An algorithm for computing modular nested exponentiation efficiently

Aviv Brook

Mathematics Department, UC Santa Cruz

Abstract

We present an algorithm that takes as input an arbitrarily long sequence of positive integers $a_1, a_2, \ldots, a_\ell$ and a positive integer m and computes

$$a_1^{a_2^{\cdot^{\cdot^{a_\ell}}}} \bmod m$$

efficiently (that is, without computing the value of the nested exponent).

Keywords: modular exponentiation, nested exponentiation

1. Introduction

1.1. Notation

For convenience, we define an operator E as a shorthand for nested exponentiation.

Definition 1 (nested exponentiation operator). Given a sequence of ℓ positive integers $a_1, a_2, \ldots, a_\ell$, define the operator E recursively as follows.

$$E(a_1, a_2, \ldots, a_\ell) = \begin{cases} 1 & \text{if } \ell = 0, \\ a_1^{E(a_2,\ldots,a_\ell)} & \text{if } \ell > 0. \end{cases}$$

We call $b = a_1$ the **base** and $e = E(a_2, \ldots, a_\ell)$ the **exponent** of $E(a_1, a_2, \ldots, a_\ell)$.

Note that sometimes we write $a \equiv b \pmod{m}$ and sometimes we write $a \bmod m = b$. The two expressions are related yet different.

When we write $a \equiv b \pmod{m}$ (*with* parentheses), we refer to the congruence relation modulo m. This is what a mathematician would think of when they hear the term "modulo".

When we write $a \bmod m = b$ (*without* parentheses), we refer to the modulo operation. That is, b is the remainder of the Euclidean division of a by m. This is what a computer scientist would think of when they hear the term "modulo". Note that we also write $a \operatorname{div} m = q$ to denote the associated quotient.

Email address: avbrook@ucsc.edu (Aviv Brook)

1.2. A simple base case

Suppose that we are given two positive integers b, e and would like to compute $E(b, e) \bmod m$ for some positive integer m. We refer to this as *simple modular exponentiation*. This is well-studied as it is a common operation in public-key cryptosystems, such as RSA. For this reason, many programming languages have built-in functions to compute $b^e \bmod m$ efficiently, such as Python's `pow`. Most such procedures generally use a principle called *exponentiation by squaring* and can perform this computation with $O(\log e)$ multiplications. Suppose we have a procedure MOD-EXP that takes as input positive integers b, e, m and returns $b^e \bmod m$.

1.3. Taking it one step further

The primary goal of this paper is to take this concept one step further – we extend the discussion to arbitrarily long sequences of positive integers, representing what we call a nested exponential, modulo m. We are interested in computing $E(a_1, a_2, \ldots, a_\ell) \bmod m$ given a sequence of positive integers $a_1, a_2, \ldots, a_\ell$ and a modulus $m > 0$.

When dealing with numbers defined as such, increasing the value of any a_i or the length ℓ of the sequence by small amounts greatly influences the magnitude of $E(a_1, a_2, \ldots, a_\ell)$. It is therefore not possible for us to simply compute the value of the exponent $E(a_2, \ldots, a_\ell)$ then call MOD-EXP(b, e, m) in the general case.

Even computations on sequences of five single-digit numbers cannot be computed with this method. For example, $E(2, 3, 4, 5)$ is a 10^{476} TB number so any computer would run out of memory before finishing the computation of $E(b, 2, 3, 4, 5) \bmod m$ for any b and m. Suppose now that we want to perform this computation on a sequence of ten 20-digit integers modulo some m. Computing the exponent then using simple modular exponentiation would not work even on a modern-day supercomputer due to both time and memory constraints.

We therefore introduce a novel approach to perform such computations efficiently. Given positive integers $a_1, a_2, \ldots, a_\ell$, let $b = a_1$ be the base and $e = E(a_2, \ldots, a_\ell)$ be the exponent of $E(a_1, a_2, \ldots, a_\ell)$. We define a recursive function that computes $b^e \bmod m$ by computing two carefully defined subproblems. The recursion terminates when the sequence has length $\ell = 2$, in which case we invoke MOD-EXP. At every step in the recursion, the algorithm has three key parts.

(1) **Split the modulus into coprime factors:** find positive integers n and h such that $m = nh$ and n is coprime to both h and b.
(2) **Compute two subproblems:** evaluate $r_1 = b^e \bmod n$ and $r_2 = b^e \bmod h$ separately. The former involves recursively computing $e \bmod \varphi(n)$, where φ is Euler's totient function.
(3) **Combine the results of the subproblems:** use the computed values for r_1 and r_2 to compute $b^e \bmod m$. This involves invoking a special case of the Chinese remainder theorem.

In this paper, we discuss each of these three steps separately before combining them to yield the full algorithm. We implemented the algorithm in Python and organised our work into a Jupyter notebook which can be found here. It can also be installed as a Python package.

1.4. The context of this paper

Before we continue, we would like to place this work in context. There is already a large body of research on computing modular addition (iterated succession), multiplication (iterated addition), and exponentiation (iterated multiplication). The natural next step would be considering modular *tetration* (iterated exponentiation). We could not find any papers explicitly treating the computation of modular tetration, though some papers do discuss the subject [1]. Our algorithm can compute modular tetration as a special case but can, in fact, solve more generalised instances as well. This is because we do not assume that $a_1 = a_2 = \cdots = a_\ell$.

2. Preliminaries

We use a number of functions as subroutines in our main algorithm. We implemented these functions in our Python package but briefly mention their functionality here.

(1) Given positive integers b, e, m, MOD-EXP(b, e, m) returns $b^e \bmod m$ (computed using exponentiation by squaring).
(2) Given integers a and b, GCD(a, b) returns $\gcd(a, b)$ (computed using Euclid's algorithm).
(3) Given nonnegative integers a and b, EXT-GCD(a, b) returns integers g, x, y where $g = \gcd(a, b)$ and $ax+by = g$ (computed using the extended Euclidean algorithm).
(4) Given a positive integer n, TOTIENT(n) returns $\varphi(n)$, where φ is Euler's totient function.

Like MOD-EXP, GCD and EXT-GCD are also well-known and run quickly in time $O(\log(\max(a, b)))$. TOTIENT(n) is our algorithm's main bottleneck as it depends on the time it takes to factor n into primes. As far as we know or suspect, there is no efficient non-quantum algorithm for general integer factorisation. For this reason, our algorithm runs fast when m is small (on the order of 10-20 digits), even when the a_is and ℓ are large.

3. The algorithm

Given a sequence of positive integers $a_1, a_2, \ldots, a_\ell$ and a positive integer m, we want to compute

$$a_1^{a_2^{\cdot^{\cdot^{a_\ell}}}} \bmod m = E(a_1, a_2, \ldots, a_\ell) \bmod m.$$

Let $b = a_1$ be the base and $e = E(a_2, \ldots, a_\ell)$ be the exponent of $E(a_1, a_2, \ldots, a_\ell)$. We are interested in computing $b^e \bmod m$.

3.1. Split the modulus into coprime factors

The first step of our algorithm involves finding two positive integers n and h such that

(a) $m = nh$,
(b) n and h are coprime, and
(c) b and n are coprime.

The reader may notice that we factor m into coprime integers n and h in this first step of the algorithm. However, our procedure does not require general integer factorisation – only a few steps of the Euclidean algorithm. Note that the complexity of integer factorisation *does* play a role in a later step, where we compute a totient.

More specifically, we define three sequences recursively. Define $n_0 = m$, $h_0 = 1$, and $g_0 = \gcd(b, m)$; for all $k \geq 0$, define

$$n_{k+1} = n_k/g_k, \qquad h_{k+1} = h_k g_k, \qquad g_{k+1} = \gcd(g_0, n_{k+1}).$$

In the resulting sequences, note that $n_k h_k = m$ for all $k \geq 0$. The following algorithm computes these sequences, stopping when $g_k = 1$ (i.e., when $\gcd(g_0, n_k) = 1$).

Theorem 1. For all $k > 0$, the above sequence satisfies $m = n_k h_k$ and $h_k \mid g_0^k$. Moreover, there exists a positive integer $K \leq \log_2(m)$ such that the sequences remain constant for $k \geq K$, and $\gcd(b, n_K) = \gcd(n_K, h_K) = 1$.

Proof. The equality $m = n_k h_k$ is nearly obvious, since it is true for $k = 0$ and the product $n_k h_k$ does not change from step to step.

To show that $h_k \mid g_0^k$, observe that for any step $k > 0$, $g_k = \gcd(g_0, n_k)$ and is therefore a factor of g_0. Since h_k is the product of k such factors of g_0, we must have that $h_k \mid g_0^k$.

To prove that the sequences stabilise, let $k > 0$ be any step during which $g_k > 1$. Thus, $n_k/n_{k+1} = g_k \geq 2$ and so $n_{k+1} \leq n_k/2$. Hence, the sequence n_k decreases by a factor of at least 2 at each step until $g_K = n_K = 1$ and $h_K = m$ for some K. We therefore arrive at $g_K = 1$ and the sequences stabilise after no more than $\log_2(n_0) = \log_2(m)$ steps.

To prove that $\gcd(n_K, h_K) = 1$, observe that $g_K = \gcd(g_0, n_K) = 1$. This means that g_0 and n_K share no common factor > 1. We must therefore also have that $\gcd(n_K, g_0^K) = 1$. Since we proved that h_K is a factor of g_0^K, we must have that $\gcd(n_K, h_K) = 1$.

Finally, to prove that $\gcd(b, n_K) = 1$, observe that

$$\begin{aligned}
\gcd(b, n_K) &= \gcd(b, \gcd(m, n_K)) && \text{(since } n_K \mid m\text{)}, \\
&= \gcd(\gcd(b, m), n_K) && \text{(by associativity of gcd)}, \\
&= \gcd(g_0, n_K) && \text{(since } g_0 = \gcd(b, m)\text{)}, \\
&= 1 && \text{(since } g_K = \gcd(g_0, n_K) = 1\text{)}.
\end{aligned}$$

□

Algorithm 1 Split the modulus into coprime factors.

Input: positive integers b and m.

Output: positive integers g, K, n, h such that

- $g = \gcd(b, m)$,
- K is the total number of steps and $h \mid g^K$,
- $m = nh$, and
- $\gcd(b, n) = \gcd(g, n) = \gcd(n, h) = 1$.

```
1: function SPLIT-MOD(b, m)
2:     g_0 := GCD(b, m)
3:     n_1 := m/g_0,  h_1 := g_0
4:     k := 1
5:     while g_k := GCD(g_0, n_k) > 1 do
6:         n_{k+1} := n_k/g_k,  h_{k+1} := h_k g_k
7:         k := k + 1
8:     return g_0, k, n_k, h_k                ▷ (g, K, n, h) = (g_0, k, n_k, h_k)
```

Corollary 2. Algorithm 1 halts, computing SPLIT-MOD(b, m) in $O(\log m)$ steps.

Proof. This follows from Theorem 1. □

3.2. Compute two subproblems

Algorithm 1 computes four positive integers g, K, n, h satisfying $g = \gcd(b, m)$, $h \mid g^K$, $m = nh$, and $\gcd(b, n) = \gcd(g, n) = \gcd(n, h) = 1$. Recalling that $b = a_1$ and $e = E(a_2, \ldots, a_\ell)$, we now want to separately compute

$$r_1 = b^e \bmod n \qquad \text{and} \qquad r_2 = b^e \bmod h.$$

Since b and n are coprime, Euler's totient theorem guarantees that

$$b^e \bmod n = b^{e \bmod \varphi(n)} \bmod n.$$

We may therefore compute $\varphi(n)$ using TOTIENT then compute $e_1 = e \bmod \varphi(n) = E(a_2, \ldots, a_\ell) \bmod \varphi(n)$ by making a recursive call to our main algorithm. We then compute r_1 by calling MOD-EXP(b, e_1, n).

To compute r_2, first let $b_1 = b/g$. We want to compute

$$r_2 = b^e \bmod h = (b_1 g)^e \bmod h = (b_1^e g^e) \bmod h.$$

Consider two cases: (1) $e \geq K$, and (2) $e < K$.

Case 1 ($e \geq K$). In the case that $e \geq K$, observe that since $h \mid g^K$, we have that $g^K \bmod h = 0$. Thus,

$$r_2 = (b_1^e g^e) \bmod h = (b_1^e g^K g^{e-K}) \bmod h = 0.$$

Case 2 ($e < K$). In the case that $e < K$, we know that e is small since $K = O(\log m)$. We are therefore able to directly evaluate e and then compute r_2 by simply calling MOD-EXP(b, e, h).

Case 1 is more common since our algorithm is most useful when e is large. Otherwise, there would be little reason to use our algorithm over MOD-EXP. But even if not, we can still compute r_2 efficiently in Case 2.

To implement this step, we need an efficient method to check whether $e < K$ without having to compute e. We can use a simple recursive function that takes the logarithm at every recursive step. At every step, our function can compare the current value of K with the current base e_1 of e. We may be able to optimise this function by terminating the recursion earlier when e is clearly growing beyond K. An efficient way to do this would be checking if the bit length of e is bounded by the bit length of K before computing the logarithm. We denote the bit length of a positive integer a by $B(a) = \lfloor \log_2(a) \rfloor + 1$.

Theorem 3. Let K be a positive number and $e_1, \ldots, e_\ell$ be a sequence of $\ell \geq 2$ positive integers. Then $e_2 \cdot (B(e_1) - 1) \geq B(\lceil K \rceil)$ implies $E(e_1, e_2, \ldots, e_\ell) \geq K$.

Proof. Let $e = E(e_1, e_2, \ldots, e_\ell)$ and observe that

$$\begin{aligned}
\log_2(K) &\leq \log_2(\lceil K \rceil) && \text{(since } K \leq \lceil K \rceil\text{)}, \\
&< \lfloor \log_2(\lceil K \rceil) \rfloor + 1 = B(\lceil K \rceil) \\
&\leq e_2 \cdot (B(e_1) - 1) \\
&\leq E(e_2, \ldots, e_\ell) \cdot (B(e_1) - 1) && \text{(since } e_2 \leq E(e_2, \ldots, e_\ell)\text{)}, \\
&= E(e_2, \ldots, e_\ell) \cdot \lfloor \log_2(e_1) \rfloor \\
&\leq E(e_2, \ldots, e_\ell) \cdot \log_2(e_1) \\
&= \log_2(E(e_1, e_2, \ldots, e_\ell)) = \log_2(e).
\end{aligned}$$

It follows that $K \leq e$. □

Algorithm 2 Check if $e < K$.

Input: a sequence of positive integers $e_1, e_2, \ldots, e_\ell$ and a positive number K.
Output: `True` if $E(e_1, e_2, \ldots, e_\ell) < K$, `False` otherwise.

```
1:  function POW-LT((e_1, e_2, ..., e_ℓ), K)
2:      if ℓ = 0 then                          ▷ we defined E(e_1, ..., e_ℓ) = 1 when ℓ = 0
3:          return True if 1 < K, False otherwise
4:      else if ℓ = 1 or e_1 = 1 then                                  ▷ if e_1 = 1, e = 1
5:          return True if e_1 < K, False otherwise
6:      if e_2 · (B(e_1) − 1) ≥ B(⌈K⌉) then
7:          return False
8:      if t := log_{e_1} K > 1 then
9:          return POW-LT((e_2, ..., e_ℓ), t)
10:     return False
```

If POW-LT returns `True`, we need to evaluate e. Define a function POW-LIST that takes as input a sequence of positive integers $e_1, \ldots, e_\ell$ and recursively computes $E(e_1, \ldots, e_\ell)$ using binary exponentiation. We implemented this function in our Jupyter notebook but omit it here as it is fairly trivial.

3.3. Combine the results of the subproblems

We now have two remainders r_1 and r_2, where

$$r_1 = b^e \bmod n \qquad \text{and} \qquad r_2 = b^e \bmod h.$$

We now want to use these values to compute $b^e \bmod m$. Since n and h are coprime, we can invoke a special case of the Chinese remainder theorem.

Theorem 4. Let n and h be coprime positive integers and let x and y be Bézout coefficients for n and h: integers that satisfy the linear Diophantine equation

$$nx + hy = 1.$$

Then, if a is a positive integer,

$$a \bmod m = [h(a \bmod n)(y \bmod n) + n(a \bmod h)(x \bmod h)] \bmod m.$$

This can be proved using simple algebra. We therefore compute the Bézout coefficients x and y of n and h by calling EXT-GCD(n, h). This computation takes time $O(\log(\max(n, h))) = O(\log m)$. We then invoke Theorem 4 to compute

$$b^e \bmod m = [h(b^e \bmod n)(y \bmod n) + n(b^e \bmod h)(x \bmod h)] \bmod m.$$

3.4. Tying it all together

We combine the three parts to yield our full algorithm MOD-NEST-EXP. But, first, we make slight modifications:

(a) We rewrite Algorithm 1 to reuse variables whenever possible. That is, we only need one variable g to hold the value of g_0 and two variables g_{new} and n to hold the current values of g_k and n_k respectively at each step $k \geq 1$. We observe that h_k is never used within the while loop so we may simply compute h after the loop terminates. This replaces K multiplications with one division.

(b) Right after computing $g = \gcd(b, m)$, we check if $g = 1$. If so, Euler's theorem guarantees that $b^e \bmod m = b^{e \bmod \varphi(m)} \bmod m$. We may therefore compute TOTIENT(m), immediately make a recursive call to MOD-NEST-EXP to compute $e_1 = e \bmod \varphi(m)$, and then return MOD-EXP(b, e_1, m) without having to follow the three standard steps we described for our algorithm.

Algorithm 3 Modular nested exponentiation.

Input: a sequence of positive integers $a_1, a_2, \ldots, a_\ell$ and a positive integer m.
Output: $E(a_1, a_2, \ldots, a_\ell) \bmod m$.

function MOD-NEST-EXP($(a_1, a_2, \ldots, a_\ell), m$)
  if $m = 1$ **then** ▷ 1 divides every integer
    return 0
  if $\ell = 0$ **then** ▷ we defined $E(e_1, \ldots, e_\ell) = 1$ when $\ell = 0$
    return $1 \bmod m$
  else if $\ell = 1$ **then**
    return $a_1 \bmod m$
  else if $\ell = 2$ **then** ▷ this is the recursive base case
    return MOD-EXP(a_1, a_2, m)

  $b := a_1$, $e := (a_2, \ldots, a_\ell)$ ▷ let b be the base and e be the exponent
  $g := \text{GCD}(b, m)$
  if $g = 1$ **then** ▷ if $\gcd(b, m) = 1$, immediately invoke Euler's theorem
    return MOD-EXP$(b, \text{MOD-NEST-EXP}(e, \text{TOTIENT}(m)), m)$

  Part 1 (split m into coprime factors):
  $n := m/g$, $k := 1$
  $g_{\text{new}} := \text{GCD}(g, n)$
  while $g_{\text{new}} > 1$ **do**
    $n := n/g_{\text{new}}$, $k := k + 1$
    $g_{\text{new}} := \text{GCD}(g, n)$
  $h := m/n$ ▷ computing h at the end is more efficient

  Part 2 (compute r_1 and r_2):
  $r_1 := \text{MOD-EXP}(b, \text{MOD-NEST-EXP}(e, \text{TOTIENT}(n)), n)$
    ▷ $r_1 = b^e \bmod n$ is computed recursively
  $r_2 := 0$ ▷ $r_2 = b^e \bmod h = 0$ if $e \geq k$
  if POW-LT(e, k) **then** ▷ if $e < k$, directly evaluate r_2
    $r_2 := \text{MOD-EXP}(b, \text{POW-LIST}(e), h)$

  Part 3 (combine r_1 and r_2 using the Chinese remainder theorem):
  $q, x, y := \text{EXT-GCD}(n, h)$ ▷ we already know that $q = 1$
  return $[hr_1(y \bmod n) + nr_2(x \bmod h)] \bmod m$

4. Empirical results

We implemented the algorithm in Python. We used `gmpy2`'s `powmod`, `gcd`, and `gcdext` to compute MOD-EXP, GCD, and EXT-GCD respectively and we used `sympy`'s `totient` function to compute TOTIENT. We implemented functions `pow_lt`, `pow_list`, and `mod_nest_exp` to compute POW-LT, POW-LIST,

and MOD-NEST-EXP respectively.

We benchmarked the runtime of `mod_nest_exp` by running the function on a set of randomly generated inputs and timing each execution. We generated inputs by fixing three positive integers B (the bit-length of the modulus m), b (the bit-length of each integer a_i in the sequence), and ℓ (the length of the input sequence). We generated ℓ uniformly psuedorandom b-bit integers (from the set $\{2^{b-1}, \ldots, 2^b - 1\}$) and one random B-bit integer using Python's `random.randint`. We then ran `mod_nest_exp` on these inputs and recorded the runtime using Python's `time.time`. For every fixed B, b, ℓ, we generated 1000 inputs, timed each of them, and took the mean and standard deviation of the runtimes, as summarised in Table 1. All runs were executed on the attached notebook using the Google Colab hosted runtime.

Table 1: Mean runtimes and standard deviations (in ms) from 1000 `mod_nest_exp` runs on sequences of ℓ pseudorandom b-bit positive integers over a B-bit modulus.

		sequence length ℓ					
		10		100		1000	
B	b	mean	stdev	mean	stdev	mean	stdev
16	16	0.16	0.10	0.17	0.09	0.25	1.82
	128	0.16	0.07	0.17	0.08	0.20	0.09
	1024	0.16	0.06	0.17	0.08	0.23	0.09
32	16	0.63	0.54	0.65	0.47	0.72	0.48
	128	0.74	1.95	0.67	0.49	0.71	0.49
	1024	0.66	0.47	0.67	0.45	0.76	0.53
64	16	14.53	55.50	12.41	45.63	12.92	39.91
	128	15.07	59.37	13.93	55.71	17.68	66.83
	1024	11.95	40.19	12.75	39.65	14.61	50.07

As we claimed earlier, the bit-length of m has the biggest influence on the runtime of the algorithm. Increasing either ℓ or b while keeping the other variables constant has no discernable effect on the function's runtime. This is because `totient` is orders of magnitude slower than any of the other subroutines.

Acknowledgement

We are incredibly thankful for Martin H. Weissman for advising this paper and for his help with optimising the algorithm.

References

[1] J. Brennan, B. Geist, Analysis of iterated modular exponentiation: The orbits of $x\alpha \bmod n$, Designs, Codes and Cryptography 13 (3) (1998) 229–245. doi:10.1023/A:1008289605486.