6. Conclusion

The importance of simple intuitive semantic rules for language constructs should be respected. If we have no simple way to describe a semantic rule, then we had better face it by showing the odd rule explicitly with an explanation.

For the use of the principle of exclusion, we should be careful for the implicit inclusion of undesired alternatives. Thus for the new ISO Pascal standard, we propose to explicitly specify only the look-ahead I/O and the lazy I/O as alternative implementations. The user will understand this kind of approach as he knows the reason.

For the use of the principle of intersection, we should make sure that the resultant rule can be described by a new simple rule without a quantifier such as 'for all input files . . .'. Thus for the Ada specification, we propose to formally specify the compiler time 'no aliasing conditions' for parameters of array, record or private types. The run-time aliasing error should not be allowed since the user can hardly test all input files. If there is a chance of a run-time error, then the program cannot be released for use. So compiler time detection should be enforced. The error report from the compiler informs the user to consider efficient parameter passing problems for all array, record and private parameter types. If it is necessary to do the information copying for some parameters, then the user ought to do it explicitly.

Acknowledgement

I would like to thank Dr. Cynthia Brown for her patient reading and helpful advice on this paper.

CHINGMIN JIM LO
Computer Science Department
Indiana University
Bloomington
Indiana 47405
USA

References

1. D. M. Berry, Remarks on R.D. Tennent's language design methods based on semantic principles: Algol 68, a language designed using semantic principles. *Acta Informatica* **15** (1), 83–98 (1981).
2. A. M. Addyman, A draft proposal for Pascal. *SIGPLAN Notices* **15** (4), 1–66 (1980).
3. K. Jensen and N. Wirth, *PASCAL User Manual and Report, 2nd Edn,* Springer-Verlag (1976).
4. J. B. Saxe and A. Hisgen, Lazy evaluation of the file buffer for interactive I/O. *Pascal News* **13**, 93–94 (1978).
5. *Ada Reference Manual,* United States Dept. of Defense (July 1980).
6. H. Perkins, Lazy I/O is not the answer. *SIGPLAN Notices* **16**(4), 81–86 (1981).
7. K. Tai, Comments on parameter passing techniques in programming languages. *SIGPLAN Notices* **17**(2), 24–27 (1982).
8. *Formal Definition of the Ada Programming Language—Preliminary Version for Public Review,* Honeywell Inc. (1981).

Received June 1982

Factoring Medium-Sized Integers

The factoring of integers is an important problem, and one well-suited to computers, and many algorithms have been proposed for this. This paper compares various algorithms, and discusses the choice of parameters for the algorithms, based on experiments with numbers from 10^{13} to 10^{20}. We conclude with recommendations on the design of a factoring algorithm.

1. Introduction

For the purpose of factoring numbers, it is possible to classify the integers (at least roughly) by size:

(a) *small* integers, typically those less than 2^{32}, or some such machine-related bound (normally those that will fit in one or two words)
(b) *medium-sized* integers, which are larger than small integers, but less than about 10^{20} or 10^{25}
(c) *large* integers, which are larger than the previous classifications.

Throughout this paper, N will denote a number which it is intended to factor.

The small integers are easy to factor, since a search through all primes less than $\sqrt{N}$ is relatively economical (there are 6542 primes less than 2^{16}, for example), and, if space for this cannot be afforded, then the search should be for factors of 2, 3, 5, and those numbers congruent to 1, 7, 11, 13, 17, 19, 23 and 29 (mod 30). This will give 17479 possibilities up to 2^{16}, substantially more than a direct table.

When it comes to large integers, as defined above, the problem is, if not totally intractable, at least very difficult. Not only are large† amounts of computer time required, but special theory, adapted to the form of the number to be factored, is generally necessary (for some recent examples, consider Refs 1 and 2).

The aim of this note is to provide a guide to the intermediate field of medium-sized integers, where several algorithms exist (see Ref. 3 section 4.5.3 (pp. 369–398)) and can be run practically in a few seconds or minutes. We develop no new mathematics in this paper, but hope that the reader will find our recommendations useful if he wants to implement a factoring package for medium-sized integers.

2. The algorithms

Knuth[3] quotes the following algorithms:

1. Algorithm A (Factoring by division) on pp. 364–365
2. Algorithm B (Monte Carlo factorization) on p. 370 (see also Refs 4 and 5)
3. Algorithm C (Factoring by addition and subtraction, often known as Fermat's method) on p. 371
4. Algorithm D (Factoring with sieves) on p. 373
5. Algorithm P (Probabilistic primality test) on p. 379
6. Algorithm E (Factoring via continued fractions) on pp. 381–382 (see also Refs 6–8)

Other general-purpose algorithms include:

7. Draim's algorithm (see Ref. 9 or Ref. 10, pp. 32–35)
8. Deterministic primality test[11]
9. Schnorr's method.[12]

We can immediately remark that the probabilistic primality test is far faster than the deterministic one, as well as being much simpler to program. There therefore seems little point in using the deterministic test, and, as Ref. 3 points out, the difference in running times can be so great that the probability of undetected hardware error while running the deterministic test is greater than the probability of several applications of the probabilistic test all failing. The existence, and asymptotic complexity, of a deterministic test is, nevertheless, of theoretical interest.

Draim's algorithm is essentially a variant of the standard method of dividing by all the odd numbers, though it can be adapted so as to divide by those not divisible by 3 or 5, or even to divide by just the primes. It is more complicated to program, but has the advantage that the dividend decreases, at least for a while, rather than remaining constant. The second author has used this method to advantage on machines without a hardware division instruction, when decreasing the dividend can bring substantial gains.

Schnorr's method is, asymptotically, the fastest method (known to the authors) of factoring large numbers. We do not have a complete implementation of it, but do not expect it to be competitive with the methods Knuth quotes in the medium-sized range.

We have already dealt with Knuth's Algorithm A, and we note that Algorithm C is only useful when the number has two factors which are very close to its square root. In general, this is unlikely to happen, and Algorithm C is not recommended, since its asymptotic time is $v - \sqrt{N}$, where v is the larger factor of N that this algorithm finds. There are several variants on this algorithm: one, due to Lehman[13] finds factors whose ratio, instead of being close to one, is close to simple rationals, and this has a running time $O(N^{1/3})$.

Knuth's algorithm D is a variant on Fermat's method, which uses precomputed tables of residues to decide if $x^2 - N$ can be a perfect square. This can speed up the operation quite significantly (and special sieving hardware, such as D. H. Lehmer[6] uses, can give very

† We note that Ref. 1 talks of 'a couple of days CPU time', and Ref. 2 refers to a program that ran for over 18 months.

substantial performance gains). Even on a conventional machine, this method has a variety of optimizations that can be applied to it:

1. We can avoid many square-root operations by keeping tables of possible endings of square-roots (we chose to work modulo 10 000).
2. We can build in special checks for the values of N modulo 8 and modulo 9.
3. We can work modulo p_1p_2, rather than modulo p_1 and p_2 separately, thus trading store† for time.

One decision that has to be made is the number of moduli to be used: our experience was that numbers around 10^n factored most efficiently with max $(4,n/2)$ moduli.‡ However, it is our experience that this algorithm is only faster than the methods to be described below in special cases.

3. The main contenders

This leaves us with two algorithms, both of which Knuth mentions, viz. his algorithms B and E.

3.1. Monte Carlo

Algorithm B has a running time (in terms of arithmetic operations) of $O(\sqrt{p_{t-1}})$, where p_{t-1} is the second-largest prime factor of N. This is, of course, bounded by $N^{1/4}$, but is often much lower (indeed, $<N^{0.106}$ half the time, see Ref. 3, p. 368). Thus this method is best when the factors are not evenly balanced. However, there is a large element of randomness in this algorithm: we found the factor 2 962 963 of 10 378 154 203 801 in 13.7 s, but the factor 1 375 951, approximately half the size, of 15 000 000 000 001 was found in 2.8 s.

This algorithm can fail, in the sense that, given a number which we know not to be prime, it will still terminate saying 'no factors found'. If this happens, we can either switch to the method of the next section or use the same algorithm with a different generating function, e.g. x^2+2 instead of x^2+1. This failure appears to be extremely rare, and indeed we have never encountered such a number. The improvement of Ref. 5 requires knowing something about $N-1$, and is not truly a general method.

3.2. Continued fraction

This algorithm, originally suggested by Lehmer[6], is probably the most difficult to implement of the algorithms we are suggesting, but also the most efficient. In fact it factors a number kN, where k is small, rather than N itself. It also depends on the number of primes one wishes to use in the linear equation phase. It appears likely (Ref. 3, p. 383), that the running time is subpolynomial, in fact $O(N^{2\sqrt{(\ln\ln N)/(\ln N)}})$.

The choice of k is interesting, since it both determines the set of admissible primes (and we would like a large number of small admissible primes in our factor base), and also determines the period of $\sqrt{kN}$, which must not be too small. Ref. 3 gives a function that must be maximized for the optimal choice of k, and Ref. 7 reports that, for factoring $2^{128}+1$, they eventually decided on $k=257$ after numerous experiments. It appears to us that, nearly all the time, $k=1$ will suffice, and, indeed, we never encountered a number for which we had to choose a different k (other than $2^{128}+1$, which was too large for our computer allocation in any case).

The other parameter that has to be chosen is the number of primes to be used. It appears to us that $K=n^{2.5}/80$ is about right for numbers in the neighbourhood of 10^n, though this number will depend on the implementation. The algorithm requires K^2 words of storage for the system of linear equations, and is the only algorithm we recommend that requires a non-trivial amount of working storage.

Extensive experiments[8] show a $N^{0.154}$ behaviour, though the granularity effects of two-word numbers versus three-word numbers (in a 32-bit implementation) mean that the behaviour observed is bound to be complicated.

4. Conclusions

A simple program for factoring medium-sized integers can be written as follows:

1. Remove all small factors.
2. Test if the number is prime (three applications of the probabilistic primality test).
3. If not, try the continued fraction method.

For a more subtle program, the Monte Carlo method can be tried between steps 2 and 3, but it should be given a time limit† of perhaps 10% or 20% of the time that the continued fraction algorithm is expected to take. This will not cost much more if the Monte Carlo attempt fails, but will lead to substantial savings if it succeeds (which it should do if the factorization is unbalanced, or if N factors into the product of three or more primes).

The other conclusion is that the time taken is nearly all spent in multiple-precision arithmetic routines, and that these will probably need to be written in machine code (we observed a factor of about 3 when we converted the kernel of the multiple-precision package from BCPL into machine-code). However, the numbers involved are not very large (on a 32-bit machine they will typically be double or triple length), so careful attention has to be paid to the overheads of calling subroutines to do multiple-precision arithmetic.

Acknowledgements

We are grateful to the referee for his remarks, and to Prof. D. J. Wheeler for reading the drafts.

† In our implementation (more details can be found in Ref. 14), we were restricted to moduli greater than 32, and we used moduli 2*19, 3*17, 5*13, 7*11, 37, 41, 43 . . . .

‡ Note that this formula does depend on various details of the implementation, and suitable experiments have to be performed for each implementation, though our figures may serve as a starting point.

† A feature that many operating systems lack, or at least make very difficult to use.

R. J. MACMILLAN†
J. H. DAVENPORT‡
Emmanuel College
Cambridge CB2 3AP
UK

References

1. J. P. Buhler, R. E. Crandall and M. A. Penk, Primes of the form $n!\pm 1$ and $2.3.5\ldots p\pm 1$. *Math. Comp.* **38**, 639–643 (1982).
2. G. B. Gostin and P. B. McLaughlin, Jr., Six new factors of Fermat numbers. *Math. Comp.* **38**, 645–649 (1982).
3. D. E. Knuth, *The Art of Computer Programming, Vol. II, Semi-numerical Algorithms.* Second Edition, Addison-Wesley, 1981.
4. J. M. Pollard, A Monte Carlo method for factorization. *B.I.T.* **15**, 331–334 (1975).
5. R. Gold and J. Sattler, Modifikationen des Pollard-Algorithmus. *Computing* **30**, 77–89 (1983).
6. D. H. Lehmer and E. Lehmer, A new factorization technique using quadratic forms. *Math. Comp.* **28**, 625–635 (1974).
7. M. A. Morrison and J. Brillhart, A method of factoring and the factorization of F_7. *Math. Comp.* **29**, 183–205 (1975).
8. M. C. Wunderlicht, A running-time analysis of Brillhart's continued fraction factoring algorithm. In *Number Theory Carbondale 1979* edited by M. B. Nathanson (*Springer Lecture Notes in Mathematics 751*, Springer-Verlag, Berlin-Heidelberg-New York) 328–342 (1979).
9. N. A. Draim, An algorithm on divisibility. *Mathematics Magazine* **25**, 191–194 (1952).
10. H. Davenport, *The Higher Arithmetic.* 5th edn revised by D. J. Lewis and J. H. Davenport, Cambridge University Press, 1982.
11. L. Adleman and F. T. Leighton, An $O(n^{1/10.89})$ primality testing algorithm. *Math. Comp.* **36**, 261–266. Zbl. 452.10011. MR 82c:10009 (1981).
12. C. P. Schnorr, Refined analysis and improvement on some factoring algorithms. *Journal of Algorithms* **3**, 101–127. Zbl. 485.10004 (1982).
13. R. S. Lehman, Factoring large integers. *Math. Comp.* **28**, 637–646 (1974).
14. R. J. Macmillan, The implementation of a program which factorizes large integers on the IBM 370/165. *Dissertation for the Computer Science Tripos,* University of Cambridge (1982).
15. D. H. Lehmer and R. E. Powers, On factoring large numbers. *Bull. A.M.S.* **37**, 770–776 (1931).

Received February 1983

† Present address: 5 Overcombe Drive, Weymouth, Dorset, UK.
‡ Present address: School of Mathematics, University of Bath, Claverton Down, Bath BA2 7AY, UK.