LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-91

FACTORING NUMBERS IN O (LOG N) ARITHMETIC STEPS

ADI SHAMIR

NOVEMBER 1977

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-91

FACTORING NUMBERS IN O(log n) ARITHMETIC STEPS

Adi Shamir

November 1977

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science
(formerly Project MAC)

Cambridge Massachusetts 02139

FACTORING NUMBERS IN O(log n) ARITHMETIC STEPS

Adi Shamir

Department of Mathematics

Massachusetts Institute of Technology

Cambridge, Massachusetts 02139

Abstract

In this paper we show that a non-trivial factor of a composite number n can be found by performing arithmetic steps in a number proportional to the number of bits in n, and thus there are extremely short straight-line factoring programs. However, this theoretical result does not imply that natural numbers can be factored in polynomial time in the Turing-Machine model of complexity, since the numbers operated on can be as big as 2^{cn^2}, thus requiring exponentially many bit operations.

This report was prepared with the support of the Office of Naval Research Grant No. N000-14-76-C-0366 and National Science Foundation Grant No. 77-19754MCS.

KEY WORDS: Arithmetic Complexity - Factoring Algorithms - Prime Numbers

I. Introduction.

The problems of primality checking and factoring of natural numbers have been given much attention in the last four centuries. The development of efficient algorithms for these problems is not only theoretically interesting, but can also have important practical consequences (for example, in the field of cryptography -- see Rivest, Shamir and Adleman [1]). While it is relatively easy to determine that a given number n is composite, actually finding its factors seems to be a much harder problem. To date, all the algorithms developed for this purpose run in time which is non-polynomial in the length of the binary representation of n (e.g., Pollard [2]).

In this paper, we consider the inherent difficulty of the factoring problem from the point of view of another natural measure of complexity, namely the number of arithmetic steps (addition, subtraction, multiplication and integer division) needed in order to solve the problem. We develop an algorithm which finds a non-trivial factor of a composite number n in $O(\log n)$ arithmetic steps, and we conjecture that it is optimal. This result does not imply that natural numbers can be factored in polynomial time, since our measure of complexity ignores the size of the numbers involved. The algorithm presented in this paper is thus mainly of theoretical interest, showing that surprisingly short straight-line computer programs can factor natural numbers.

II. The Model.

We consider a very simple model of a computer, consisting of registers and a CPU. There is a fixed number of registers (R_0 through R_s), each one of which can hold a single integer of unbounded size. The CPU contains a program, which is a finite sequence of labelled instructions of the following types:

(i) L: $R_m \leftarrow R_i * R_j$, where L is a label and * is +, −, · or ÷ (a ÷ b is the largest integer not exceeding the rational quotient a/b).

(ii) L: if $R_m = 0$ then go to L_1 else go to L_2.

(iii) L: Print (R_m).

(iv) L: Halt.

The arithmetic complexity of a program is the function a(n) giving for each input n the number of arithmetic instructions of type (i) performed until the computer halts; if the computer loops forever, $a(n) = \infty$.

A factoring algorithm is a program which finds and prints out for any composite natural number n (initially placed in R_0) a non-trivial factor $1 < f < n$ that divides n; if n is a prime, the program halts without printing anything. Such a program can serve as the nucleus of more complicated types of factoring algorithms. For example, in order to find the complete prime factorization of a given number n, it suffices to find such a factor f (which is not necessarily a prime!) and then to recursively find the complete prime factorizations of f and of n ÷ f.

III. A Fast Method for Computing Factorials.

Our factorization algorithm is based on a method for computing the factorial function n! which has a low arithmetic complexity. In [3], Pratt uses a clever partition of the n factors in n! in order to reduce the obvious O(n) algorithm to an almost $O(\sqrt{n})$ algorithm, but this complexity is still too high. An O(log n) algorithm is implicit in Davis' paper [4] on the unsolvability of Hilbert's 10th problem, but a somewhat differently structured method is needed in order to obtain the final O(log n) factoring algorithm. The method we describe has the additional advantage of using smaller intermediate numbers, thus reducing the space requirements from $n^2 \log n$ bits to n^2 bits (note that n log n bits are necessary just in order to hold the final value n! in binary representation).

Let n be an even natural number. Then by definition

$$\binom{n}{\frac{n}{2}} = \frac{n!}{\left(\frac{n}{2}!\right)^2}$$

from which we get the recursive relation

$$(1) \qquad n! = \binom{n}{\frac{n}{2}} \left(\frac{n}{2}!\right)^2$$

If n is an odd natural number, we use the identity

$$(2) \qquad n! = n \cdot (n-1)!$$

in which n-1 is even and (1) is applicable. Thus for any n, the calculation of n! can be quickly reduced to that of calculating $(n \div 2)!$. What remains to be done is to find an efficient method for calculating the log n terms of the form $\binom{2k}{k}$ (with

$k = n \div 2^i$ for $1 \le i \le \log n$) obtained when the recursive definition is unfolded.

Consider now the binomial identity:

$$(3) \qquad (2^\ell + 1)^{2k} = \sum_{j=0}^{2k} \binom{2k}{j} 2^{\ell \cdot j}$$

If this number is written in binary representation, then each summand $\binom{2k}{j} 2^{\ell \cdot j}$ is just the binary representation of $\binom{2k}{j}$ shifted left $\ell \cdot j$ bits. By making ℓ big enough, each $\binom{2k}{j}$ can be shifted to a distinct block of ℓ bits in the binary representation of $(2^\ell + 1)^{2k}$, and thus can be easily isolated.*

The minimum usable value of the block size is just the number of bits occupied by the largest term of the form $\binom{2k}{j}$. Since

$$\sum_{j=0}^{2k} \binom{2k}{j} = 2^{2k}$$

any block size of $\ell \ge 2k$ bits can be used. In particular, the value of $\binom{2k}{k}$ can be found by isolating the middle block of 2k bits in

$$(4) \qquad (2^{2k} + 1)^{2k}$$

In order to find the arithmetic complexity of this process, we note that the $2k^{th}$ power of a number can be calculated in O(log k) arithmetic steps by the well known method of successive

---

* To isolate the lower m bits in a register R, calculate $R' \leftarrow R - (R \div 2^m) \cdot 2^m$; to isolate bits $m_1 + 1$ through m_2 in R, subtract the lower m_1 bits in R from the lower m_2 bits in R, and divide the result by 2^{m_1}.

squarings.* Using this method twice in succession, (4) can also be calculated in O(log k) arithmetic steps. The calculation of the powers of 2 used in isolating the middle block in (4) requires a similar number of steps, and thus any term of the form $\binom{2k}{k}$ can be calculated in O(log k) steps.

In order to calculate n! in our method, we have to find the values of $\binom{2k}{k}$ for all k of the form $k = n \div 2^i$, $1 \le i \le \log n$. The discussion above shows how to do it in $O(\log^2 n)$ arithmetic steps, but a simple trick can reduce it to O(log n).

Since all the values of k satisfy $2k \le n$, we can replace the variable block size $\ell = 2k$ by the uniform block size $\ell = n$, and thus $\binom{2k}{k}$ can also be isolated as the middle block of n bits in

$$(5) \qquad (2^n + 1)^{2k}$$

However, since each k is $n \div 2^i$ for some i, all the log n numbers of the form (5) can be obtained free as the intermediate stages in the calculation of the single number

$$(6) \qquad (2^n + 1)^n$$

by the successive squarings method! A similar trick can be used in the computation of the powers of 2 used in order to isolate the middle blocks, and thus the arithmetic complexity of our

---

* Using the recursive relation

$$a^x = \begin{cases} (a^{x/2})^2 & \text{if } x \text{ is even} \\ a \cdot a^{x-1} & \text{if } x \text{ is odd} \end{cases}$$

algorithm for computing n! is just O(log n).

A final remark about space requirements. Due to the special form of the main recursive definition (equations (1) and (2)), it is easy to translate it into a simple iterative loop in which the register which eventually contains n! is either squared or multiplied by an auxiliary register. This eliminates the need for a recursion stack. Furthermore, no temporary storage is needed for the log n numbers $\binom{2k}{k}$, since they are used in the same natural order in which they are produced when (6) is evaluated. The entire algorithm can thus be implemented with a (small) fixed number of registers, without using any data-compacting techniques. The biggest number stored in these registers is $(2^n + 1)^n$ itself, which needs $O(n^2)$ bits in its binary representation.

IV. Factoring Natural Numbers.

Once we have an O(log n) algorithm for computing factorials, an $O(\log^2 n)$ factoring algorithm is trivial to construct. If $i < n$ are two natural numbers, then the greatest common divisor (gcd) of n and i! is greater than 1 iff n has a factor j satisfying $1<j\leq i$. Consequently, in order to find the smallest prime factor of n, we can perform a binary search on the values of i, using the predicate $\gcd(n,i!) \stackrel{?}{=} 1$ as the criterion for increasing i.

For any given value of i, i! can be computed in O(log n) arithmetic steps. Even though i! can be an enormous number,

the calculation of gcd(n,i!) requires only min(log n,log i!) $\leq$ log n arithmetic steps (a thorough discussion of gcd algorithms appears in Knuth [5]). The number of gcd calculations in the binary search is again bounded by log n, and thus the smallest (prime) factor of n can be found in $O(\log^2 n)$ arithmetic steps.

In order to construct an improved O(log n) factoring algorithm, we have to replace the relatively expensive gcd(n,i!) operation by the much cheaper operation of reduction modulo n (i!(mod n) can be calculated by just three arithmetic operations as i! - (i! ÷ n)·n).

Let i_o be the least natural number in the range $1 \leq i \leq n$ which satisfies the predicate $i! \equiv 0 \pmod{n}$ (since n divides n!, there is at least one such i), and let f be $\gcd(n,i_o)$. By a variant of Wilson's theorem (see [6]), a natural number $n > 4$ is composite iff $(n - 1)! \equiv 0 \pmod{n}$, and thus $f \leq i_o < n$ whenever $n > 4$ is composite. On the other hand, f cannot be 1, since by assumption n does not divide $(i_o - 1)!$ but does divide $i_o! = i_o \cdot (i_o - 1)!$. Consequently, $1 < f < n$ is a non-trivial factor of n whenever $n > 4$ is composite, and $f = n$ whenever $n > 4$ is a prime. Note that f is not necessarily the smallest factor (or even a prime factor) of n, as demonstrated in the case $n = 18$, $i_o = 6$, $f = 6$.

The factorization problem has thus been reduced to the problem of finding i_o, since f can be calculated from n and i_o in at most log n additional steps. The predicate $i! \stackrel{?}{\equiv} 0 \pmod{n}$

has the useful property that $i_1! \equiv 0 \pmod{n}$ and $i_2 > i_1$ imply that $i_2! \equiv 0 \pmod{n}$, which makes a fast binary search possible. However, so far the algorithm's arithmetic complexity is still $O(\log^2 n)$, since log n factorials (each one of which requires $O(\log n)$ steps) must be evaluated. We shall now make use of our factorial algorithm's special structure in order to combine many of the subcomputations involved. Throughout the discussion, n is assumed to be greater than 4.

Rather than find i_o directly, we first find the interval between two successive powers of 2 in which it is contained. If 2^{j_1} is the smallest power of 2 which exceeds n, then calculating $2^{j_1}!$ (in $O(\log n)$ arithmetic steps) in our method gives us the factorials of all the smaller powers of 2 for free. By reducing each one of these factorials mod n (using 3 arithmetic steps), we can find in just $O(\log n)$ arithmetic steps the (uniquely defined) power j_o such that $2^{j_o}! \not\equiv 0 \pmod{n}$ and $2^{j_o+1}! \equiv 0 \pmod{n}$. By the definition of i_o, $2^{j_o} < i_o \leq 2^{j_o+1}$.

If we try to locate i_o by binarily searching in the interval $(2^{j_o}, 2^{j_o+1}]$, we run into a problem: the successive factorials we have to calculate are no longer related in an obvious way, and calculating each one separately leaves us with an $O(\log^2 n)$ algorithm. We bypass this difficulty by spending log n steps at this stage on the evaluation of $f = \gcd(n, 2^{j_o}!)$ and then distinguishing between two cases:

(i) If $f \neq 1$, then f is a non-trivial factor of n ($f = n$ contradicts the assumption that $2^{j_o}! \not\equiv 0 \pmod{n}$), and we

do not have to proceed.

(ii) If f = 1, then the evaluation of the predicate $i! \stackrel{?}{\equiv} 0 \pmod{n}$ for i in the range of interest can be simplifed considerably, as described below.

Let i be an even integer, $2^{j_o} < i \leq 2^{j_o+1}$. By (1),

$$i! = \binom{i}{\frac{i}{2}} \left(\frac{i}{2}!\right)^2,$$

and $\frac{i}{2} \leq 2^{j_o}$. By assumption, $\gcd(n, 2^{j_o}!) = 1$, and thus also $\gcd(n, \frac{i}{2}!) = 1$ and $\gcd\left(n, \left(\frac{i}{2}!\right)^2\right) = 1$. Consequently, n divides i! if and only if n divides $\binom{i}{\frac{i}{2}}$, and thus the predicate $i! \stackrel{?}{\equiv} 0 \pmod{n}$ can be replaced by the easier predicate $\binom{i}{\frac{i}{2}} \stackrel{?}{\equiv} 0 \pmod{n}$ for any even i in the interval $(2^{j_o}, 2^{j_o+1}]$. Odd integers in this interval are treated similarly, by using the identity $i! = i \cdot (i-1)!$ first.

When the interval $(2^{j_o}, 2^{j_o+1}]$ is binarily searched for the value of i_o, at most log n numbers of the form $\binom{i}{\frac{i}{2}}$ have to be calculated, and then reduced modulo n. What remains to be done is to show that all these numbers can be calculated in O(log n) arithmetic steps.

As described in Section III, each number $\binom{i}{\frac{i}{2}}$ can be isolated as the middle block of n bits in $(2^n+1)^i$. We can precalculate (in log n steps) these i^{th} powers for all the values of i which are powers of 2, and store them in successive blocks of bits in one of the registers (with the biggest power occupying the lowest order bits). The value of $(2^n+1)^i$ for any

other i can now be calculated as a partial product of these precalculated numbers, using the binary representation of i as a guide (for example, $(2^n+1)^{10} = (2^n+1)^8 \cdot (2^n+1)^2$).

The binary search for the value of i_o in the interval $(2^{j_o}, 2^{j_o+1}]$ can be easily arranged in such a way that a new trial value i_{k+1} differs from the previous trial value i_k by exactly 2^{j_o-k} ($k = 1,2,\ldots,j_o$), with $i_1 = 2^{j_o+1}$. In this case, $(2^n+1)^{i_{k+1}}$ can be obtained from $(2^n+1)^{i_k}$ by multiplying or dividing the latter by the precomputed value $(2^n+1)^{2^{j_o-k}}$, which is a single operation. Since we use the precomputed powers in a simple high-to-low order, we can right shift the storage register after each use in such a way that the next power to be used is readily available in the low order bits of the register.

An improved method, which does not use a storage register, is to replace the "binary search" by a "Fibonacci search"; note that numbers of the form $(2^n+1)^{F_i}$ can be quickly calculated in either an ascending or a descending order by multiplying or dividing a pair of successive elements in this sequence. While it is possible to squeeze the log n precomputed numbers in the simple method into an $O(n^2)$ bit storage register (by using variable-size blocks and a tricky retrieval scheme), the improved method gives a straightforward $O(n^2)$ bound on the memory requirements of our factoring algorithm.

This complete the proof that a non-trivial factor of a number n (if it exists) can be found by performing at most

O(log n) arithmetic operations. An interesting open problem is to determine whether O(log n) is also a lower bound on the arithmetic complexity of factoring algorithms. We conjecture that it is.

Acknowledgements

The author wishes to thank Vaughan Pratt for posing the question answered in this note, Albert Meyer and Ron Rivest for supplying valuable comments, and Len Adleman for pointing out the implicit existence of another factorial algorithm in Davis' paper.

Bibliography

[1] A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, by Ronald Rivest, Adi Shamir and Len Adleman, Technical report MIT/LCS/TM-82, April 1977.

[2] Theorems on Factorization and Primality Testing, by J.M. Pollard, Proc. Camb. Phil. Soc., 1974, pp. 521-528.

[3] The Competence/Performance Dichotomy in Programming, by V. R. Pratt, Proceedings of the Fourth ACM Symposium on Principles of Programming Languages, 1977.

[4] Hilbert's Tenth Problem Is Unsolvable, by Martin Davis, American Mathematical Monthly, March 1973.

[5] The Art of Computer Programming, Volume 2, by D.E. Knuth, Addison Wesley, 1969.

[6] An Introduction to the Theory of Numbers, by G.H. Hardy and E.M. Wright (fourth edition), Oxford press, 1960.