EXPONENTIATING FASTER WITH ADDITION CHAINS

Y. Yacobi
Bellcore, 445 South St.
Morristown, NJ 07962
yacov@bellcore.com

The similarity between fast computations with huge numbers and data compression is investigated. For example, in data-compression, frequent messages are assigned short codes, while in fast computations we store and reuse the results of frequent computations. In compression we sometimes send the difference of consecutive messages, if its usually small (Δ modulation), while in fast computation we compute $x^{n+\Delta}=x^n \cdot x^\Delta$, if x^n is already known, and $\Delta \ll n$.

We demonstrate the similarity by applying a modification of the Lempel-Ziv data compression algorithm to fast exponentiation, to result runtime which is comparable to the best known methods, in the practical range (500 –1000 bits), for random exponents. For compressible exponents we gain on the average the compression ratio. This may be applicable to Diffie-Hellman like cryptographic systems. It is an open question whether compressible exponents are safe.

1. Introduction

The similarity between fast computations with huge numbers and data compression is investigated. For example, in data-compression, frequent messages are assigned short codes, while in fast computations we store and reuse the results of frequent computations. In compression we sometimes send the difference of consecutive messages, if its usually small (Δ modulation), while in fast computation we compute $x^{n+\Delta}=x^n \cdot x^\Delta$, if x^n is already known, and $\Delta \ll n$.

We demonstrate the similarity by applying a modification of the Lempel-Ziv data compression algorithm to fast exponentiation, to result runtime which is comparable to the best known methods, in the practical range (500 –1000 bits), for random exponents [Q]. For compressible exponents we gain on the average the compression ratio. This may be applicable to Diffie-Hellman like cryptographic systems. It is an open question whether compressible exponents are safe.

I.B. Damgard (Ed.): Advances in Cryptology - EUROCRYPT '90, LNCS 473, pp. 222-229, 1991.

Suppose we are required to compute x^n. We view n as a binary string $<n_L,...n_1,n_0>$, i.e. $n=\sum_{i=0}^{L} n_i \cdot 2^i$. When exponentiating using the binary method [K] there are two kinds of operations, squaring and multiplications. For example, if we want to raise x^{1101} (the exponent is written here in binary) we can compute $x^{1\cdot 2^3}\cdot x^{1\cdot 2^2}\cdot x^{0\cdot 2^1}\cdot x^{1\cdot 2^0}=((((x)^2\cdot x)^2)\cdot 1)^2)\cdot x$. In this example we had three squarings and two multiplications. This is a special case of *addition chain*. In general, an addition chain for n ([K] page 444) is a sequence of integers $1=a_0,a_1,\cdots,a_r=n$ with the property that $a_i=a_j+a_k$ for some k and j, $k\leq j<i$, for all $i=1,2,..r$. Binary addition chains as seen in the above example, determine an upper bound on the number of operations (squarings+multiplications) needed for the exponentiation, namely, $l(n)\leq L+v(n)-1$. Here $l(n)$ is the length of the addition chain, which equals the number of operations, and $v(n)$ is the Hamming weight of n, i.e. the number of 1's in the binary representation of n. Erdös [E] proved that for minimum length addition chains,

$$l(n)=L+L/log(L)+o(L/log(L))$$

There are some heuristics for finding addition chains [BC], but generally finding the minimum length addition chain is an NP-hard problem [D], and therefore all practical methods which work for arbitrary exponents, or very long exponents, avoid it, and use a longer addition chain. For our method, we have on the average

$$l(n)=L+(log(L)-loglog(L))/2+1.5\cdot L/log(L)$$

The first two expressions are the number of squarings, and the last expression is the number multiplications. The m-ary addition chain method ([K] page 444, [CL] pp. 110-111) runs in time $l(n)=L+m/2+L/m+2^{m-1}$ (the first two terms are squarings, the last two terms are multiplications), where m is an optimization parameter. Asymptotically it is optimal (choose $m=clog(L)$, such that $c^{-1}=1+\epsilon$, for ϵ as small as you wish). However, in the practical range the average running times of this method and of the new method are comparable.

Both methods use about L squarings, but squarings are usually cheaper than multiplications. This is true not only for the naive schoolboy multiplication method, where squaring is twice as fast as multiplying, but also for all FFT based multiplication methods. To begin with, when squaring one needs only one FFT transformation (and one inverse), while for multiplication two transformations are needed (and one inverse). Also, in the transformed domain multiplication is pairwise, which means that squaring is broken into many small squarings. The size of squaring table is the square root of the size of multiplication table. This means that for the same space complexity for some machines we can gain another factor of 2 for squaring. Altogether squaring may be 3 times faster than multiplying for FFT algorithms, in some

machines. In some other applications the situation may be reversed. For example, for elliptic curve arithmetic it seems that squaring is more expensive than multiplication.

We can use a very similar method to multiply large numbers.

2. The Method

We first explain the m-ary method, and introduce few of our computational improvements. This may help following the description of the new algorithm.

2.1 The m-ary Method

Let m be an optimization parameter. Given base X, compute the set S of exponents $\{X^{e_i}\}$, for all odd e_i of length m bits. To compute X^n, write the binary representation of n as follows. $n=e_k 0^{Z_k},e_{k-1}0^{Z_{k-1}},\ldots,e_1 0^{Z_1}$, Here 0^{Z_i} means a run of Z_i zeroes. For all i $Z_i \geq 0$. The most significant segment, e_k, may be shorter than m bits. The computation of X^n is accomplished by

$$X^n=(\ldots((X^{e_k})^{2^{Z_k+|e_{k-1}|}}\cdot X^{e_{k-1}})^{2^{Z_{k-1}+|e_{k-2}|}}\cdot\ldots\cdot X^{e_1})^{2^{Z_1}}.$$

To compute the set S we can use the following ideas, which we later use in our method. Create a complete half binary tree, such that the root has only one son, with arc labeled "1", and all the rest of the nodes have two sons each. The tree is of depth m, and each node is associated with the path which leads from the root to that node in the natural way. Hence each leaf corresponds to one of the e_i's. Our aim is to store in each leaf the value of "its" X^{e_i}. To do so we start from the root and store in each node the value of its $X^{e'_i}$ ($e'_i<m$ for non leaf nodes). Suppose that we completed this task for level (depth) i. The computation of level $i+1$ is done as follows. Rights sons (arcs labeled 0) are copied from their fathers (a most significant new bit, which equals 0 does not change the result). A left son is computed by $X^{2^i}\cdot father$. So each left son requires one multiplication, and the whole level requires one squaring, since $X^{2^i}=(X^{2^{i-1}})^2$.

Altogether we get m squarings, and 2^{m-1} multiplications, to create the tree. Using the tree requires L/m multiplications and $L-m/2$ squarings (on the average $|e_k|=m/2$).

The new method is very similar to the m-ary method. The difference is that we compute in the above tree only those e_i's which actually appear in the exponent, and we don't truncate the tree to a prespecified depth.

2.2 The New Method

The intuition behind the method is that if some patterns in the binary representation of the exponent reappear, then we don't have to recompute their contribution, if we already stored it. The method is similar to the m-ary method, with two differences. First, in the m-ary method we may do some precomputations, which are not used later, while in the new method we

precompute exactly those subexponents that we are about to use. Second, we do not bound our precomputations to a fixed predetermined length. Rather our algorithm is adaptive to the particular exponent at hand. A similar intuition led to the Lempel-Ziv [LZ] compression algorithm.

We parse n right to left (i.e., from low order to high order bits) the way Lempel and Ziv [LZ] do in their compression method, i.e. we create a binary "compression" tree, where the path e_i from the root to node i is a segment of the exponent, and node i contains the partial result x^{e_i}. The structure of the tree makes it easy to add a new leaf, given its parent. We describe first a subroutine that creates the binary tree, and then the main program, which uses the tree to compute x^n.

The exponent is called here *seq*, and is treated sometimes as an integer, and sometimes as a sequence of bits. Each segment e_i of the exponent is an odd number of length $L_i \geq 1$, and is preceded (to the right) by $Z_i \geq 0$ zeroes.

subroutine **build-tree**
begin

— **Init:** Store $e_0=1$; $Z_0 \leftarrow L_{-1} \leftarrow 0$ in node #0, and set $L_0 \leftarrow 0$. put parse # 0 right of *seq*; $i \leftarrow 1$;

— **While** there are more symbols in *seq* **do**

— **begin**

— $L_i \leftarrow 0$; $Z_i \leftarrow 0$.

— Scan *seq* from the $i-1_{th}$ parse to the left.

— While the new symbol (ns) $=0$ $Z_i \leftarrow Z_i+1$.

— (ns=1) $L_i \leftarrow L_i+1$; start following the path of the tree defined by *seq* , incrementing L_i, until a leaf is reached.

— Scan one more symbol (*ns*), add a new arc from the last visited leaf, label it *ns*, add a new leaf and a new parse, and label them i. e_i is the integer represented by the segment which starts at parse $i-1$ and ends at parse i, not including leading zeroes to the right.

— Compute x^{e_i}, and store it together with $L_{i-1}=|e_{i-1}|, e_{i-1}$, and Z_i in leaf i.

— $i \leftarrow i+1$;

— **end;**
end.

Let $k=max(i)$. As in the m-ary method, the exponent looks as follows. $n=e_k 0^{z_k} e_{k-1} 0^{z_{k-1}} \ldots e_1 0^{z_1}$

Algorithm 1:

begin

— Call **build-tree** subroutine.

— $x^n \leftarrow ((\ldots((x^{e_k})^{2^{z_k+L_{k-1}}} \cdot x^{e_{k-1}})^{2^{z_{k-1}+L_{k-2}}} \cdot x^{e_{k-2}} \ldots)^{2^{z_2+L_1}} \cdot x^{e_1})^{2^{z_1}}$

end.

Remark: e_{i-1} is stored in "node e_i" as a back pointer, later used by algorithm 1 to determine the right sequencing of the partial results.

The method is general, and applies to any multiplicative group.
A numerical example appears in the appendix.

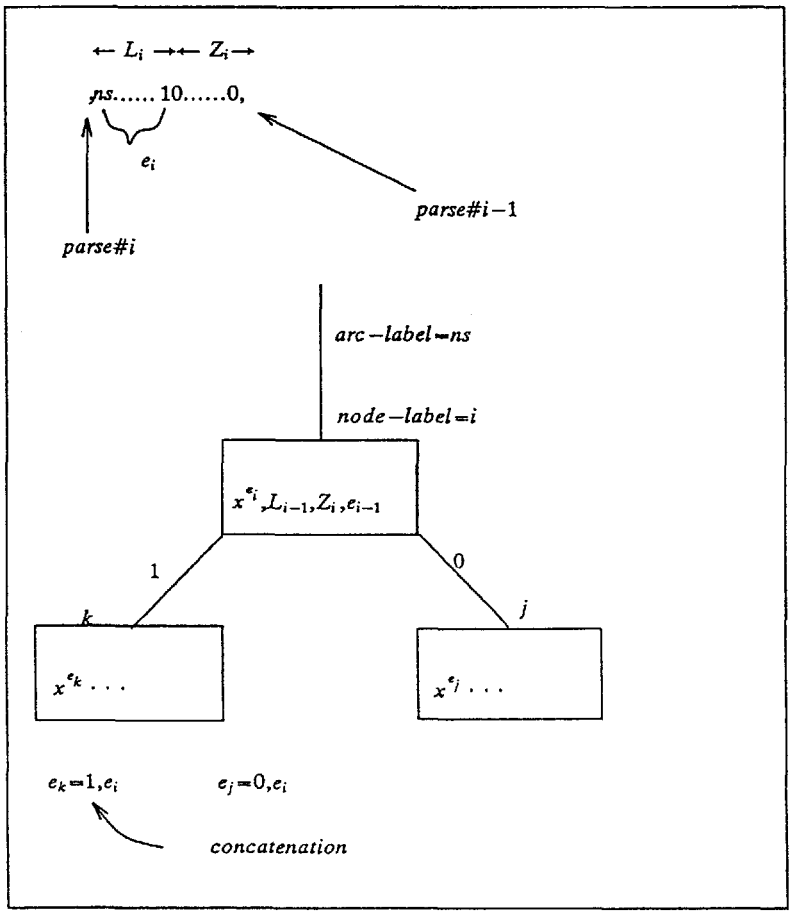

fig 1. The general structure of the tree

3. Complexity

3.1 Time complexity

We first estimate the tree size for a random exponent. Then we use it to count the number of operations needed in the entire computation. On the average we expect to get balanced trees (applying the Lempel-Ziv algorithm to random sequences result balanced trees).

Let h be the depth of the tree, k the number of nodes in the tree (maybe one less than the k of line 2 algorithm 1, if $e_k=e_j$ for some $j<k$), and, as before, $L=log(n)$ the length of the exponent.

Lemma 1: on the average $h=log(k)$, and $L=k\cdot log(k)$.

Proof: Our tree is half a binary tree (and we assume that on the average that half is complete. This assumption is true for random exponent). Hence at depth i we have 2^{i-1} nodes. The path to each node at depth i covers on the average (including leading zeroes) a segment of length $i+1$ of the exponent (because the probability of a run of exactly j zeroes in a random string is $2^{-(j+1)}$, hence the expected length of runs of zeroes is $\sum_{j=1}^{\infty} j\cdot 2^{-(j+1)}=1$). So, $L=\sum_{i=1}^{h}(i+1)\cdot 2^{i-1}$, and by simple induction on h we get $L=h\cdot 2^h$.

Since a full binary tree of depth h has $2^{h+1}-1$ nodes, our tree has $k=2^h$ nodes. So, $h=log_2 k$, and $L=k\cdot \log_2(k)$.

Q.E.D.

Corollary: $k=L/log(L)+o(L/log(L))$. (i.e. $\lim_{L\to\infty}\frac{k}{L/log(L)}=1$.)

Lemma 2: The average time complexity of Algorithm 1 is

$l(n)=L-(log(L)-loglog(L))/2+1.5\cdot(L/log(L)+o(L/log(L))$.

Proof: There are four types of operations involved in the above computation. Squarings and multiplications during the construction of the tree, denoted TS and TM respectively, and exponentiations and multiplications which appear in line (2) of algorithm 1., denoted S and M, respectively. $\#S=L-h/2$, this follows from the fact that the last segment e_k is of expected length $h/2$. $h=log(k)$, hence by the corollary of Lemma 1, $h\sim log(L)-loglog(L)$.

#M=#nodes in the tree= k.

As for the cost of building the tree, note that for each depth i, we have to compute once x^{2^i}, which is done using one squaring, then we apply it on the average to half the nodes using $x^{(1,z)}=x^{2^{|z|}}\cdot x^z$. Here $(1,z)$ denotes 1 concatenated to the left of z. The other half of the nodes we get for free, since $x^{(0,z)}=x^z$. So, $\#TM=\frac{1}{2}\cdot k$ on the average $(=v(n))$, and $\#TS=h$. We

conclude that the cost of our method is,

$l(n)=\#S+\#M+\#TM+\#TS=(L-h/2)+k+k/2+h=l+h/2+1.5\cdot k=$
$L+(log(L)-loglog(L))/2+1.5\cdot(L/log(L)+o(L/log(L)).$

Q.E.D.

Asymptotically the m-ary method is superior. It achieves the lower bound. However, for a practical exponent size like 512, or 1024 the new method is comparable for random exponents, and superior for compressible exponents. On the average the gain in the number of multiplications is the compression ratio. However, for individual compressible sequences it may differ from that ratio. For compressible exponents the tree is always skewed. If it is skewed to the left the gain is smaller than the compression ratio, while if it is skewed to the right the gain is greater than the compression ratio.

Open problem: What is the complexity of the discrete log problem, given that the exponent is compressible (with a given compression ratio)?

If the fact that the exponent is compressible does not affect the complexity of discrete-log then more efficient Diffie-Hellman like key distribution systems can be designed.

3.2 Space complexity

The tree has $k=L/log(L)$ nodes. If the exponentiation is modular, as is the case in modern cryptography, then each node stores $L=log(n)$ bits. All together we have space complexity $L^2/log(L)$. (We assumed here that the modulus and the exponent are of the same size.)

Acknowledgments

I would like to thank Shimon Even, Rich Graveman, Stuart Haber, and Arjen Lenstra for many helpful discussions.

4. References

[BC] J. Bos, M. Coster: "Addition Chain Heuristics", *Crypto'89.*

[CL] H. Cohen, and A.K. Lenstra: "Implementation of a New Primality Test", *Math. of Computation* , Vol. 48, No. 177, Jan. 1987, pp.103-121.

[D] Downey, Leony, and Sethi: "Computing Sequences With Addition Chains", *SIAM Jour, Comput.* 3, (1981), pp.638-696.

[E] P. Erdös: "Remarks on Number Theory III, on Addition Chains", *Acta Arith. VI* (1960), pp.77-81.

[GKP] Graham, Knuth and Patashnik: " Concrete Mathematics, a Foundation for Computer Science", *Addison Wesley*, 1988.

[LZ] Lempel and Ziv: "Compression of Individual Sequences via Variable Rate Coding" *IEEE Trans of Inf. Theory* , IT-24 No. 5 (September 1978) pp.530-536.

[K] Knuth: "The Art of Computer Programming, Vol. 2, Seminumerical Algorithms", *Addison-Wesley*, 1980, pp.441-462.

[Q] J.J. Quisquater: (A talk on m-ary exponentiation at the Rump session of Eurocrypt'90)

[S] Schönhage: "A Lower Bound on The Length of Addition Chains", *Theoretical Computer Science* Vol. 1 (1975), pp.229-242.

5. Appendix

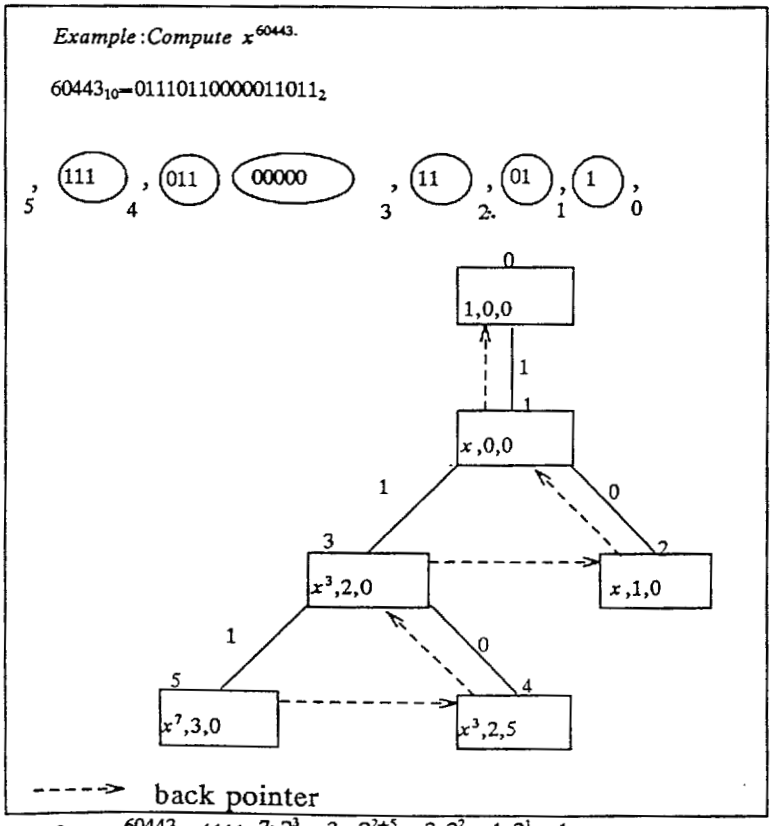

fig. 2: $x^{60443}=((((x^7)^{2^3}\cdot x^3\cdot)^{2^{2+5}}\cdot x^3)^{2^2}\cdot x^1)^{2^1}\cdot x^1$