AN ALGORITHM FOR MODULAR EXPONENTIATION

Robert Willoner & I-Ngo Chen

Intel Corporation, Santa Clara, California &
Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada

Abstract

The best known algorithm for modular exponentiation, $M^e \bmod t$ for arbitrary M, e and t is of $O(n^3)$ where n is the number of bits in the largest of M, e and t. This paper presents an $O(n^2)$ algorithm for the problem where $M^e \bmod t$ is required for many values of M and e with constant t. Some preprocessing is done on t, and the results are applied repeatedly to different values of M and e. The main algorithm involves on-line arithmetic in a redundant number system. An immediate application is in encoding/decoding of messages in an RSA-based public-key cryptosystem.

Introduction

There has been considerable mention in the recent literature [1] about "public-key cryptosystems". A necessary component of such a PKCS is a "trap-door one-way function". Rivest, Shamir and Adleman [2] have suggested the use of modular exponentiation as the trap-door algorithm for the encoding/decoding of messages in a PKCS. This involves repeated modular exponentiation of blocks of different messages, using the same modulus. A quadratic algorithm for this will be presented in this paper.

The concept of linear, on-line arithmetic must first be developed. An operation is said to perform in linear time if its execution time is bounded from above (and below) by some constant multiple of the length of the longest of its operands. The on-line property is satisfied if, in order to generate the ith bit of the result, it is necessary and sufficient to have the operands available to the ith bit only. The operands are assumed to "flow through" the operator in a bit-by-bit, least-significant-bit-first fashion, and the results are produced in the same manner. The advantages of this mode of computation stem from the fact that a sequence of operations can be performed in an overlapped fashion, resulting in a significant speeding up over traditional sequential algorithms.

A linear on-line multiplier has been designed and published [3]. The reader is assumed to be familiar with the notation used and the results reported in that paper.

A Quadratic Algorithm for Modular Exponentiation

The problem of modular exponentiation for arbitrary arguments has been studied extensively [4]. The procedure recommended in [2] is given as Algorithm 1. This algorithm is of time complexity $O(n^3)$, where n is the maximum of the lengths of the three operands.

A more suitable algorithm for the PKCS problem is similarly based but also makes use of the linear on-line concepts developed in [3]. It is presented as Algorithm 2. As opposed to the previous algorithm, this one does not efficiently perform modular exponentiation on three arbitrary numbers of length n. Rather, if t is constant, it rapidly computes $M^e \bmod t$ for changing values of M (e may be constant or variable). This is, of course, exactly what a PKCS requires.

An analysis of Algorithm 2 will show that it performs in time $O(n^2)$. The basic elements of Algorithm 1 will also be found in Algorithm 2. Lines 9-10 perform the initialization. Line 11 is the start of the main loop for repeated squaring and multiplication. Lines 12-45 correspond to

the squaring of Q, and lines 44-61 correspond to the multiplying of Q by M (if the appropriate bit in e is 1). The key idea in the algorithm is the saving of approximately n divisions which are otherwise required. The technique for this is as follows.

A table of residues modulo t of powers of 2 from 1 through 2^v, where $v = 2(n + \lceil \log n \rceil)$, is constructed. Since t is constant for many values of M, it is irrelevant how this table is computed, and the results may later be retrieved from a ROM. The two sets of multiplications are performed on-line as described in [3]. As a given bit of a particular product $S = [s(2n)----s(1)]$ is produced by the multiplier, rather than letting the result Q increase in size exponentially, the corresponding residue is added to the accumulated value of Q. The use of redundant notation allows this to be done in one time step. In this way, Q never gets longer than $u = n + \lceil \log n \rceil$ bits. In order to keep Q in redundant notation, the temporary storage locations Q1 and Q2 are needed. The bits of Q are q(1), . . ., q(u). The residue (if the corresponding bit of S is 1) is stored in Q2, as specified in lines 35-36 and 71-72. The reduction of Q in one time step to redundant notation is done in lines 37-43 and 73-79. This entire sequence is performed n times and there are n residues to be added at most twice. Hence, the worst time for the bulk of this algorithm is $2n^2 = O(n^2)$. There is also a final division to be performed in line 82. As this is done only once, the particular algorithm used is of no interest, as long as it takes no more than $O(n^2)$ time.

An example of this procedure is given as Figure 1.

n = 4

M = 111
e = 1011
t = 1101

Residue table

p	2^p mod t
1	1
2	10
3	100
4	1000
5	11
6	110
7	1100
8	1011
9	1001
10	101
11	1010
12	111

Partial Powers

	Q
1	111
10	1010
100	10110
101	11000
1010	10001
e = 1011	11100

$M^e \equiv 10 \pmod t$

Figure 1. Example of modular exponentiation.

Conclusions

A quadratic algorithm for modular exponentiation with constant modulus has been described. It is particularly suitable for encoding/decoding messages in a PKCS based on the RSA algorithm. The need for such a system for such applications as secure electronic mail, electronic funds transfer (EFT), and even rapid encoding and decoding of telephone conversations, is becoming increasingly obvious in our technological society.

References

1. Diffie, W. and M. E. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, vol. IT-22, no. 6, pp. 644-654, November 1976

2. Rivest, R. L., A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, vol. 21, no. 2, pp. 120-126, February 1978

3. Chen, I-N. and R. Willoner, "An O(n) Parallel Multiplier with Bit-Sequential Input and Output", IEEE Transactions on Computers, vol. C-28, no. 10, October 1979

4. Knuth, D. E., The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, 1971

Algorithm 1.

Modular exponentiation by repeated squaring and multiplication

```
begin

    comment This algorithm evaluates the
       encryption function M^e mod t where
       M, e and t are arbitrary n bit
       numbers, e.g. e = [e(n)----e(1)];

    comment Initialize;
1.  Q <-- 1;

        comment Examine the bits of e
           from left to right. Square and
           multiply Q by M (mod t)
           according to the value of e(i);

2.      for i <-- n step -1 until 1 do

3.      begin

4.          Q <-- rem[Q*Q, t];
5.          if e(i) = 1 then
               Q <-- rem[Q*M, t]

6.      end

end.
```

Algorithm 2.

Quadratic modular exponentiation algorithm

```
begin

    comment Given M = [m(n)----m(1)],
       e = [e(n)----e(1)] and t =
       [t(n)----t(1)] (M is given on-line
       with the least significant bit
       first), this algorithm computes Q
       = [q(n)----q(1)], the ciphertext
       corresponding to M (Q is also
       produced on-line);

    comment Initialize;
1.  u <-- n + ⌈log n⌉;
2.  v <-- 2u;

    comment Compute the u residues r1,
       r2, . . . , ru of t;
3.  p <-- 1;
4.  for i <-- 1 until u do
5.     begin
6.        [ri(n)----ri(1)] <-- rem[p, t];
7.        p <-- 2p
8.     end;

9.  q(1) <-- 1;
10. for i <-- 2 until v do q(i) <-- 0;

11. for k <-- n step -1 until 1 do
12. begin
       comment Square Q;
13.    for i <-- 1 until u do

14.    begin
15.       for j <-- 1 until u do
16.          begin
17.             A(j) <-- 0;
18.             b(j) <-- 0
19.          end;

20.    if q(i) = 1 then
21.       begin
22.          for j <-- 1 until i do
                A(i+j-1) <-- q(j);
23.          if i > 1 then
24.             for j <-- 1 until i-1 do
                   b(i+j-1) <-- q(j)
25.       end;

26.    for j <-- 2n-1 step -1 until 1 do
27.       begin
28.          t(1) <-- S23[A(j), b(j),
                C1(j-1), C2(j-2), s(j)];
29.          t(2) <-- S45[A(j), b(j),
                C1(j-1), C2(j-2), s(j)];
30.          t(3) <-- S135[A(j), b(j),
                C1(j-1), C2(j-2), s(j)];
31.          C1(j) <-- t(1);
32.          C2(j) <-- t(2);
33.          s(j) <-- t(3);
34.       end;

35.    for j <-- 1 until n do
          Q2(j) <-- 0;
36.    if s(i) = 1 then
          for j <-- 1 until n do
             Q2(j) <-- ri(j);

37.    for j <-- n-1 step -1 until 1 do
38.       begin
39.          t(1) <-- S13[Q1(j), Q2(j-1),
                q(j)];
40.          t(2) <-- S23[Q1(j), Q2(j-1),
                q(j)];
41.          Q1(j) <-- t(1);
42.          q(j) <-- t(2);
43.       end

44. if e(k) = 1 then
45. begin
       comment Multiply Q by M;
46.    for i <-- 1 until u do

47.    begin
48.       for j <-- 1 until u do
49.          begin
50.             A(j) <-- 0;
51.             b(j) <-- 0
52.          end;

53.    if q(i) = 1 then
54.       begin
55.          for j <-- 1 until i do
                A(i+j-1) <-- m(j)
56.       end;
```

```
57.      if m(i) = 1 then
58.         begin
59.            if i > 1 then
60.               for j <-- 1 until i-1 do
                           B(i+j-1) <-- q(j)
61.         end;

62.      for j <-- 2n-1 step -1 until 1 do
63.         begin
64.            t(1) <-- S23[A(j), B(j),
                   C1(j-1), C2(j-2), s(j)];
65.            t(2) <-- S45[A(j), B(j),
                   C1(j-1), C2(j-2), s(j)];
66.            t(3) <-- S135[A(j), B(j),
                   C1(j-1), C2(j-2), s(j)];
67.            C1(j) <-- t(1);
68.            C2(j) <-- t(2);
69.            s(j) <-- t(3);
70.         end;

71.      for j <-- 1 until n do
                Q2(j) <-- 0;
72.      if s(i) = 1 then
            for j <-- 1 until n do
                        Q2(j) <-- ri(j);

73.      for j <-- n-1 step -1 until 1 do
74.         begin
75.            t(1) <-- S13[Q1(j), Q2(j-1),
                           q(j)];
76.            t(2) <-- S23[Q1(j), Q2(j-1),
                           q(j)];
77.            Q1(j) <-- t(1);
78.            q(j) <-- t(2)
79.         end
80.      end
81.   end;
82.   [q(n)----q(1)] <-- rem[Q, t]

   end.
```