Number Theory

Assignment 2 — MTH3251

Fast Modular Exponentiation

Modular exponentiation is an important block in the RSA encryption algorithm which is done at both the sender and reciever sides. Therefore, the modular exponentiation algorithm must be fast and efficient for large integer numbers to make the RSA algorithm practical.

1. **Right-to-Left Binary Modular Exponentiation**: This algorithm is called right-to-left because it scans the exponent e bit-wise from right to left. S is initialized to b and $S \times S$ is calculated in each iteration, so it is also called *repeated squaring*.

Algorithm 1 Right-to-Left Binary Modular Exponentiation

Require: Non-negative Integers b, e, n, where $e = (e_{k-1}e_{k-2}\dots e_1e_0)_2$

Ensure: $y = b^e \pmod n$

```
A ← 1
S ← b
for i ∈ {0, ..., k − 1} do
    if e_i = 1 then
        A ← (A × S) %n
    end if
    S ← (S × S) %n
end for
return A
```

2. **Left-to-Right Binary Modular Exponentiation**: This algorithm first appeared 200 BCE in India but is known as "Left-to-right binary exponentiation" or a *square and multiply* algorithm. The Right-to-Left Algorithm described earlier computes $A.S$ if e_i is 1. The computation $A.b$ can sometimes be more effective for a fixed b instead of an arbitrary large S, especially if the base b is a small integer. This algorithm requires one temporary register less than the Right-to-Left version because the variable A is discarded.

   Rationale of left-to-right traversal: consider how you "build" a number from its binary representation when seen from MSB to LSB. You begin with 1 for the MSB. For each new bit you see you double the result, and if the bit is 1, you add 1.

   For example consider the binary 1101. Begin with 1 for the leftmost 1. We have another bit, so we double. That's 2. Now, the new bit is 1, so we add 1, that's 3. We have another bit, so again double, that's 6. The new bit is 0, so nothing is added. And we have one more bit, so once again double, getting 12, and finally adding 1, getting 13. Indeed, 1101 is the binary representation of 13.

Omar Elgendy: oelgendy@ieee.org

Algorithm 2 Left-to-Right Binary Modular Exponentiation

Require: Non-negative Integers b, e, n, where $e = (e_{k-1}e_{k-2}\dots e_1e_0)_2$

Ensure: $y = b^e \pmod n$

```
S ← b
for i ∈ {k − 2, . . . , 1, 0} do
    S ← (S × S) %n
    if e_i = 1 then
        S ← (S × b) %n
    end if
end for
return S
```

3. **CRT**: Computation of $b^e \pmod n$ can be accelerated by using the Chinese Remainder Theorem (CRT). Since the sender and receiver know the prime factors p, q of the modulus $n = pq$, then we can first do the modular exponentiations in the smaller moduli p, q:

$$y_p = b^e \pmod p, \quad y_q = b^e \pmod q \tag{1}$$

then, we combine the 2 results using CRT to get the final result in modulo n:

$$y = b^e \pmod n \equiv y_p \times q \times x_q + y_q \times p \times x_p \pmod n \tag{2}$$

where x_q is the multiplicative inverse of q in modulo p and x_p is the multiplicative inverse of p in modulo q. They can be obtained by solving the following linear congruences using extended GCD algorithm.

$$qx_q \equiv 1 \pmod p, \quad px_p \equiv 1 \pmod q \tag{3}$$

For computing the modular exponentiations in (1), please use Left-to-Right algorithm.

Implement the three previous algorithms in Python or C and test the runtime of each one on this example and confirm that the results are correct:

$$b = 11, e = 12354, p = 7919, q = 5153, n = pq = 40806607$$

Compare between the runtime of the three algorithms and state which is the fastest one in your implementation. Note: for a fair comparison, the CRT algorithm should have a parallel implementation, but this is out of scope of this course.

Note: In python, please JIT compile the functions first using `numba` python module

Note: In C++, the integer division process does not give the quotient, so do not use it in the extended GCD algorithm. Instead, you can use `std::lldiv(a, b)`. Also, make sure the variables used for extended GCD are signed because the algorithm has negative values. For other algorithms, the variable could be either signed or unsigned.

Omar Elgendy: oelgendy@ieee.org