,0,PAR) {
    if ((u32)cur * pinv > plim)
      subprod[i] = subprod[i] * cur % mod;
    cur++;
  }
}
rep(i,0,PAR) prod = prod * subprod[i] % mod;
for (; cur < lim; cur++) {
  if ((u32)cur * pinv > plim)
    prod = prod * cur % mod;
}
```

This gives us a (still pretty awful) runtime of 11.848s. Rewriting the divisibility check as

```
subprod[i] * ((u32)cur * pinv > plim ? cur : 1) % mod;
```

apparently improves it further to 10.933s; removing it entirely gives 10.391s. Raising `PAR` further makes performance worse, and turning on -O3 makes no difference. Newer Intel processors are somewhat faster at division than the Haswell I'm benchmarking on, but even with a judge running on ultra-modern hardware we would be pretty far from the 1 second goal.

So... where do we go from here? We can't turn to SIMD, because there's no SIMD integer division or modulo instruction. Well, what we *can* do is roll our own modular multiplication.

3.1 Barrett reduction

One way of doing modular reduction quickly is by using Barrett reduction. The way it works is by writing $a\%b = a - \lfloor a/b \rfloor b$, and approximately computing the value of $\lfloor a/b \rfloor$ as $\lfloor (\lfloor (2^{64} - 1)/b \rfloor \cdot a)/2^{64} \rfloor$. We can then apply a final correction if the resulting approximate value of $a\%b$ is outside the range $[0, b)$. Since $b =$

`mod` will be constant over many multiplications, we can precompute the value of $\lfloor (2^{64} - 1)/b \rfloor$, turning the work needed for a single modular reduction into just two multiplications, a subtraction and a range correction.

```
struct Barrett {
  u64 b, m;
  Barrett(u64 b) : b(b), m(-1ULL / b) {}
  u64 reduce(u64 a) {
    u64 q = (u64)((u128(m) * a) >> 64), r = a - q * b;
    return r - b * (r >= b);
  }
};
```

Note that while we appear to use 128-bit arithmetic in the `reduce` function, which seems slow, the x86 instruction set actually has an instruction for $64 * 64 \to 128$ bit multiplication, making the computation of `q` into a single instruction. (Other instruction sets commonly also have this sort of operation – it's useful for implementing arbitrary precision integers.)

The given Barrett reduction implementation is valid for all 64-bit integers $a \geq 0, b > 0$. This follows from a few lines of algebra:

- $r \geq 0$: $a - \lfloor (\lfloor (2^{64} - 1)/b \rfloor \cdot a)/2^{64} \rfloor \cdot b \geq a - ((((2^{64} - 1)/b) \cdot a)/2^{64}) \cdot b = a/2^{64} \geq 0$.
- $r < 2b$: $a - \lfloor (\lfloor (2^{64} - 1)/b \rfloor \cdot a)/2^{64} \rfloor \cdot b < a - ((\lfloor (2^{64} - 1)/b \rfloor \cdot a)/2^{64} - 1) \cdot b \leq a - ((((2^{64} - 1)/b - (b-1)/b) \cdot a)/2^{64} - 1) \cdot b = b(1 + a/2^{64}) < 2b$.
- Finally, none of the computations overflow.

Let us rewrite the factorial loop using the above Barrett reduction:

```
#pragma GCC optimize("O3")
...

Barrett ba(mod);
...

const int PAR = 16;
u64 subprod[PAR];
rep(i,0,PAR) subprod[i] = 1;
while (cur + PAR <= lim) {
  rep(j,0,PAR) {
    u64 cur2 = (u32)cur * pinv > plim ? cur : 1;
    subprod[j] = ba.reduce(subprod[j] * cur2);
    cur++;
  }
}
rep(i,0,PAR) prod = ba.reduce(prod * subprod[i]);
for (; cur < lim; cur++) {
  u64 cur2 = (u32)cur * pinv > plim ? cur : 1;
  prod = ba.reduce(prod * cur2);
}
```

This runs in 2.122s, which quite a lot better than before! As a side note, Barrett reduction corresponds roughly to what compilers do to optimize `a % b` where `b` is a compile-time constant.

3.2 Relaxed Barrett reduction

The step in the Barrett reduction where we reduce the result into $[0, b)$ is nice for giving an answer in canonical form, but it is a bit unnecessarily slow. If we accept working with numbers in the range $[0, 2b)$, we can skip that part and do a reduction only at the end:

```
u64 reduce(u64 a) {
  return a - (u64)((u128(m) * a) >> 64) * b;
}
...

prod %= mod;
```

1.246s! Nice. In reference to the side note from before, this is actually *faster* than what the compiler would have produced with the naive version if `mod` was constant, which is a cool feat.

3.3 Floating-point modmul

Similarly to the idea behind Barrett reduction, we can compute a/b approximately using floating point numbers, and use that to get a good enough `a % b`, in the range $(-(1+\epsilon)b, (1+\epsilon)b)$. We combine multiplication and reduction in order to keep to 32-bit integers, making use of integer overflow:

```
int modmul(int a, int b, int m) {
  return (int)((u32)a * b - (u32)(int)(1.0 / m * a * b) * m);
}
```

There is still precomputation happening here, but it's a bit harder to spot: it's done automatically by the compiler which recognizes that `1.0 / m` is constant across loop iterations.

To get a sense of the range in which the modular multiplication works, let us without loss of generality assume $a, b \geq 0$ (negative numbers just negate the result), and look at an upper bound for

$$ab - \lfloor r(r(r(1/m) \cdot a) \cdot b) \rfloor \cdot m$$

where $r(x)$ denotes rounding x to the nearest double. $r(x)$ obeys $|r(x) - x| \leq x \cdot 2^{-53}$, so we get

$$\begin{aligned} ab - \lfloor r(r(r(1/m) \cdot a) \cdot b) \rfloor \cdot m &< ab - (r(r(r(1/m) \cdot a) \cdot b) - 1) \cdot m \\ &\leq ab - (ab/m \cdot (1 - 2^{-53})^3 - 1) \cdot m \\ &\leq m + 3ab \cdot 2^{-53}. \end{aligned}$$

Hence, as long as $3|ab| < 0.01cm \cdot 2^{53}$ the result will fall within $(-1.01m, 1.01m)$, and if $|a|, |b| \leq 1.01m$ and $m < 2^{31}/1.01$, that inequality certainly holds and nor will the output overflow. If not for using 32-bit integers as input parameters, we would be able to go higher, up to 2^{51} or so. For a more thorough analysis on the same subject, see [2].

Let's try using this for our program:

```
const int PAR = 16;
... parallel loop elided
for (; cur < lim; cur++) {
  int cur2 = (u32)cur * pinv > plim ? cur : 1;
  prod = modmul((int)prod, cur2, mod);
}
prod %= mod;
if (prod < 0) prod += mod;
```

This runs in 2.078s, which is worse than Barrett reduction. However, it's not completely useless – it works with larger numbers, and the avoidance of $64 * 64 \rightarrow 128$-bit multiplications makes it more amenable to SIMD.

3.4 Montgomery multiplication

In comparison to Barrett reduction and the floating-point based modular multiplication, Montgomery multiplication is more complicated. The idea is that we take the usual division algorithm that involves subtracting multiples of b from a until we reach the smallest possible non-negative value, and turn it around, instead *adding* multiples of b until we reach the (approximately) smallest possible value that's divisible by 2^{32}. The number of multiples to add can be computed using a single multiplication. We then right-shift by 32, producing a value congruent to $a/2^{32} \pmod{b}$ in the range $[0, 2b)$, which we can range-reduce if we feel like it. This operation that computes $a/2^{32} \pmod{b}$ is customarily called `redc`. We can compensate for the additional factor of $2^{-32} \pmod{b}$ by pre-multiplying all numbers by 2^{32},[2] or by multiplying by $(2^{32})^{\#\text{multiplications}}$ $\pmod{b}$ at the very end. In code:

```
struct Mont {
  u32 m, npr; // npr = -(m^-1) mod 2^32

  Mont(u32 mod) : m(mod), npr(-invp2(mod)) {}

  u32 redc(u64 a) {
    u32 b = (u32)a * npr;
    u64 c = a + (u64)b * m;
    return (u32)(c >> 32);
  }
```

[2] This is known as putting numbers in Montgomery form, and can be done using `redc(x * (2^64 % mod))`. The product of two numbers in Montgomery form is another number in Montgomery form, and we get numbers back in regular form using `redc(x)`.

```
};
```

To see that it works, observe that:

- $c = a + a \cdot (-m^{-1}) \cdot m = 0 \pmod{2^{32}}$
- $c = a \pmod{m}$
- $c/2^{32} \geq 0$
- $c/2^{32} < a/2^{32} + m$

Hence, as long as $a < m \cdot 2^{32}$, the result will end up in $[0, 2 \cdot m)$.

This method only works if the modulus is odd, since it depends on the existence of $m^{-1} \pmod{2^{32}}$. In our case, we have ensured this by special-casing $p = 2$.

Let us apply this to our program, as usual:

```
const int PAR = 4;
u32 subprod[PAR];
rep(i,0,PAR) subprod[i] = 1;
int reductions = lim - cur + PAR;
... parallel loop elided
rep(i,0,PAR) prod = mont.redc((u64)prod * subprod[i]);
for (; cur < lim; cur++) {
  u32 cur2 = (u32)cur * pinv > plim ? cur : 1;
  prod = mont.redc((u64)prod * cur2);
}
prod = prod * modpow(2, 32LL * reductions, mod) % mod;
```

This runs in 1.367s. That is again slightly worse than the Barrett reduction, but as with the floating-point modmul, the potential for SIMD is higher since there are no $64 * 64 \rightarrow 128$-bit multiplications.

3.5 64-bit Montgomery multiplication

By changing the Montgomery multiplication to use 64-bit integers (and 128-bit ones within redc), we can multiply two numbers at a time into the accumulator.

```
u64 invp2(u64 x) { ... }
u64 Mont::redc(u128 a) { ... }
...

u64 subprod[PAR];
while (cur + PAR * 2 <= lim) {
  rep(j,0,PAR) {
    u64 cur2 = (u32)cur * pinv > plim ? cur : 1;
    subprod[j] = subprod[j] * cur2;
    cur++;
  }
  rep(j,0,PAR) {
    u64 cur2 = (u32)cur * pinv > plim ? cur : 1;
    subprod[j] = mont.redc((u128)subprod[j] * cur2);
    cur++;
  }
}
```

Unfortunately the divisibility checks start to get expensive here for some reason, and this does not represent a speedup over the previous version. (Previously removing the divisibility checks made at most a ~5% performance runtime.) With divisibility checks removed (`cur2 = cur`) we get a runtime of 0.948s, and something like this should be achievable by special-casing different values of `p` and looping in different manners. I did not look into this, however.

3.6 Applying SIMD

We still have the potential to go further using SIMD. Let us consider the different modular multiplication algorithms we have and how we could vectorize them.

As alluded to, the Barrett reduction is hard to vectorize, because the x86 SIMD instruction set is missing an $64 * 64 \to 128$-bit multiplier. The relevant instructions that do exist are `_mm_mullo_epi32` ($32 * 32 \to 32$-bit multiplication), `_mm_mul_epi32` (signed $32 * 32 \to 64$-bit multiplication, using the bottom 32 bits of each 64-bit lane) and `_mm_mul_epu32` (unsigned $32 * 32 \to 64$-bit multiplication). We could potentially create a $64 * 64 \to 128$-bit multiplier out of these, but it would be hard without add-with-carry instructions, and it wouldn't be fast.

On the other hand, both the float-based modular multiplication and the Montgomery multiplication are in principle easy to vectorize. The main annoyance comes from the fact that, as with the 64-bit Montgomery, the divisibility check by p starts to become a bottleneck. A fix for that which is relatively unintrusive (but unfortunately did not transfer well to the 64-bit Montgomery) is to keep a variable indicating the next number divisible by p that we should skip over. We can then check in the inner loop whether we cross such a boundary, and if so correct for it.

Here is a SIMD version of the float-based modular multiplication, which uses that trick:

```
int cur = 0, nextp = 0; // nextp = p * ceil(cur / p)
...

const int PAR = 8;
typedef __m128i mi;
typedef __m256d md;
mi msubprod[PAR];
mi mcur = _mm_setr_epi32(cur, cur + 1, cur + 2, cur + 3);
rep(i,0,PAR) msubprod[i] = _mm_set1_epi32(1);
mi ms = _mm_set1_epi32(mod);
md minv = _mm256_set1_pd(1.0 / mod);
```

```
while (nextp < cur) nextp += p;
mi mnextp = _mm_set1_epi32(nextp - 1);
mi mp = _mm_set1_epi32(p);

int maxStep = (p == 3 ? 6 : 5) * PAR;
while (cur + maxStep <= lim) {
  rep(j,0,PAR) {
    mi mprod = msubprod[j];
    mi mnext = _mm_add_epi32(mcur, _mm_set1_epi32(4));
    cur += 4;

    // Skip over numbers divisible by p. For p = 3 we skip two,
    // for p = 5 one, and for larger p either zero or one.
    // This will be a well-predicted branch, and if the
    // statements within happen to run we trade some cycles for
    // a longer skip in 'cur'.
    if (nextp <= cur) {
      mcur = _mm_sub_epi32(mcur,
                           _mm_cmpgt_epi32(mcur, mnextp));
      mnext = _mm_add_epi32(mnext, _mm_set1_epi32(1));
      mnextp = _mm_add_epi32(mnextp, mp);
      nextp += p;
      cur++;
      if (p == 3) {
        mcur = _mm_sub_epi32(mcur,
                             _mm_cmpgt_epi32(mcur, mnextp));
        mnext = _mm_add_epi32(mnext, _mm_set1_epi32(1));
        mnextp = _mm_add_epi32(mnextp, mp);
        nextp += p;
        cur++;
      }
    }

    mi mab = _mm_mullo_epi32(mprod, mcur);
    mi mfl = _mm256_cvttpd_epi32(
      _mm256_mul_pd(
        _mm256_mul_pd(minv, _mm256_cvtepi32_pd(mcur)),
        _mm256_cvtepi32_pd(mprod)
      )
    );
    msubprod[j] = _mm_sub_epi32(mab, _mm_mullo_epi32(ms, mfl));
    mcur = mnext;
  }
}
rep(i,0,PAR) {
  union { int i[4]; mi m; } u;
  u.m = msubprod[i];
  rep(j,0,4) prod = prod * u.i[j] % mod;
}
```

```
for (; cur < lim; cur++) {
  int cur2 = (u32)cur * pinv > plim ? cur : 1;
  prod = modmul((int)prod, cur2, mod);
}
prod %= mod;
if (prod < 0) prod += mod;
```

It runs in 0.728s on input that has p as a large prime, and could go down to 0.656s if we managed to get rid of the number-skipping machinery entirely. Setting mod to powers of 3, 5, 7, 11 we observe a very similar runtime, indicating that the reduced number of multiplications seems to make up for the complex code within the condition.

The code uses AVX instructions, but no AVX2 ones. For comparison, the scalar float-based modmul ran in 2.078s, and the Barrett reduction in 1.246s.

Here is a SIMD version of the Montgomery multiplication, along the same implementation lines but using AVX2:

```
int cur = 0, nextp = 0;
...

const int PAR = 16;
typedef __m256i mi;
mi msubprod[PAR];
rep(i,0,PAR) msubprod[i] = _mm256_set1_epi64x(1);
mi mcur = _mm256_setr_epi64x(cur, cur + 1, cur + 2, cur + 3);
mi mnpr = _mm256_set1_epi64x(mont.npr);
mi mm = _mm256_set1_epi64x(mod);
int reductions = lim - cur + 4 * PAR;

while (nextp < cur) nextp += p;
mi mnextp = _mm256_set1_epi64x(nextp - 1);
mi mp = _mm256_set1_epi64x(p);

int maxStep = (p == 3 ? 6 : 5) * PAR;
while (cur + maxStep <= lim) {
  rep(j,0,PAR) {
    mi mnext = _mm256_add_epi64(mcur, _mm256_set1_epi64x(4));
    cur += 4;

    if (nextp <= cur) {
      reductions--;
      ... as before, except with _mm256_..._epi64
    }

    mi ma = _mm256_mul_epu32(msubprod[j], mcur);
    mi mb = _mm256_mul_epu32(ma, mnpr);
    mi mc = _mm256_add_epi64(ma, _mm256_mul_epu32(mb, mm));
    msubprod[j] = _mm256_srli_epi64(mc, 32);
    mcur = mnext;
  }
}

rep(i,0,PAR) {
  union { u64 i[4]; mi m; } u;
  u.m = msubprod[i];
  rep(j,0,4)
    prod = mont.redc((u64) prod * u.i[j]);
}
...
```

This runs in 0.455s, or 0.423s without the number-skipping. The scalar version ran in 1.367s, so this corresponds to a 3x speed-up, which is pretty good!

3.7 Summary

The following table summarizes the different versions:

Algorithm	Time	Ideal time
Naive loop	20.839s	18.983s
Optimized divisibility test	19.288s	18.983s
Parallel loop	10.933s	10.391s
Barrett reduction	2.122s	1.612s
Relaxed Barrett reduction	1.246s	1.195s
Float-based	2.078s	1.983s
Montgomery	1.367s	1.184s
Montgomery, 64-bit	-	0.948s
Float-based, SIMD	0.728s	0.656s
Montgomery, SIMD	0.455s	0.423s

Ideal time represents time with divisibility check removed.

In conclusion, relaxed Barrett reduction is a good default modular reduction algorithm. If higher performance is required, one can use vectorized versions to win a factor 2-3; either with Montgomery multiplication, which is fast but requires AVX2 and has a complex API, or with float-based multiplication.

References

[1] D. Lemire, O. Kaser, and N. Kurz, "Faster remainder by direct computation: Applications to compilers and software libraries," *Software: Practice and Experience*, vol. 49, no. 6, pp. 953–970, 2019.

[2] S. Lindholm, "Correctness of KACTL's modmul." https://github.com/kth-competitive-programming/kactl/blob/master/doc/modmul-proof.tex, 2020.