RSA Implementation

Çetin Kaya Koç

Oregon State University

Contents:

- Exponentiation heuristics
- Multiplication algorithms
- Computation of GCD and Inverse
- Chinese remainder algorithm
- Primality testing

RSA Encryption & Decryption

$$C := M^e \bmod n$$

$$M := C^d \bmod n$$

where

n : Modulus ($\log_2 n = k \geq 512$)

e : Encryption exponent

d : Decryption exponent

M : Plaintext ($0 \leq M \leq n-1$)

C : Cryptotext ($0 \leq C \leq n-1$)

Modular Exponentiation

We do **NOT** compute

$$C := M^e \pmod{n}$$

by first computing

$$M^e$$

and then computing the remainder

$$C := (M^e) \mod n$$

Temporary results must be reduced modulo n at each step of the exponentiation.

Exponentiation

$$M^{15}$$

- How many multiplications are needed ?

Naive Answer:

$$M \to M^2 \to M^3 \to M^4 \to M^5 \to \ldots \to M^{15}$$

Requires 14 multiplications.

Binary Method:

$$M \to M^2 \to M^3 \to M^6 \to M^7 \to M^{14} \to M^{15}$$

Requires 6 multiplications.

Let k be the number of bits of e, i.e.,

$$\begin{aligned} k &= 1 + \lfloor \log_2 e \rfloor \\ e &= (e_{k-1} e_{k-2} \cdots e_1 e_0) = \sum_{i=0}^{k-1} e_i 2^i \end{aligned}$$

for $e_i \in \{0, 1\}$.

Binary Method

Input: M, e, n.

Output: $C := M^e \bmod n$.

Step 1. **if** $e_{k-1} = 1$ **then** $C := M$ **else** $C := 1$

Step 2. **for** $i = k - 2$ **downto** 0

2a. $C := C \cdot C \pmod{n}$

2b. **if** $e_i = 1$ **then** $C := C \cdot M \pmod{n}$

Step 3. **return** C

Example:

$e = 250 = (11111010)$, thus $k = 8$.

Initially, $C = M$ since $e_{k-1} = e_7 = 1$.

i	e_i	Step 2a	Step 2b
7	1	M	M
6	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
5	1	$(M^3)^2 = M^6$	$M^6 \cdot M = M^7$
4	1	$(M^7)^2 = M^{14}$	$M^{14} \cdot M = M^{15}$
3	1	$(M^{15})^2 = M^{30}$	$M^{30} \cdot M = M^{31}$
2	0	$(M^{31})^2 = M^{62}$	M^{62}
1	1	$(M^{62})^2 = M^{124}$	$M^{124} \cdot M = M^{125}$
0	0	$(M^{125})^2 = M^{250}$	M^{250}

The number of multiplications: $7 + 5 = 12$.

The **binary method** requires:

- Squarings: $k - 1$.

  (Step 2a)

- Multiplications: The number of 1s in the binary expansion of e, excluding the MSB.

  (Step 2b)

The total number of multiplications:

$$\begin{aligned} \text{Maximum:} \quad & (k-1) + (k-1) && = 2(k-1) \\ \text{Minimum:} \quad & (k-1) + 0 && = k-1 \\ \text{Average:} \quad & (k-1) + \tfrac{1}{2}(k-1) && = 1.5(k-1) \end{aligned}$$

By scanning the bits of e

2 at a time: **quaternary method**

3 at a time: **octal method**

etc.

m at a time: m**-ary method**

Consider the quaternary method:

$$e = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$$

- Some preprocessing required.
- At each step 2 squarings performed.

Example: Quaternary method.

bits	j	M^j
00	0	1
01	1	M
10	2	$M \cdot M = M^2$
11	3	$M^2 \cdot M = M^3$

$$e = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$$

bits	Step 2a	Step 2b
11	M^3	M^3
11	$(M^3)^4 = M^{12}$	$M^{12} \cdot M^3 = M^{15}$
10	$(M^{15})^4 = M^{60}$	$M^{60} \cdot M^2 = M^{62}$
10	$(M^{62})^4 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

The number of multiplications: $2+6+3 = 11$.

Example: Octal method.

bits	j	M^j
000	0	1
001	1	M
010	2	$M \cdot M = M^2$
011	3	$M^2 \cdot M = M^3$
100	4	$M^3 \cdot M = M^4$
101	5	$M^4 \cdot M = M^5$
110	6	$M^5 \cdot M = M^6$
111	7	$M^6 \cdot M = M^7$

$$e = 250 = \underline{011}\ \underline{111}\ \underline{010}$$

bits	Step 2a	Step 2b
011	M^3	M^3
111	$(M^3)^8 = M^{24}$	$M^{24} \cdot M^7 = M^{31}$
010	$(M^{31})^8 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

The number of multiplications: $6+6+2 = 14$.
(Compute only M^2 and M^7 : $4+6+2 = 12$)

Assume $2^d = m$ and $\frac{k}{d}$ is an integer.

The average number of multiplications plus squarings required by the m-ary method:

- Preprocessing Multiplications:

  $m - 2 = 2^d - 2$

- Squarings:

  $(\frac{k}{d} - 1) \cdot d = k - d$

- Multiplications:

  $\frac{m-1}{m} \cdot (\frac{k}{d} - 1) = (1 - 2^{-d}) \cdot (\frac{k}{d} - 1)$

There is an optimum d for every k.

Average number of multiplications

k	BM	MM	d^*	Savings %
8	11	10	2	9.1
16	23	21	2	8.6
32	47	43	2,3	8.5
64	95	85	3	10.5
128	191	167	3,4	12.6
256	383	325	4	15.1
512	767	635	5	17.2
1024	1535	1246	5	18.8
2048	3071	2439	6	20.6

Reduction in the number of preprocessing multiplications:

Consider the following exponent for $k = 16$ and $d = 4$

$$\underline{1011}\ \underline{0011}\ \underline{0111}\ \underline{1000}$$

which implies that we need to compute M^w (mod n) for only $w = 3, 7, 8, 11$.

$$\begin{aligned} M^2 &= M \cdot M \\ M^3 &= M^2 \cdot M \\ M^4 &= M^2 \cdot M^2 \\ M^7 &= M^3 \cdot M^4 \\ M^8 &= M^4 \cdot M^4 \\ M^{11} &= M^8 \cdot M^3 \end{aligned}$$

This requires 6 multiplications. Computing all of the exponent values would require $16 - 2 = 14$ preprocessing multiplications.

Sliding Window Techniques

Based on adaptive (data dependent) m-ary partitioning of the exponent.

- Constant length nonzero windows

  Rule: Partition the exponent into zero words of any length and nonzero words of length d.

- Variable length nonzero windows

  Rule: Partition the exponent into zero words of length at least q and nonzero words of length at most d.

Constant length nonzero windows:

Example: For $d = 3$, we partition

$$e = 3665 = (111001010001)_2$$

as

$$\underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$$

Average number of multiplications

k	m-ary	d^*	CLNW	d^*	%
128	167	4	156	4	6.6
256	325	4	308	5	5.2
512	635	5	607	5	4.4
1024	1246	5	1195	6	4.1
2048	2439	6	2360	7	3.2

First compute M^j for odd $j \in [1, m-1]$.

bits	j	M^j
001	1	M
010	2	$M \cdot M = M^2$
011	3	$M \cdot M^2 = M^3$
101	5	$M^3 \cdot M^2 = M^5$
111	7	$M^5 \cdot M^2 = M^7$

$$3665 = \underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$$

bits	Step 2a	Step 2b
111	M^7	M^7
00	$(M^7)^4 = M^{28}$	M^{28}
101	$(M^{28})^8 = M^{224}$	$M^{224} \cdot M^5 = M^{229}$
0	$(M^{229})^2 = M^{458}$	M^{458}
001	$(M^{458})^8 = M^{3664}$	$M^{3664} \cdot M^1 = M^{3665}$

Variable length nonzero windows:

Example: $d = 5$ and $q = 2$.

$\underline{101}\ 0\ \underline{11101}\ 00\ \underline{101}$

$\underline{10111}\ 000000\ \underline{1}\ 00\ \underline{111}\ 000\ \underline{1011}$

Example: $d = 10$ and $q = 4$.

$\underline{1011011}\ 0000\ \underline{11}\ 0000$

$\underline{11110111}\ 00\ \underline{1111110101}\ 0000\ \underline{11011}$

Average number of multiplications

	m-ary		VLNW		for q^*
k	d^*	T/k	d^*	T_2/k	%
128	4	1.31	4	1.20	7.8
256	4	1.27	4	1.18	6.8
512	5	1.24	5	1.16	6.4
1024	5	1.22	6	1.15	5.8
2048	6	1.19	6	1.13	5.0

The Factor Method

The **factor method** is based on factorization of the exponent $e = rs$ where r is the smallest prime factor of e and $s > 1$.

We compute M^e by first computing M^r and then raising this value to the sth power.

$$(M^r)^s = M^{rs} = M^e$$

If e is prime, we first compute M^{e-1} then multiply this quantity by M.

Factor method: $55 = 5 \cdot 11$.

Compute $M \to M^2 \to M^4 \to M^5$;

Assign $y := M^5$;

Compute $y \to y^2$;

Assign $z := y^2$;

Compute $z \to z^2 \to z^4 \to z^5$;

Compute $z^5 \to (z^5 y) = y^{11} = M^{55}$.

Total: 8 multiplications.

Binary method: $e = 55 = (110111)_2$

$5 + 4 = 9$ multiplications.

The Power Tree Method

Consider the node e of the kth level, from left to right.

Construct the (k+1)st level by attaching below node e the nodes

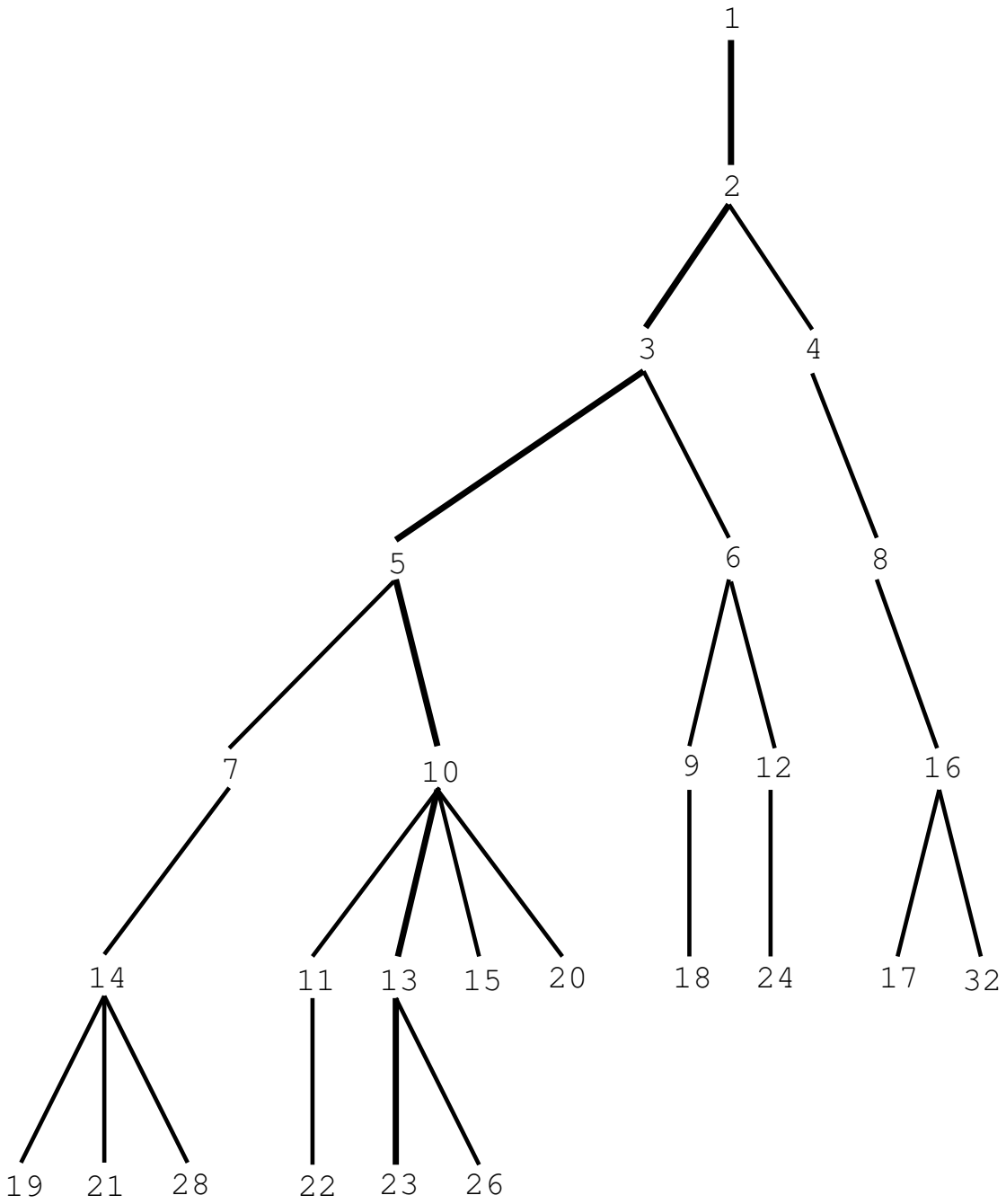
$$e + a_1, e + a_2, e + a_3, \ldots, e + a_k$$

where

$$a_1, a_2, a_3, \ldots, a_k$$

is the path from the root of the tree to e. (Note: $a_1 = 1$ and $a_k = e$)

Discard any duplicates that have already appeared in the tree.


1
2
3
4
5
6
8
7
10
9
12
16
14
11
13
15
20
18
24
17
32
19
21
28
22
23
26

Computation using power tree:

Find e in the power tree. The sequence of exponents that occurs in the computation of M^e is found on the path from the root to e.

Example: $e = 23$ requires 6 multiplications.

$$M \to M^2 \to M^3 \to M^5 \to M^{10} \to M^{13} \to M^{23}$$

Since $23 = (10111)$, the **binary method** requires $4 + 3 = 7$ multiplications.

Since $23 - 1 = 22 = 2 \cdot 11$, **the factor method** requires $1 + 5 + 1 = 7$ multiplications:

$$M \to M^2 \quad (:= y)$$

$$y \to y^2 \to y^4 \to y^8 \to y^{10} \to y^{11}$$

$$y^{11} \to M^{23}$$

Addition Chains

Consider a sequence of integers

$$a_0, a_1, a_2, \ldots, a_r$$

with $a_0 = 1$ and $a_r = e$. The sequence is constructed in such a way that for all k there exist indices $i, j \leq k$ such that

$$a_k = a_i + a_j$$

The **length** of the chain is r.

A short chain for a given e implies an efficient algorithm for computing M^e.

Example: $e = 55$

BM:	1	2	3	6	12	13	26	27	54	55
QM:	1	2	3	6	12	13	26	52	55	
FM:	1	2	4	5	10	20	40	50	55	
PTM:	1	2	3	5	10	11	22	44	55	

- Finding the **shortest** addition chain is NP-complete.

- **Upper-bound** is given by binary method:

  $\lfloor \log e \rfloor + H(e) - 1$

  $H(e)$ is the Hamming weight of e

- **Lower-bound** given by Schönhage:

  $\log e + \log H(e) - 2.13$

- **Heuristics:** binary, m-ary, adaptive m-ary, sliding windows, power tree, factor.

- Statistical methods, such as **simulated annealing**, can be used to produce short addition chains for certain exponents.

Vector Addition Chains

A list of vectors with the following properties:

- The initial vectors are the unit vectors: $[1,0,0]$, $[0,$
- Each vector is the sum of two earlier vectors.
- The last vector is equal to the given vector.

This problem arises in conjunction with reducing the preprocessing multiplications in the m-ary methods and the sliding window techniques.

For example, given the vector $[7, 15, 23]$, we obtain a vector addition as

$$\begin{array}{c} [1,0,0] \\ [0,1,0] \quad [0,1,1] \quad [1,1,1] \quad [0,1,2] \quad [1,2,3] \\ [0,0,1] \end{array}$$

$$[1,3,5] \quad [2,4,6] \quad [3,7,11] \quad [4,8,12] \quad [7,15,23]$$

which is of length 9.

An addition sequence is simply an addition chain where the i requested numbers occur somewhere in the chain.

It has been established that an addition sequence of length r and i requested numbers can be converted to a vector addition chain of length $r + i - 1$ with dimension i.

Addition-Subtraction Chains

Convert the binary number to a signed-digit representation using the digits $\{0, 1, -1\}$.

These techniques use the identity

$$2^{i+j-1} + 2^{i+j-2} + \cdots + 2^i = 2^{i+j} - 2^i$$

to collapse a block of 1s in order to obtain a **sparse** representation of the exponent.

Example:

$$\begin{aligned} (011110) &= 2^4 + 2^3 + 2^2 + 2^1 \\ (100\bar{1}0) &= 2^5 - 2^1 \end{aligned}$$

These methods require that $M^{-1} \pmod{n}$ be supplied along with M.

Recoding Binary Method

Input: M, M^{-1}, e, n.

Output: $C := M^e \bmod n$.

Step 0. Obtain signed-digit recoding d of e.

Step 1. **if** $d_k = 1$ **then** $C := M$ **else** $C := 1$

Step 2. **for** $i = k - 1$ **downto** 0

2a. $C := C \cdot C \pmod n$

2b. **if** $d_i = 1$ **then** $C := C \cdot M \pmod n$

if $d_i = \bar{1}$ **then** $C := C \cdot M^{-1} \pmod n$

Step 3. **return** C

This algorithm is especially useful for elliptic-curve cryptosystems since the inverse is available at no cost.

Example:

$e = 119 = (1110111)$.

Binary method: $6 + 5 = 11$ multiplications.

Exponent: 01110111
Recoded Exponent: $1000\bar{1}00\bar{1}$

d_i	Step 2a	Step 2b
1	M	M
0	$(M)^2 = M^2$	M^2
0	$(M^2)^2 = M^4$	M^4
0	$(M^4)^2 = M^8$	M^8
$\bar{1}$	$(M^8)^2 = M^{16}$	$M^{16} \cdot M^{-1} = M^{15}$
0	$(M^{15})^2 = M^{30}$	M^{30}
0	$(M^{30})^2 = M^{60}$	M^{60}
$\bar{1}$	$(M^{60})^2 = M^{120}$	$M^{120} \cdot M^{-1} = M^{119}$

The number of multiplications: $7 + 2 = 9$.

Modular Multiplication

Computation of $y = a \cdot b \pmod{n}$ can be performed by using:

- Multiply and then divide:

  Multiply $x := a \cdot b$ ($2k$-bit number)

  Divide: $y := x \ \% \ n$ (k-bit number)

- Blakley's method:

  The multiplication steps are interleaved with reduction steps.

- Montgomery's method:

  Uses predominantly modulo 2^j arithmetic.

Multiprecision Multiplication

$$a = (a_{s-1}a_{s-2}\cdots a_0)_W = \sum_{j=0}^{s-1} a_i W^i$$

$$b = (b_{s-1}b_{s-2}\cdots b_0)_W = \sum_{j=0}^{s-1} b_i W^i$$

Radix $W = 2^w$: wordsize of the computer.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				t_{03}	t_{02}	t_{01}	t_{00}
			t_{13}	t_{12}	t_{11}	t_{10}	
		t_{23}	t_{22}	t_{21}	t_{20}		
	t_{33}	t_{32}	t_{31}	t_{30}			
t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0

$t_{ij} = b_i \cdot a_j$: (Carry, Sum) pairs.

a, b : a_i, b_i for $i = 0, 1, \ldots, s-1$

t : t_i for $i = 0, 1, \ldots, 2s-1$

Standard Multiplication Algorithm

for $i = 0$ **to** $s-1$
    begin
    $C := 0$
    for $j = 0$ **to** $s-1$
        begin
        $(C, S) := t_{i+j} + a_j \cdot b_i + C$
        $t_{i+j} := S$
        end
    $t_{i+j+1} := C$
    end

This algorithm requires $s^2 = (k/w)^2$ innerproduct steps:

$$(C, S) := t_{i+j} + a_j \cdot b_i + C$$

in other words, $O(k^2)$ bit operations.

The variables t_{i+j}, a_j, b_i, C, and S each hold a single-word, or a w-bit number.

From this operation, we obtain a double-word, or a $2w$-bit number since

$$2^W - 1 + (2^W - 1)(2^W - 1) + 2^W - 1 = 2^{2W} - 1$$

Example: $a \cdot b = 348 \cdot 857$

i	j	Step	(C, S)	Partial t
0	0	$t_0 + a_0 b_0 + C$	$(0, *)$	000000
		$0 + 8 \cdot 7 + 0$	$(5, 6)$	00000**6**
	1	$t_1 + a_1 b_0 + C$		
		$0 + 4 \cdot 7 + 5$	$(3, 3)$	0000**3**6
	2	$t_2 + a_2 b_0 + C$		
		$0 + 3 \cdot 7 + 3$	$(2, 4)$	000**4**36
				00**2**436
1	0	$t_1 + a_0 b_1 + C$	$(0, *)$	
		$3 + 8 \cdot 5 + 0$	$(4, 3)$	0024**3**6
	1	$t_2 + a_1 b_1 + C$		
		$4 + 4 \cdot 5 + 4$	$(2, 8)$	002**8**36
	2	$t_3 + a_2 b_1 + C$		
		$2 + 3 \cdot 5 + 2$	$(1, 9)$	00**9**836
				0**1**9836
2	0	$t_2 + a_0 b_2 + C$	$(0, *)$	
		$8 + 8 \cdot 8 + 0$	$(7, 2)$	019**2**36
	1	$t_3 + a_1 b_2 + C$		
		$9 + 4 \cdot 8 + 7$	$(4, 8)$	01**8**236
	2	$t_4 + a_2 b_2 + C$		
		$1 + 3 \cdot 8 + 4$	$(2, 9)$	0**9**8236
				298236

Squaring is easier:

Squaring is an easier operation than multiplication since half of the single-precision multiplications can be skipped. This is due to the fact that $t_{ij} = a_i \cdot a_j = t_{ji}$.

				a_3	a_2	a_1	a_0
$\times$				a_3	a_2	a_1	a_0
				t_{03}	t_{02}	t_{01}	t_{00}
			t_{13}	t_{12}	t_{11}	t_{01}	
		t_{23}	t_{22}	t_{12}	t_{02}		
$+$	t_{33}	t_{23}	t_{13}	t_{03}			
				$2t_{03}$	$2t_{02}$	$2t_{01}$	t_{00}
			$2t_{13}$	$2t_{12}$	t_{11}		
		$2t_{23}$	t_{22}				
$+$	t_{33}						
t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0

Standard Squaring Algorithm

for $i = 0$ **to** $s - 1$
    $(C, S) := t_{i+i} + a_i \cdot a_i$
    for $j = i + 1$ **to** $s - 1$
        $(C, S) := t_{i+j} + 2 \cdot a_j \cdot a_i + C$
        $t_{i+j} := S$
    $t_{i+j+1} := C$
return $(t_{2s-1} t_{2s-2} \cdots t_0)$

However, note that the carry-sum pair produced by operation

$$(C, S) := t_{i+j} + 2 \cdot a_j \cdot a_i + C$$

in Step 4 may be 1 bit longer than a double-word, or a $2w$-bit number since

$$(2^w - 1) + 2(2^w - 1)(2^w - 1) + (2^w - 1)$$

is equal to $2^{2w+1} - 2^{w+1}$ and

$$2^{2w} - 1 < 2^{2w+1} - 2^{w+1} < 2^{2w+1} - 1$$

Recursive Algorithms

Assuming $k = 2h$, we decompose a and b into two equal-size parts:

$$\begin{aligned} a &:= 2^h a_1 + a_0 \\ b &:= 2^h b_1 + b_0 \end{aligned}$$

Then

$$\begin{aligned} t &:= a \cdot b \\ &:= (2^h a_1 + a_0)(2^h b_1 + b_0) \\ &:= 2^{2h} a_1 b_1 + 2^h (a_1 b_0 + a_0 b_1) + a_0 b_0 \end{aligned}$$

Standard Recursive Algorithm

sra(a, b)
$t_0 :=$ sra(a_0, b_0);
$t_1 :=$ sra(a_0, b_1);
$t_2 :=$ sra(a_1, b_0);
$t_3 :=$ sra(a_1, b_1);
return $(2^{2h} t_3 + 2^h (t_1 + t_2) + t_0)$

The standard recursive algorithm breaks the multiplication of **two** $2h$-bit integers into multiplication of **four** h-bit integers.

The number of bit operations is found by solving the recursion

$$T(k) = 4T(k/2) + \alpha k$$

with $T(1) = 1$. This gives $T(k) = O(k^2)$ bit operations.

Karatsuba-Ofman algorithm is based on the observation that **three** h-bit multiplications suffice:

$$\begin{aligned} t_0 &:= a_0 \cdot b_0 \\ t_3 &:= a_1 \cdot b_1 \\ t_1 + t_2 &:= (a_1 + a_0) \cdot (b_1 + b_0) - t_0 - t_3 \end{aligned}$$

Karatsuba-Ofman Recursive Algorithm

kora(a, b)
$t_0 :=$ kora(a_0, b_0);
$t_3 :=$ kora(a_1, b_1);
$t_{12} :=$ kora$(a_1 + b_0, b_1 + b_0)$;
return $(2^{2h} t_3 + 2^h t_{12} + t_0)$

The number of bit operations is given by the recursion

$$T(k) = 3T(k/2) + \beta k$$

with $T(k) = 1$. This gives

$$O(k^{\log_2 3}) = O(k^{1.58})$$

bit operations.

The Karatsuba-Ofman algorithm is **asymptotically** faster.

However, due to the **recursive** nature of the algorithm, there is a large **overhead**.

Karatsuba-Ofman algorithm starts paying off as k gets larger. After about $k = 100$, it starts being faster than the regular algorithm.

We also have the option of stopping at any point during the recursion. For example, we may apply one level of recursion and then compute the required three multiplications using the standard algorithm.

FFT-based Multiplication Algorithm

The fastest multiplication algorithm is based on the fast Fourier transform. The FFT algorithm is used to multiply polynomials. Multiprecision integers can be considered as polynomials evaluated at the radix.

For example: $348 = 3x^2 + 4x + 8$ at $x = 10$.

Similary, $857 = 8x^2 + 5x + 7$ at $x = 10$.

In order to multiply 348 by 857, we first multiply the polynomials

$$(3x^2 + 4x + 8)(8x^2 + 5x + 7)$$

then evaluate the resulting polynomial

$$24x^4 + 47x^3 + 105x^2 + 68x + 56$$

at the radix $x = 10$ to obtain the product $348 \cdot 857 = 298236$.

Let the polynomials $a(x)$ and $b(x)$

$$a(x) = \sum_{i=0}^{k-1} a_i x^i \ , \quad b(x) = \sum_{i=0}^{k-1} b_i x^i$$

denote the multiprecision numbers

$$a = (a_{k-1} a_{k-2} \cdots a_0)$$

$$b = (b_{k-1} b_{k-2} \cdots b_0)$$

represented in radix W where a_i and b_i are the 'digits' with the property $0 \leq a_i, b_i \leq W - 1$.

Let the integer $l = 2k$ be a power of 2. Given the primitive lth root of unity ω, the following algorithm computes the product

$$t = (t_{l-1} t_{l-2} \cdots t_0)$$

FFT-based Multiplication Algorithm

1. Evaluate $a(w^i)$ and $b(w^i)$ for $i = 0, 1, .., l-1$ by calling the fast Fourier transform procedure.

2. Multiply pointwise to obtain

$$\{a(1)b(1), a(w)b(w), \ldots, a(w^{l-1})b(w^{l-1})\}$$

3. Interpolate $t(x) = \sum_{i=0}^{l-1} t_i x^i$ by evaluating

$$l^{-1} \sum_{i=0}^{l-1} a(w^i)b(w^i)x^i$$

on $\{1, w^{-1}, \ldots, w^{-(l-1)}\}$ by calling the fast Fourier transform procedure.

4. Return the coefficients

$$(t_{l-1}, t_{l-2}, \ldots, t_0)$$

Which field to use?

Complex field is unsuitable for computer implementation since the lth primitive root of unity $e^{2\pi j/l}$ (where $j = \sqrt{-1}$) may be **irrational**. These numbers cannot be represented or operated on our computers which perform **finite precision arithmetic**.

Pollard showed that a finite field in which l^{-1} and a primitive lth root of unity exist can be used. The choice is the Galois field of p elements where p is a prime and l divides $p - 1$.

Example:
$p = 2130706433 = 127 \cdot 2^{24} + 1$. The field $GF(p)$ can be used to compute FFTs of size $l = 2^{24} \approx 10^7$ on a 32-bit machine.

The FFT-based multiplication algorithm was discovered by Schönhage and Strassen. It requires

$$O(k \log k \log \log k)$$

bit operations to multiply two k-bit numbers.

However, the constant in front of the order function is very high. It starts paying off for numbers with **several thousand** bits.

Computation of the Remainder

Given t, the computation of R which satisfies

$$t = Q \cdot n + R$$

with $R < n$. Here t is a $2k$-bit number and n is a k-bit number.

The numbers t and n are positive, so are the results Q and R.

Since we are not interested in the quotient, steps of the division algorithm can be simplified.

Two algorithms are of interest to us:

- Restoring division
- Nonrestoring division

Restoring Division Algorithm

$R_0 := t$
$n := 2^k n$
for $i = 1$ **to** k
    $R_i := R_{i-1} - n$
    if $R_i < 0$ **then** $R_i := R_{i-1}$
    $n := n/2$
return R_k

We give an example of the restoring division algorithm for computing 3019 mod 53, where

$$3019 = (101111001011)_2$$
$$53 = (110101)_2$$

The result is

$$51 = (110011)_2$$

R_0		101111	001011	t
n		110101		subtract
	-	000110		negative rem.
R_1		101111	001011	restore
$n/2$		11010	1	subtract
	+	10100	1	positive rem.
R_2		10100	101011	not restore
$n/2$		1101	01	subtract
	+	0111	01	positive rem.
R_3		0111	011011	not restore
$n/2$		110	101	subtract
	+	000	110	positive rem.
R_4		000	110011	not restore
$n/2$		11	0101	
$n/2$		1	10101	
$n/2$			110101	subtract
	-		000010	negative rem.
R_5			110011	restore
R			110011	Final rem.

Nonrestoring Division Algorithm

The nonrestoring division algorithm allows a negative remainder.

Suppose $R_i = R_{i-1} - n < 0$, then the restoring algorithm assigns $R_i := R_{i-1}$ and performs a subtraction with the shifted n, obtaining

$$R_{i+1} = R_i - n/2 = R_{i-1} - n/2$$

However, if $R_i = R_{i-1} - n < 0$, then the nonrestoring algorithm lets R_i remain negative and adds the shifted n in the following cycle. Thus, it obtains

$$R_{i+1} = R_i + n/2 = (R_{i-1} - n) + n/2$$

which is equal to $R_{i-1} - n/2$, i.e., the same value.

$R_0 := t$
$n := 2^k n$
for $i = 1$ **to** k
    if $R_{i-1} > 0$ **then** $R_i := R_{i-1} - n$
    else $R_i := R_{i-1} + n$
    $n := n/2$
return R_k

Since the remainder is allowed to stay negative, we use 2's complement coding to represent such numbers.

Also, note that the nonrestoring division algorithm may require a final restoration cycle in which a negative remainder is corrected by adding the last value of n back to it.

Example: Computation of $51 = 3019 \bmod 53$ using the nonrestoring division algorithm.

R_0	0101111	001011	t
n	0110101		subtract
R_1	1111010		negative rem.
$n/2$	011010	1	add
R_2	010100	1	positive rem.
$n/2$	01101	01	subtract
R_3	00111	01	positive rem.
$n/2$	0110	101	subtract
R_4	0000	110	positive rem.
$n/2$	011	0101	
$n/2$	01	10101	
$n/2$	0	110101	subtract
R_5	1	111110	negative rem.
n	0	110101	add (restore)
R	0	110011	Final rem.

Blakley's Method

Let a_i and b_i represent the bits of the k-bit numbers a and b, respectively. The product t ($2k$-bit number) can be written as

$$t = a \cdot b = \left(\sum_{i=0}^{k-1} a_i 2^i \right) \cdot b = \sum_{i=0}^{k-1} (a_i \cdot b) 2^i$$

This formulation yields the shift-add multiplication algorithm. Blakley's algorithm uses this formulation and furthermore reduces the partial product modulo n at each step.

$R := 0$
for $i = 0$ **to** $k - 1$
    begin
    $R := 2R + a_{k-1-i} \cdot b$
    $R := R \bmod n$
    end
return R

Assuming that $0 \leq a, b, R \leq n-1$, the new R will be in the range

$$0 \leq R \leq 3n-3$$

since

$$\begin{aligned} R &:= 2R + a_j \cdot b \\ &\leq 2(n-1) + (n-1) = 3n-3 \end{aligned}$$

At most 2 subtractions will be needed to bring the new R to the range $[0, n-1]$. Thus, we can use

If $R \geq n$ then $R := R - n$
If $R \geq n$ then $R := R - n$

Blakley's algorithm computes the remainder R in k steps, where at each step one left shift, one addition, and at most two subtractions are performed; the operands involved in these computations are of length k bits.

Montgomery's Method

This method replaces division by n operation with division by $r = 2^k$.

Assuming n is a k-bit integer, i.e.,

$$2^{k-1} < n < 2^k$$

we assign $r = 2^k$.

Now, we map the integers $a \in [0, n-1]$ to the integers $\bar{a} \in [0, n-1]$ using the one-to-one mapping

$$\bar{a} := a \cdot r \pmod{n}$$

We call $\bar{a}$ the n-residue of a.

We now define the **Montgomery product** of two n-residues as

$$\text{MonPro}(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

where r^{-1} is the inverse of r modulo n. Also we need n' such that

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

r^{-1} and n' are computed by the extended Euclid algorithm.

function MonPro($\bar{a}, \bar{b}$)
Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $m := t \cdot n' \bmod r$
Step 3. $u := (t + m \cdot n)/r$
Step 4. **if** $u \geq n$ **then return** $u - n$
else return u

This subroutine requires only modulo r arithmetic, which is easily accomplished on a computer if $r = 2^j$.

Theorem 1:

$$\begin{aligned} \text{If} \quad c &= a \cdot b \pmod{n} \\ \text{Then} \quad \bar{c} &= \text{MonPro}(\bar{a}, \bar{b}) \end{aligned}$$

$$\begin{aligned} \underline{\text{Proof:}} \quad \bar{c} &= c \cdot r \pmod{n} \\ &= a \cdot b \cdot r \pmod{n} \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \\ &= \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \\ &= \text{MonPro}(\bar{a}, \bar{b}) \end{aligned}$$

Theorem 2:

$$c = \text{MonPro}(\bar{c}, 1)$$

$$\begin{aligned} \underline{\text{Proof:}} \quad c &= c \cdot r \cdot r^{-1} \pmod{n} \\ &= \bar{c} \cdot r^{-1} \pmod{n} \\ &= \bar{c} \cdot 1 \cdot r^{-1} \pmod{n} \\ &= \text{MonPro}(\bar{c}, 1) \end{aligned}$$

MonPro procedure can be utilized to compute

$$c := a \cdot b \pmod{n}$$

as follows:

function ModMul(a, b, n) $\{$ n is odd $\}$
Step 1. Compute n' using Euclid's algorithm.
Step 2. $\bar{a} := a \cdot r \bmod n$
Step 3. $\bar{b} := b \cdot r \bmod n$
Step 4. $\bar{c} :=$ MonPro$(\bar{a}, \bar{b})$
Step 7. $c :=$ MonPro$(\bar{c}, 1)$
Step 8. **return** c

Since preprocessing operations

- computation of n' and
- conversion from ordinary to n-residue
- conversion from n-residue to ordinary

are time-consuming, it is not a good idea to use Montgomery's method for a single modular multiplication.

However, it is very suitable for modular exponentiation.

function ModExp(M, e, n) { n is odd }
Step 1. Compute n' using Euclid's algorithm.
Step 2. $\bar{M} := M \cdot r \bmod n$
Step 3. $\bar{C} := 1 \cdot r \bmod n$
Step 4. **for** $i = k - 1$ **down to** 0 **do**
Step 5. $\quad \bar{C} :=$ MonPro($\bar{C}, \bar{C}$)
Step 6. $\quad$ **if** $e_i = 1$ **then** $\bar{C} :=$ MonPro($\bar{M}, \bar{C}$)
Step 7. $C :=$ MonPro($\bar{C}, 1$)
Step 8. **return** C

The above function uses the binary method. However, anyone of the addition chain heuristics can be used here as well.

Example: Computation of $7^{10} \pmod{13}$

$r = 2^k = 16$. Since

$$16 \cdot 9 - 13 \cdot 11 = 1$$

we have $r^{-1} = 9$ and $n' = 11$.

$M = 7$

$\bar{M} := M \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8$

$C = 1$

$\bar{C} := C \cdot r \pmod{n} = 1 \cdot 16 \pmod{13} = 3$

Thus, $\bar{C} = 3$ and $\bar{M} = 8$

e_i	Step 5	Step 6
1	MonPro$(3,3) = 3$	MonPro$(8,3) = 8$
0	MonPro$(8,8) = 4$	
1	MonPro$(4,4) = 1$	MonPro$(8,1) = 7$
0	MonPro$(7,7) = 12$	

Step 7: $C = \text{MonPro}(12,1) = 4$

• Computation of MonPro$(3,3)$:

$t := 3 \cdot 3 = 9$
$m := 9 \cdot 11 \pmod{16} = 3$
$u := (9 + 3 \cdot 13)/16 = 48/16 = 3$

• Computation of MonPro$(8,1)$:

$t := 8 \cdot 1 = 8$
$m := 8 \cdot 11 \pmod{16} = 8$
$u := (8 + 8 \cdot 13)/16 = 112/16 = 7$

Computation of GCD and Inverse

Euclid's algorithm is based on the following observation:

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

Thus, given a and b, we compute $\gcd(a, b)$ by successive modular reductions:

$$\begin{aligned} r_1 &= a \bmod b \\ r_2 &= b \bmod r_1 \\ r_3 &= r_1 \bmod r_2 \\ r_4 &= r_2 \bmod r_3 \\ &\vdots \end{aligned}$$

until we reach an $r_i = 0$.

The greatest common divisor of a and b is then $\gcd(a, b) = r_{i-1}$.

Example: Computation of $\gcd(1812, 1572)$

$$\begin{aligned} 240 &= 1812 \bmod 1572 \\ 132 &= 1572 \bmod 240 \\ 108 &= 240 \bmod 132 \\ 24 &= 132 \bmod 108 \\ 12 &= 108 \bmod 24 \\ 0 &= 24 \bmod 12 \end{aligned}$$

Thus, $\gcd(1812, 1572) = 12$.

Given a and b with with $0 \leq a \leq b$, Euclid's algorithm requires less than

$$\frac{\log_e(\sqrt{5}\ b)}{\log_e(\frac{1+\sqrt{5}}{2})} - 1$$

steps to compute $\gcd(a, b)$.

The **binary Euclid algorithm** replaces modular reductions by shift (division by 2) and subtraction operations. It is based on the following observations:

For any a and b

$$\gcd(a, b) = \gcd(b, b - a)$$

If a and b are both even then

$$\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$$

If a is odd and b is even then

$$\gcd(a, b) = \gcd(a, b/2)$$

Example: Computation of $\gcd(182, 98)$

$$\begin{aligned}
\gcd(182, 98) &= 2 \cdot \gcd(91, 49) \\
&= 2 \cdot \gcd(49, 42) \\
&= 2 \cdot \gcd(49, 21) \\
&= 2 \cdot \gcd(21, 28) \\
&= 2 \cdot \gcd(21, 14) \\
&= 2 \cdot \gcd(21, 7) \\
&= 2 \cdot \gcd(7, 14) \\
&= 2 \cdot \gcd(7, 7) \\
&= 2 \cdot 7 \\
&= 14
\end{aligned}$$

Extended Euclid algorithm

Given a and b, the extended Euclid algorithm computes g, u, and v such that

$$g = \gcd(a, b) = u \cdot a + v \cdot b$$

This algorithm is used to compute the modular inverse. If $g = 1$, then

$$1 = u \cdot a + v \cdot b$$

implies that

$$\begin{aligned} 1 &= u \cdot a \pmod{b} \\ 1 &= v \cdot b \pmod{a} \end{aligned}$$

and therefore

$$\begin{aligned} u &= a^{-1} \pmod{b} \\ v &= b^{-1} \pmod{a} \end{aligned}$$

EEA(a, b, g, u, v)
begin
$(g_0, g_1) = (a, b)$
$(u_0, u_1) = (1, 0)$
$(v_0, v_1) = (0, 1)$
while $g_1 \neq 0$ **do**
begin
$q = g_0$ div g_1
$(g_0, g_1) = (g_1, g_0 - g_1 \cdot q)$
$(u_0, u_1) = (u_1, u_0 - u_1 \cdot q)$
$(v_0, v_1) = (v_1, v_0 - v_1 \cdot q)$
end
$g = g_0$; $u = u_0$; $v = v_0$
end

Example: $a = 21$ and $b = 16$

iteration	q	g_0	g_1	u_0	u_1	v_0	v_1
0	-	21	16	1	0	0	1
1	1	16	5	0	1	1	-1
2	3	5	1	1	-3	-1	4
3	5	1	0	-3	16	4	-21
		•		•		•	

EEA returns $g = 1$, $u = -3$ and $v = 4$

This implies

$$1 = -3 \cdot 21 + 4 \cdot 16$$

Therefore

$$\begin{aligned} 21^{-1} &= -3 \pmod{16} \\ 16^{-1} &= 4 \pmod{21} \end{aligned}$$

Chinese remainder theorem

Let m_1 and m_2 be relatively prime integers, i.e., $\gcd(m_1, m_2) = 1$. Given $x_1 \in [0, m_1)$ and $x_2 \in [0, m_2)$, there exists a unique integer $X \in [0, m_1 m_2)$ such that

$$\begin{aligned} X &= x_1 \bmod m_1 \\ X &= x_2 \bmod m_2 \end{aligned}$$

which is determined by

$$X = x_1 c_1 m_2 + x_2 c_2 m_1 \pmod{m_1 m_2}$$

where

$$\begin{aligned} c_1 &= m_2^{-1} \bmod m_1 \\ c_2 &= m_1^{-1} \bmod m_2 \end{aligned}$$

Another (simpler) formula for X:

$$X = x_1 + m_1 \, [(x_2 - x_1) c_2 \bmod m_2]$$

Fast Decryption using CRT

Quisquater & Couvreur showed that the decryption process, i.e., computation of

$$M := C^d \pmod{n}$$

can be performed faster using the CRT.

Knowing the factors p and q of n, we compute

$$\begin{aligned} M_1 &:= C^d := C^{d_1} \pmod{p} \\ M_2 &:= C^d := C^{d_2} \pmod{q} \end{aligned}$$

where

$$\begin{aligned} d_1 &:= d \pmod{p-1} \\ d_2 &:= d \pmod{q-1} \end{aligned}$$

M is then computed using the Chinese remainder algorithm:

$$M := M_1 + p\ [(M_2 - M_1)p^{-1} \bmod q]$$

The sizes of p and q are half of the size of n.

This technique reduces a k-bit modular exponentiation into two $k/2$-bit modular exponentiations plus the CRT step given above.

Also p^{-1} can be precomputed and saved.

Typical Actual Timing Results

Proc.	MHz	W	512-bit	CRT	Ref.
56000	20	24	44 ms	yes	•
68020	20	32	445 ms	yes	†
80386	16	32	568 ms	yes	†
2105	10	16	1320 ms	no	∗
80286	8	16	2500 ms	yes	†

• Kaliski & Dussé 1990

† Laurichesse & Blain 1991

∗ Koç 1992

RSA Estimates for 512-bit with CRT

Processor	MHz	W	T (sec)
Motorola 56000	20	24	0.04
TI TMS320C30	33	12	0.05
NEC uPD77C25	8	15	0.38
Analog Devices 2105	10	16	0.38
TI TMS320C25	8	16	0.54
Mitsubishi M3770	8	16	4.80
Intel 8096	4	16	8.76
Zilog Z80180	10	8	10.75
Zilog Z80	10	8	48.55
Motorola 6811	2	8	49.89
Intel 8051	12	8	55.65
Motorola 6805SC27	2	8	68.04
Hitachi HD63701	2	8	75.80

Primality Testing

Fermat's Theorem: If m is prime, then for any a such that $\gcd(a, m) = 1$ we have

$$a^{m-1} = 1 \pmod{m}$$

If m is not prime, it is still possible (but probably not very likely) that the above holds.

If m is an odd composite number and a is an integer such that $\gcd(a, m) = 1$ and

$$a^{m-1} = 1 \pmod{m}$$

then we say that m is a **pseudoprime to the base** a.

For example, 341 is a pseudoprime to the base 2 since

$$2^{341-1} = 1 \pmod{341}$$

A pseudoprime is a number that "pretends" to be a prime by passing Fermat's test.

Pseudoprimality Test

Fermat's theorem provides a simple and efficient test for discarding nonprimes. Any number that fails the test is not prime.

We first check if m is a pseudoprime for the base 2. If not, then m is composite. If m passes for the base 2, we check for other bases. For example,

$$3^{341-1} = 56 \pmod{341}$$

and thus 341 is **composite**.

If m fails Fermat's test for a single base $a \in [1, m)$, then m fails for at least half of the bases $a \in [1, m)$.

Thus, unless m happens to pass the test for all possible a with $\gcd(a, m) = 1$, we have at least a 50 % chance that m will fail the test for a randomly chosen a.

Pseudoprimality Test

Given m

- Randomly pick an $a \in [1, m)$
- Compute $g = \gcd(a, m)$
- If $g \neq 1$ then m is composite
- If $g = 1$ then compute $h = a^{m-1} \bmod m$
- If $h \neq 1$ then m fails the test
- If $h = 1$ then m passes the test

If m passes the test then it is probably prime; if m fails the test then it is composite.

Repeat this test for r different bases. The probability of a composite m passing r tests is 1 in 2^r.

Unfortunately, there are numbers which pass Fermat's test for all bases to which they are relatively prime. These are **Carmichael** numbers and there are infinitely many of them.

Strong Pseudoprimality Test

Given m

- Write $m - 1 = 2^s t$ where t is odd
- Randomly pick an $a \in [1, m)$
- Compute $h = a^t \bmod m$
- If $h = -1$ or $+1$, then m passes
- Compute $h_i = a^{2^i t}$ for $i = 1, 2, \ldots, s$
- If $h_i = -1$ for some $i < s$, then m passes
- Otherwise m fails

Repeat this test for r different bases. The probability of a composite m passing r tests is 1 in 4^r.

If m passes the test for 100 different bases, then the probability that m is composite is less than 10^{-60}.

Good news: There are no strong Carmichael numbers.

Example: $m = 25326001$

Write $m - 1 = 25326000 = 8 \cdot 1582875 = 2^4 \cdot t$

Pick $a = 2$
$h = 2^t \bmod m$ gives $h = -1$
PASS

Pick $a = 3$
$h = 3^t \bmod m$ gives $h = -1$
PASS

Pick $a = 7$
$h = 7^t \bmod m$ gives $h = 19453141$
$h_1 = 7^{2t} \bmod m$ gives $h_1 = 16857740$
$h_2 = 7^{4t} \bmod m$ gives $h_2 = 11448587$
$h_3 = 7^{8t} \bmod m$ gives $h_3 = 10127250$
FAIL

Thus, m is composite.

$(25326001 = 2251 \cdot 11251)$

Generating RSA Primes

1. Pick a k-bit odd m uniformly at random from $[m/2, m]$

2. Apply test division on m by all primes less than a certain small prime

3. If m passes trial division test, then apply the (strong) pseudoprimality test for r different bases $2, 3, 5, 7, \ldots$

4. If m passes all r (strong) pseudoprimality tests, then m is a prime number with high probability

5. If m fails, take $m := m + 2$ go to Step 2