

From 1966 Ph.D. Thesis "On the Minimum Computation Time of Functions" by Stephen A. Cook

Harvard University

CHAPTER III

POSITIVE RESULTS

Dept. of Mathematics

16. The Toom algorithm for multiplication.

A. L. Toom [14] has shown that for every n , there is a logic net of no more than $C_1 n C_2^{\sqrt{\log n}}$ components which realizes the multiplication of two arbitrary n -digit binary numbers, for some constants C_1, C_2 . Implicit in his construction is an algorithm for carrying out multiplication that is considerably more efficient than the standard multiplication algorithm, provided the numbers of digits in the factors are sufficiently large. In fact, the main purpose of this chapter is to show that the Toom algorithm can be utilized by a multitape Turing machine to multiply any two n -digit numbers within $n 2^{5\sqrt{\log n}}$ steps, for all n . We shall also show that the method can be extended to cover other computations besides multiplication, and derive some consequences concerning the relative difficulty of computing the radix expansions of real numbers in different radix bases (section 21). One result (20.1) concerning the rate at which a Turing machine can compute the radix expansions of algebraic numbers improves on a theorem of Hartmanis and Stearns [6].

Toom's method (slightly modified) can be presented as follows. Suppose M and N are the integers to be multiplied, and suppose the base b expansions of M and N are $\mu_n \mu_{n-1} \dots \mu_0$ and $\nu_n \nu_{n-1} \dots \nu_0$, respectively. Thus

$$M = \sum_{k=0}^n \mu_k b^k, \text{ and}$$

16.1.

$$N = \sum_{k=0}^n v_k b^k.$$

Choose integers q, r such that $n + 1 \leq q(r + 1)$ and divide the expansions of m and n into $r + 1$ segments of q digits each. Thus, if α_k is the number represented by the k^{th} segment of M and β_k is the number represented by the k^{th} segment of N , then

16.2.

$$\left. \begin{aligned} \alpha_k &= \sum_{i=0}^{q-1} \mu_{i+kq} b^i \\ \beta_k &= \sum_{i=0}^{q-1} v_{i+kq} b^i \end{aligned} \right\} \quad k = 0, 1, \dots, r,$$

where we define $\mu_k = v_k = 0$ for $k > n$. Next define the two polynomials $P(x)$ and $Q(x)$ by

$$P(x) = \sum_{k=0}^r \alpha_k x^k, \text{ and}$$

16.3.

$$Q(x) = \sum_{k=0}^r \beta_k x^k,$$

so that $M = P(b^q)$ and $N = Q(b^q)$. To find the product MN it is sufficient to find the coefficients of the polynomial $P(x)Q(x)$ and evaluate

the polynomial at $x = b^q$. Find the coefficients by first calculating the values

$$16.4. \quad \left. \begin{array}{l} m_k = P(k) \\ n_k = Q(k) \end{array} \right\} \quad k = 0, 1, \dots, 2r,$$

then finding the product $m_k n_k$ by applying the general method all over again, and finally solving for the γ_k 's the system of equations

$$16.5. \quad \gamma_0 + \gamma_1 k + \gamma_2 k^2 + \dots + \gamma_{2r} k^{2r} = m_k n_k, \quad k = 0, 1, \dots, 2r.$$

A near-optimal relationship between q and r is given by $r = 2^{\lceil \sqrt{\log q} \rceil}$, where $\lceil x \rceil$ is the greatest integer in x .

Although Toom designed his logic net to multiply integers in radix notation it turns out the same method works for multiplying polynomials over the integers modulo 2; or more generally, polynomials over Z_b , where b is prime, or polynomials over any finite field. A multitape Turing machine can utilize the algorithm to multiply such polynomials (i.e. compute the function f defined in 5.7) at the same rapid rate as for ordinary multiplication: $T(n) \leq n^{2^{5\sqrt{\log n}}}$.

It seems especially interesting that this type of multiplication can be performed so rapidly, since it is so easy to visualize the dependence of the output on the input (say for the case $b = 2$). Each successive output digit is a sum of products of the preceding input symbols, and if each output calculation were independent of the others, the computing time would grow as n^2 . It is far from obvious that some

of the work used in generating early outputs can be used for later outputs as well, but in fact the method of Toom shows a great time savings can be affected by working on producing long sequences of outputs simultaneously.

To adapt the Toom algorithm to polynomial multiplication, treat $M = M(W)$ and $N = N(W)$ as polynomials in the indeterminate W with coefficients μ_0, \dots, μ_n and ν_0, \dots, ν_n from Z_b . Thus equations 16.1 hold when b is replaced by W . The α_k 's and β_k 's become polynomials in W of degree $q - 1$ or less so that 16.2 holds when b is replaced by W and 16.3 holds as it is. Thus $P(x) = P(W, x)$ and $Q(x) = Q(W, x)$ are polynomials in the two indeterminants W and x such that $M = P(W, W^q)$ and $N = Q(W, W^q)$. In equations 16.4, k must be replaced by k^* , where k^* is the polynomial in W whose string of coefficients is the base b notation for the integer k . Thus $0^*, 1^*, \dots, (2r)^*$ are $2r + 1$ distinct polynomials none of whose degrees exceeds $\lceil \log_b(2r + 1) \rceil$. Then the system of equations 16.5, with k replaced by k^* , can be solved for the γ_k 's as before, as we shall see in section 18. This is the step in the algorithm that requires b to be prime, since if b is not prime, then the polynomials over Z_b do not form an integral domain, and the system 16.5 may not have a unique solution.

It is interesting that the time estimates derived here for carrying out the various parts of the algorithm for ordinary multiplication turn out to be essentially the same as the estimates made by Toom on the amount of equipment required by the logic net. A little thought shows

that this cannot be an instance of a general phenomenon, since logic nets can be designed to compute non-recursive functions; something Turing machines cannot do at any speed. In fact, the estimates for the logic net are very easily verified, but we shall spend some effort in showing that the Turing machine is able to carry out the necessary bookkeeping and set-up operations fast enough so that the corresponding time estimates apply. The main difficulty comes in solving the system of linear equations: the inverse of the matrix of the system can be built into the logic net, but not into the finite state control of the Turing machine, since q and r can be arbitrarily large. Hence the Turing machine must compute the inverse.

The problem, in other words, is to show the method is uniform in n , the number of digits in the factors. It might be of some interest to isolate and study the class of functions which have the property that the minimum amount of equipment needed to realize them by a logic net grows at the same rate as the computation time of the fastest Turing machine which computes them. Of course we have no proof that multiplication is such a "uniform" function, but it does seem plausible.

17. A Turing machine which incorporates the algorithm.

In this section we give a very general description of a multitape Turing machine which realizes the Toom algorithm, and then we estimate the computing time of the machine. The estimate depends on an inequality (17.6) whose proof is postponed until section 19, where we give a more detailed description of the machine.

17.1. Theorem: For every base b , there is a multitape Turing machine which, given an arbitrary pair of n -digit input integers in base b notation, will compute their product in base b notation within $n 2^{5\sqrt{\log n}}$ steps, for all n . The same applies to polynomial multiplication over Z_b (cf. 5.7), provided b is prime.

The input-output arrangements of the Turing machine are those discussed in section 5 (cf. 5.6 and 5.7). In sum, the digits of the factors are written on a special read-only input tape, a pair of digits on each square of the tape. The machine advances the tape at irregular intervals to read the information on this tape. The actual computation is carried out on several ordinary linear tapes, which have one read-write head per tape. The output digits are given by an output function λ from the displays of the machine to the set of digits $\{0, 1, \dots, b - 1\}$. Although 6.1 could be taken as the definition of a multitape Turing machine, it will be convenient to assume that the multiplying machine has a finite set of internal states (which are not provided for in definition 6.1). Thus we can think of the machine as having special output states which designate the product digits. There is no on-line restriction; that is the machine reads input digits faster than it designates output digits.

The machine described in section 19 has many tapes, but it is possible to take advantage of a result of Hennie and Stearns [9] to show the number of tapes can be reduced to two with only slight loss in time. Hennie and Stearns show that any multitape Turing machine can be simulated by a two-tape Turing machine at the rate of $n \log n$

steps on the two-tape machine for n steps on the multitape machine. The slower rate of the two-tape machine boosts the computation time from $n^{2^{5/\log n}}$ to $n^{2^{(5+\epsilon)/\log n}}$, but ϵ can be taken to be an arbitrarily small positive number by applying the speed-up theorem of Hartmanis and Stearns [6].

The multiplying machine, whether it multiplies polynomials or integers, utilizes the algorithm described in the previous section. The discussion below assumes integer multiplication, but it applies, with only a few modifications, to polynomial multiplication. It is convenient to think of the machine as having a program which is organized into a main routine and a subroutine. The main routine handles the input and output, and initializes the parameters, while the subroutine does most of the computation.

The machine does not operate on the entire input string at once, but rather the computation proceeds in stages. During the first stage, the main routine sets the parameters r and q equal to the initial values r_1 and q_1 (described in section 19) and reads in $q_1(r_1 + 1)$ pairs of input digits, thus determining the initial values for the factors M and N . The main routine then makes M and N available to the subroutine, which calculates the product MN and returns control to the main routine. The main routine designates the product as output, updates the values of q and r , and the process begins all over again. In general, the values of q and r are selected from the sequences q_1, q_2, \dots and r_1, r_2, \dots , which the main routine computes using the

recursion equations

$$q_{i+1} = r_i q_i$$

17.2.

$$r_{i+1} = 2^{\lfloor \sqrt{\log q_{i+1}} \rfloor},$$

where $\lfloor x \rfloor$ means the greatest integer in x . On the i^{th} stage, the factors M and N consist of the first (i.e. lowest order) $q_i(r_i + 1)$ pairs of input digits. In order for the subroutine to calculate the product MN , it must first calculate the lesser products of the form $m_k n_k$ appearing on the right side of equation 16.5, and hence it must be able to call (i.e. re-enter) itself with new input parameters, while not destroying the old ones. We shall prove in lemma 17.3 below that if q_i and r_i are properly chosen, then the numbers m_k and n_k defined in 16.4 have at most $q_i + q_{i-1} = q_{i-1}(r_{i-1} + 1)$ digits each, so that the subroutine is able to set the parameters q and r equal to q_{i-1} and r_{i-1} while calculating $m_k n_k$ (whereas the values were q_i and r_i for calculating MN). In the process of calculating $m_k n_k$ the subroutine must in general call itself again, but since the values of q and r decrease with each increase in the depth of nesting of the calls, the maximum possible nesting depth on the i^{th} stage of the computation is at most i .

17.3. Lemma. Suppose the values of q and r are taken to be q_i and r_i in the execution of the Toom algorithm. Then the base b notations for the integers m_k and n_k defined in 16.4 have no more than $q_{i-1}(r_{i-1} + 1)$

digits each, provided only the initial values r_1 and q_1 are chosen so that

$$17.4. \quad r_1 = 2^{\lceil \sqrt{\log q_1} \rceil},$$

and q_1 is sufficiently large.

Proof. We have

$$m_k = \sum_{j=0}^{r_1} \alpha_j k^j < (r_1 + 1) b^{q_1} (2r_1)^{r_1},$$

since each α_j has at most q_1 digits. Thus, if $l(m_k)$ is the number of digits in m_k , then (since $b \geq 2$)

$$17.5. \quad \begin{aligned} l(m_k) &\leq \log_b m_k + 1 \leq \log(r_1 + 1) + q_1 + r_1 \log(2r_1) + 1 \\ &\leq q_1 + r_1^2 \end{aligned}$$

if r_1 is sufficiently large. Now set $t = i - 1$. Then $q_i = q_t r_t$, and

$$\begin{aligned} \log r_1^2 &= 2 \log r_1 = 2 \lceil \sqrt{\log q_1} \rceil \leq 2 \sqrt{\log(q_t r_t)} \\ &\leq 4 \sqrt{\log q_t} \leq \log q_t, \end{aligned}$$

if q_t is sufficiently large. Thus $r_1^2 \leq q_t$, so by 17.5, $l(m_k) \leq q_1 + q_{i-1} = q_{i-1}(r_{i-1} + 1)$, which proves the lemma for the case of m_k . The proof for n_k is the same.

We shall now find an upper bound for the time required by the Turing machine to multiply, using the time estimates calculated in section 19. First, let t_1 be the maximum possible number of steps the

subroutine could require to multiply two numbers for which it sets $q = q_i$ and $r = r_i$. The time taken by the subroutine to perform the multiplication can be divided into two parts, depending on whether the machine is currently directly under the control of the original entrance to the subroutine, or whether it is under the control of a later entrance. According to section 19, the first part does not exceed $Dr_i^5 q_i$ steps for some constant D . Since the subroutine must call itself $2r_i + 1$ times to compute the $2r_i + 1$ products $a_k n_k$, and since by lemma 17.3 the values of q and r for each of these times are at most $q = q_{i-1}$, $r = r_{i-1}$, the second part of the multiplying time does not exceed $(2r_i + 1)t_{i-1}$. Therefore,

$$17.6. \quad t_i \leq Dr_i^5 q_i + (2r_i + 1)t_{i-1}, \quad i = 2, 3, \dots$$

From 17.6 we shall prove by induction

$$17.7. \quad t_i \leq Cq_{i+1} 2^{4\sqrt{\log q_{i+1}}}, \quad i = 1, 2, \dots$$

for some constant C . We assume 17.4 holds, and that q_1 is large enough for the assertions below to be valid. When $i = 1$, 17.7 certainly holds for sufficiently large C . Now suppose $i > 1$. By the induction hypothesis, we have

$$t_{i-1} \leq Cq_i 2^{4\sqrt{\log q_i}},$$

and by 17.2,

$$r_i = 2^{\lceil \sqrt{\log q_i} \rceil}.$$

Since $(2r_i + 1) \leq 2^{1.1} r_i$, 17.6 now gives

$$17.8. \quad t_i \leq q_i 2^{d+5[\sqrt{Q}]} + q_i 2^{1.1+[\sqrt{Q}]+4\sqrt{Q}+c},$$

where $d = \log D$, $Q = \log q_i$, and $c = \log C$. To carry out the induction, we must establish 17.7. Since $q_{i+1} = q_i r_i$, this is equivalent to showing

$$17.9. \quad t_i \leq q_i 2^{[\sqrt{Q}]+4\sqrt{Q}+[\sqrt{Q}]+c}.$$

Thus we need only show the right side of 17.8 does not exceed the right side of 17.9. We do this by showing the second term of 17.8 does not exceed $2^{-.2}$ times the right side of 17.9, and the first term of 17.8 does not exceed $(1 - 2^{-.2})$ times the right side of 17.9.

That is, we establish the two inequalities

$$17.10. \quad d + 5[\sqrt{Q}] \leq [\sqrt{Q}] + 4\sqrt{Q} + [\sqrt{Q}] + c - \ell$$

and

$$17.11. \quad 1.1 + [\sqrt{Q}] + 4\sqrt{Q} + c \leq [\sqrt{Q}] + 4\sqrt{Q} + [\sqrt{Q}] + c - .2,$$

where $-\ell = \log(1 - 2^{-.2})$. If we choose C so large that $c = \log C \geq d + \ell$, then 17.10 is valid, and 17.11 certainly holds for sufficiently large Q (and hence for sufficiently large q_i). Hence the induction is complete and 17.7 is established.

Now let $T_1(n)$ be the number of steps required for the machine to designate the first n (low-order) digits of the product of the two numbers on the input tape. Let q_1 be the value of q selected by the

subroutine in multiplying an input segment which includes the first n input pairs of digits. Then $q_1 < n$. Previous to this pass of the subroutine, the subroutine was entered at most $i - 1$ times from the main routine, each time with a value of q less than q_1 . Since the main routine does much less computation than the subroutine, it is clear that at any point in the computation after at least one output digit has been designated, the total time to date used by the subroutine exceeds the time used by the main routine. From these facts and the inequalities $j < r_j$ and $r_j t_{j-1} < t_j$, $j = 1, 2, \dots$, we conclude

$$17.12. \quad T_1(n) < 4t_1.$$

Since $q_1 < n$, $q_{i+1} = q_i r_i$, and $r_i = 2^{\lceil \sqrt{\log q_i} \rceil}$, we have by 17.7 and 17.12

$$17.13. \quad T_1(n) < 4Cn 2^{4\sqrt{\log n} + \sqrt{\log n} + \sqrt{\log n}}$$

But for any $x > 0$, $\sqrt{x} + \sqrt{x} < \sqrt{x} + \frac{1}{2}$, so 17.13 yields

$$17.14. \quad T_1(n) < 16Cn 2^{5\sqrt{\log n}}.$$

To establish the time bound in theorem 17.1, it suffices to eliminate the constant $16C$. According to the speed-up theorem of Hartmanis and Stearns [6], the computation time $T(n)$ of a multitape Turing machine can always be decreased by a constant factor $\epsilon > 0$, provided $\epsilon T(n) \geq n$ for all n , and provided the machine is a sequence generator (i.e. has no inputs). The speed-up theorem is proved by

replacing the old machine with a new one with a larger alphabet of tape symbols. Each of the new tape symbols encodes a finite sequence of the old, so one step of the new machine does the work of several steps of the old. The condition of no inputs does not hold for our multiplying machine, but the method of Hartmanis and Stearns can be made to apply anyway. All that is necessary is to have the multiplying machine encode segments of the input tape into the more compact format as they are read in.

The time bound $T(n) \leq n 2^{5\sqrt{\log n}}$ stated in theorem 17.1 requires that the machine put out product digits at the rate of one per machine cycle for some initial period of time. This is easily accomplished by attaching a large finite-state multiplier which multiplies by "table look-up" until the subroutine is able to supply the first string of product digits.

Thus the proof of 17.1 will be complete as soon as the inequality 17.6 is established in section 19.

18. Solution of the linear equations.

The algorithm used to solve the system 16.5 of linear equations is the standard one of applying elementary row operations to the coefficient matrix until it is reduced to the identity matrix. We shall discuss this algorithm in some detail to show that it can be carried out sufficiently rapidly both for the case of integer multiplication and polynomial multiplication.

It will be convenient to cast the discussion in general setting.

Thus we assume the problem is to solve for the γ_k 's the system

$$18.1. \quad \gamma_0 + x_i \gamma_1 + x_i^2 \gamma_2 + \dots + x_i^{n-1} \gamma_{n-1} = y_i, \quad i = 1, 2, \dots, n,$$

where x_1, x_2, \dots, x_n and y_1, y_2, \dots, y_n are elements from some integral domain R . For the case of integer multiplication, R is the ring of ordinary integers, while for polynomial multiplication, R is the ring of polynomials in W over Z_b , where b is prime.

Elementary row operations are applied to the coefficient matrix

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & & & & \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{pmatrix}$$

to transform it to the identity matrix. When these same row operations are applied in order to the column matrix

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

the result is a column matrix giving the values of $\gamma_0, \dots, \gamma_{n-1}$. Since these values lie in R , it is always possible to choose the row operations in such a way that all entries in the $n \times n$ matrix are in R after each step in the computation. One simply multiplies rows by suitably large members of R . It is less obvious, but nevertheless true,

that the entries in the matrix remain in R without any "extra" multiplications. This fact will come out of the description of the algorithm below and we shall use it in the next section to keep the computation time down.

In describing the row operations, we shall restrict our attention to the $n \times n$ matrix of X 's, and ignore the column matrix. Notice that since $\gamma_0, \dots, \gamma_{n-1}$ all lie in R , the entries in the column matrix will automatically lie in R after each row operation has been applied, provided the entries in the $n \times n$ matrix remain in R .

It will be convenient to think of X_1, X_2, \dots, X_n not as members of R , but as indeterminants or variables for polynomials with coefficients in R . The specific case we are interested in can then be obtained by substituting the proper values in R for the variables X_1, \dots, X_n . We shall denote the set of all polynomials in the variables X_1, \dots, X_k with coefficients in R by $R[X_1, \dots, X_k]$. If $P(X)$ is a polynomial in X with coefficients in $R[X_1, \dots, X_k]$, then $P(X)$ is monic provided the coefficient of the highest power of X is 1; or, if X does not occur, then $P(X)$ is monic exactly if it is the constant polynomial 1. Thus 1 and $X + 3X_1^2$ are monic as polynomials in X , but $X_1 + X_2$, and XX_1 are not.

The algorithm consists of n major steps. We denote by A_k the result of applying the first k of these steps to the matrix A , $k = 0, 1, \dots, n$. We shall prove by induction on k that A_k consists of four rectangular sub-matrices, as follows:

$$A_k = \begin{pmatrix} I_k & N_k \\ O_k & T_k \end{pmatrix},$$

where I_k is the $k \times k$ identity matrix, O_k is the $(n - k) \times k$ matrix of zeroes, N_k is a $k \times (n - k)$ matrix of polynomials over the integers, and T_k is a $(n - k) \times (n - k)$ matrix whose $(i - j)^{th}$ entry is

$$p_{k+j}^{(k)}(x_{k+i}).$$

Here $p_{k+1}^{(k)}(x)$, $p_{k+2}^{(k)}(x)$, ..., $p_n^{(k)}(x)$ are monic polynomials in x of degrees $0, 1, 2, \dots, n - k - 1$ with coefficients in $R[x_1, \dots, x_k]$. Thus, in particular, $p_{k+1}^{(k)}(x)$ is simply the polynomial 1.

Notice that for $k = 0$, $A = A_0$ satisfies this hypothesis; for in this case A_k and T_k coincide, and the polynomials $p_1^{(0)}(x), \dots, p_n^{(0)}(x)$ are successive powers of x .

The $(k + 1)^{st}$ major step of the algorithm consists of the following.

(i) Subtract the $(k + 1)^{st}$ row from each succeeding row, and appropriate multiples of the $(k + 1)^{st}$ row from each preceding row, so that each entry except the diagonal entry of the $(k + 1)^{st}$ column is 0. Then the entries of the row $k + i$, $i > 1$, are

$$\underbrace{0 \quad \dots \quad 0}_{k+1} \quad p_{k+2}^{(k)}(x_{k+1}) - p_{k+2}^{(k)}(x_{k+1}) \quad \dots \quad p_n^{(k)}(x_{k+1}) - p_n^{(k)}(x_{k+1}).$$

But notice that for any monic polynomial $P(x)$, we have $P(x) - P(y) = (x - y)Q(x)$, where $Q(x)$ is monic of degree one less than $P(x)$ (and has

polynomials in Y as coefficients). Thus we can rewrite the row in the form

$$\underbrace{0 \quad \dots \quad 0}_{k+1} (X_{k+1} - X_{k+1})Q_{1+2}(X_{k+1}) \quad \dots \quad (X_{k+1} - X_{k+1})Q_n(X_{k+1}).$$

The second part of the $(k+1)^{\text{st}}$ step is

(ii) Divide row $k+1$ by $(X_{k+1} - X_{k+1})$, for $i = 2, 3, \dots, n$. The result is

$$\underbrace{0 \quad \dots \quad 0}_{k+1} Q_{k+2}(X_{k+1}) \quad \dots \quad Q_n(X_{k+1})$$

(i) and (ii) complete the $(k+1)^{\text{st}}$ step of the algorithm. The induction hypothesis is satisfied for $k+1$ if we set $P_{k+j}^{(k+1)}(X) = Q_{k+j}(X)$.

After all n major steps have been carried out, the result is the matrix A_n , the $n \times n$ identity matrix, as desired.

Time estimates for carrying out the algorithm are given in the next section.

19. Time estimates for the Turing machine.

The purpose of this section is to justify the inequality 17.6 by showing that if the subroutine chooses $q = q_1$ and $r = r_1$ for performing a given multiplication, then the time spent by the subroutine during that "pass" does not exceed $Dr_1^5 q_1$ steps, for some constant D (independent of i). (The notion of "pass" will be explained below). In order to show this, it is necessary to give a rather explicit description of the Turing machine. The description will be of a machine which multiplies

integers in base b notation, but only slight modifications are necessary to handle the case of polynomial multiplication.

The machine we shall describe has twenty linear tapes (with one read-write head per tape) in addition to the read-only input tape. The number can certainly be substantially reduced, but only with some sacrifice of clarity of exposition. We shall call ten of the twenty tapes work tapes. Of the remaining ten, three are designated scratch tapes, and the other seven are named as follows: storage tape, output tape, argument tape, r -tape, q -tape, R -tape, Q -tape. Any other specific names we use will refer to work tapes.

The heads can read and write any of the $b + 6$ symbols $0, 1, \dots, b - 1$ (the b digits), δ , ϵ , λ , $\bar{\lambda}$, $-$ (minus), blank. In general, integers are stored in base b notation on the tapes, with the symbol $'-'$ to the right of negative integers. Initially the base b expansions of two integers M and N are assumed to be on the input tape with the read-only head scanning the low order (right-most) digit. All squares of all other tapes are blank. The machine acts as though all the work tapes are one-way to the right. That is, initially it places the symbol λ on each of the work tapes, and the heads never move to the left of this symbol. The machine operates in such a way that it never leaves blanks interspersed with the other symbols on the work-tapes; that is, the only blanks are either to the left of all non-blank symbols or to the right of them.

Each time the subroutine is entered, the heads of the work tapes each move to the first blank square to the right of their present location and print a λ . The heads remain to the right of this λ until exiting from that pass through the subroutine, at which time the λ and all information to the right of it is erased. In this way the information from many different passes through the subroutine is simultaneously stored on the work tapes in segments separated by λ 's. (This should clarify the notion of "pass").

In exiting from a given pass through the subroutine the machine must know whether to return control to the subroutine or to the main routine. (There is no ambiguity about where to return within these routines since each calls the subroutine from only one point). Thus when the main routine calls the subroutine the machine prints a $\bar{\lambda}$ to the right of the information on the work tapes instead of a λ .

The input to the subroutine — namely the integers to be multiplied — is always furnished to the subroutine via the argument tape, and the subroutine writes the product on the argument tape before exiting. Since this information is transferred onto work tapes immediately after entering and after exiting respectively there is no need to preserve the information on the argument tape between passes to the subroutine. The same applies to the three scratch tapes. The subroutine does not use the remaining non-work tapes, except the storage tape, where the use is uniform for all passes.

The initial segment of the input tape currently being operated on is always stored on the storage tape. Once the main routine has read in the next increment of the input tape and determined q and r for a given iteration of the algorithm, the machine indicates the division of the numbers on the storage tape into segments of length q by placing the symbol δ every q digit-pairs, starting from the right end (except in all but the initial iteration an ϵ precedes the first q digits so no δ is needed). The $(r + 1)^{\text{st}}$ segment of length q is delineated by an ϵ instead of a δ . Thus, for example, after two iterations the storage tape looks like

$$\underbrace{\epsilon^{**}\dots*\delta}_{q_2 \text{ digits}} \quad \dots \quad \underbrace{\delta^{**}\dots*\epsilon^{**}\dots*\delta}_{q_2 \text{ digits}} \quad \underbrace{\epsilon^{**}\dots*\delta}_{q_1 \text{ digits}} \quad \dots \quad \underbrace{\delta^{**}\dots*\epsilon}_{q_1 \text{ digits}}$$

The output tape is used simply to keep track of which output digits have been designated so as to prevent repetitions.

The following are outlines of the main routine and subroutine.

Main Routine

- (1) Update the values of q and r which appear on the q and r tapes.
- (2) Initiate or continue the copying of the input tape onto the storage tape until a total of $(r + 1)q$ digit-pairs appear on the storage tape. In the process, insert several δ 's and then an ϵ , so that $r + 1$ blocks of q pairs each are delineated (in addition to any smaller blocks already delineated).
- (3) Copy the storage tape up to the last (left-most) ϵ onto the argument tape, omitting all δ 's and ϵ 's. (Erase all else on the argument tape).

- (4) Shift all heads on work tapes to the right until a blank symbol is reached, print a $\bar{\lambda}$ here, and enter the subroutine.
- (5) After returning from the subroutine, copy the product from the argument tape onto the output tape and designate as output the portion not already appearing on the output tape.
- (6) Go to 1.

Subroutine

- (1) Copy the two integers on the argument tape onto the alpha tape and beta tape (work tapes); one integer for each tape. Use δ 's to separate the information on each tape into at most $r + 1$ segments of length q each. Here q and r must be a pair (q_1, r_1) from the sequences q_1, q_2, \dots and r_1, r_2, \dots of values which have occurred (or presently occur) on the q and r tapes, and are chosen as small as possible such that the number of digit-pairs on the argument tape does not exceed $q(r + 1)$. q and r are not determined explicitly, but rather the position of the δ 's is determined by reference to the δ 's and ϵ 's already in place on the storage tape.
- (2) If the argument has fewer than $q_1 r_1$ digit-pairs, perform the multiplication directly, copy the result back onto the argument tape, and go to (10).
- (3) Calculate $m_k = \sum_{i=0}^r \alpha_i k^i$, $k = 0, 1, \dots, 2r$ and place the sequence m_0, m_1, \dots, m_{2r} , separated by δ 's, on the T1 work tape. The α_i are determined from the alpha tape; if there are fewer than $r + 1$ of them, fill in with zeroes. Do the same for the α_k 's and the beta tape, putting the result on the T2 work tape.

- (4) Mark m_0 on the T1 tape by placing an ϵ to the right of it. Do the same for n_0 on the T2 tape.
- (5) Superimpose the m_k and n_k which are marked by ϵ 's onto the argument tape, print λ 's to the right of the information on all work tapes, and go to (1) (i.e. reenter the subroutine).
- (6) After returning from the subroutine append the product $m_k n_k$ which appears on the argument tape to the list being formed on the T3 work tape, separating it from previous entries by a δ .
- (7) If the last m_k on the T1 tape (that is, m_{2r}) is the one currently marked with an ϵ , go to (8). Otherwise change the ϵ 's on the T1 and T2 tapes to δ 's, and mark the next m_k and n_k (i.e. m_{k+1} and n_{k+1}) with ϵ 's and go to (5).
- (8) The sequence $m_0 n_0, \dots, m_{2r} n_{2r}$ now appears on the T3 tape. Use it to solve for $\gamma_0, \dots, \gamma_{2r}$ the system of linear equations
- $$\gamma_0 + \gamma_1 k + \dots + \gamma_{2r} k^{2r} = m_k n_k, \quad k = 0, 1, \dots, 2r.$$
- (9) From the values obtained in (8), calculate
- $$S = \gamma_0 + \gamma_1 b^q + \dots + \gamma_{2r} b^{2rq}$$
- and place S on the argument tape.
- (10) Erase the right-most λ or $\bar{\lambda}$ and all information to the right thereof on all work tapes. If this symbol is a λ , go to (6) of the subroutine; if the symbol is a $\bar{\lambda}$, go to (5) of the main routine.

In estimating the time required by each part of the program we shall make use of the standard O notation. If f and g are functions of positive integers, then $f(n) = O(g(n))$ means there is a constant C so that $f(n) \leq Cg(n)$ for all positive n . We shall show that part (8) of the subroutine (solution of the linear equations) takes more time than any other part, but (8) requires no more than $O(r^5 q)$ machine steps. This will justify the assertion made at the beginning of this section.

The machine can make use of the three scratch tapes S_1 , S_2 , S_3 to carry out the additions, multiplications, and divisions required by the algorithm. It is not hard to see that if an s -digit number appears on S_1 , and a t -digit number on S_2 , then using the standard methods and no tapes besides S_1 , S_2 , and S_3 the machine can write on S_3 the sum of the two numbers in $O(s + t)$ steps, the product in $O(st)$ steps, and the quotient (provided this is an integer) in $O(st)$ steps.

We shall now analyze each of the parts of the program. First, the main routine:

(1) Update the values of q and r . The machine uses the four tapes labelled q , r , Q , R for this purpose. The initial values q_1 and r_1 are chosen so as to satisfy 17.4 with q_1 large enough to enable the estimates in section 17 to be valid. The new values r' , q' , R' , Q' are calculated by the equations

$$Q' = R + Q$$

$$R' = \lfloor \sqrt{Q'} \rfloor$$

$$q' = 2^{Q'}$$

$$r' = 2^{R'},$$

so that the recursion equations 17.2 are satisfied. The square root in the second equation is easily calculated digit by digit in $O(Q') = O(\log q')$ steps, while the third operation requires Q' multiplications of numbers with no more than Q' digits each and so requires $O((\log q')^3)$ steps. Thus, conservatively, part (1) requires $O(q')$ steps.

(2) - (7) These parts require $O(r'q')$ machine steps plus the time required to make a pass through the subroutine.

Now let us analyze the subroutine. The values of q and r in this analysis are the values selected by part (1) of the subroutine.

(1), (2) These parts in the subroutine require $O(rq)$ steps.

(3) Calculating each m_k and n_k requires r multiplications of numbers of $O(r \log r)$ digits and r multiplications of q -digit numbers by numbers with $O(r \log r)$ digits. The $4r + 2$ numbers $m_0, \dots, m_{2r}, n_0, \dots, n_{2r}$ can be produced using the three scratch tapes and three index tapes in $O(qr^3 \log r)$ steps.

(4) - (7) These parts require $O(rq)$ steps plus the time for the $2r + 1$ passes through the subroutine.

(8) To carry out the algorithm described in section 18, the machine first writes the augmented coefficient matrix (the matrix including the $n + 1$ st column containing the products $m_k n_k$) on the T1 tape, with successive rows following each other.

$$T1: \lambda R_0 \epsilon R_1 \epsilon \dots \epsilon R_{2r}$$

where R_k is the k^{th} row:

$$R_k: \overline{1} \overline{\delta k} \overline{\delta k^2} \delta \dots \delta k^{2r} \overline{\delta m_k n_k}$$

Here \bar{j} indicates base b notation for j . (The negative entries which will subsequently appear will have minus (-) signs to the right of them).

There are two types of elementary row operations which the machine must carry out. The first is to divide a row by an integer, and the second is to subtract a multiple of one row from another. Let us consider the second. The machine is to replace a row R_k with $R_k - CR_j$, where C is an integer, and $j \neq k$. If we assume that the parameters j , k and C are specified by three tapes: the j -tape, k -tape, and C -tape, then the machine might proceed as follows: First, scan the $T1$ tape from left to right until R_j is reached. Then multiply the entries of R_j by C and put the result on a new tape $T2$. Next, copy the tape $T1$ onto a tape $T3$ from left to right until the row R_k is reached, then continue, except put $R_k - CR_j$ (as computed using $T2$) on $T3$ in place of R_k , and then complete the copying. Finally, copy $T3$ onto $T1$.

To estimate the time required, it is necessary to find an upper bound for the magnitude of the entries of the coefficient matrix (excluding the $n + 1^{\text{st}}$ column) during the course of the algorithm. Before execution of the k^{th} step in the algorithm, the entries of the k^{th} row have only been decreased, hence none of them exceeds $(2r)^{2r}$.

Thus the maximum of the entries of the matrix increases by at most a factor of $(2r)^{2r}$ on the k^{th} step. Hence after the $2r + 1$ steps of the algorithm have been completed, none of the entries exceeds (or has exceeded) $(2r)^{4r^2}$ in absolute value (notice that the first two steps increase no entries). Also, the maximum of the entries in the $n + 1^{\text{st}}$ column has increased by at most a factor of $(2r)^{4r^2}$.

Now consider again the elementary row operation "replace R_k by $R_k - CR_j$." The numbers of digits of the entries of the transformed coefficient matrix are all $O(r^2 \log r)$ and the number of digits of the $n + 1^{\text{st}}$ column is $O(q + r^2 \log r) = O(q)$. Thus the most time consuming operation is the multiplication C times the entry in the $n + 1^{\text{st}}$ column, which requires $O(r^2(\log r)q) = O(r^3q)$ steps.

A similar analysis shows the other type of row operation — division by a constant — also requires $O(r^3q)$ steps. Since there are $(2r + 1)^2$ row operations required to execute the algorithm, and the time for the "bookkeeping" operations is small compared with the execution time for the row operations, we find that the total number of steps required for solution of the system of equations is $O(r^5q)$.

Besides the six work tapes already mentioned (T_1, T_2, T_3, j, k, C), three additional work tapes easily suffice for overall control: one to specify $2r$, one to keep track of the major step $(1, 2, \dots, 2r + 1)$ of the algorithm, and one to keep track of which row is currently being operated on. This brings the total number of work tapes to nine.

(9) Since multiplication by a power of b amounts simply to writing a string of zeroes, the number of steps required to evaluate S is comparable to the number of steps required to perform the additions; that is $O(r^2q)$.

(10) This requires at most $O(rq)$ steps.

20. Extensions of the algorithm.

Once we have a method for multiplying rapidly, it is possible to use a simple polynomial iteration to devise a method of dividing rapidly, and once we have a method of dividing rapidly, we can apply Newton's method of approximating zeroes of functions to compute algebraic functions rapidly. We present some of these methods in this section by proving theorems concerning the rate at which Turing machines can compute. The first result concerns the calculation of algebraic numbers, and has a clean proof that does not depend on a Turing machine's ability to divide.

20.1. Theorem[†]: For every real algebraic number α and base $b \geq 2$ there is a multitape Turing machine which computes the base b expansion of α in time $T(n) \leq n 2^{5\sqrt{\log n}}$.

Proof. The machine doing the computation differs from machines introduced previously in that it has no input. The theorem asserts that the machine designates the first n digits of the expansion within $n 2^{5\sqrt{\log n}}$ machine steps, for all n .

[†] Hartmanis and Stearns [6] prove a weaker version with $T(n) = n^2$.

To prove the theorem, we need the following lemma.

20.2. Lemma. For every algebraic number α there is a polynomial $F(x)$ with rational coefficients and a constant B such that $|F(x) - \alpha| \leq B|x - \alpha|^2$ for all x with $|x - \alpha| \leq 1$.

Proof. Let $f(x)$ be a polynomial of least degree over the rationals such that $f(\alpha) = 0$. Then $f(x)$ and $f'(x)$ (the derivative of $f(x)$) are relatively prime, so there are polynomials (over the rationals) $r(x)$ and $s(x)$ such that

$$f(x)r(x) + f'(x)s(x) = -1.$$

Let

$$F(x) = f(x)s(x) + x.$$

Then $F(\alpha) = \alpha$, and since

$$F'(x) = f'(x)s(x) + f(x)s'(x) + 1,$$

it follows that $F'(\alpha) = 0$. Thus α is a root of both the polynomial $F(x) - \alpha$ and its derivative, so there is a polynomial $P(x)$ such that

$$F(x) - \alpha = (x - \alpha)^2 P(x).$$

The lemma follows when B is set equal to the maximum of $|P(x)|$ on the interval $|x - \alpha| \leq 1$.

Theorem 20.1 is proved by constructing a machine which approximates α by using an iterative technique based on the equation $x_{n+1} = F(x_n)$. On the first iteration, the machine produces a finite string of digits representing an initial segment of the base b expansion of α . This segment must be sufficiently long, so that if x_1 is the number

represented by the segment, then

$$|x_1 - \alpha| \leq b^{-4-c},$$

where c is a positive integer exceeding $\log_b B$.

After n iterations, the machine has produced an initial segment representing

$$20.3. \quad |x_n - \alpha| \leq b^{-2^n - 2 - c}.$$

Now let

$$20.4. \quad d = 2^{n+1} + 2 + c.$$

On the $n + 1^{\text{st}}$ iteration, the machine proceeds according to the following instructions:

(i) Calculate an approximation y_{n+1} to $F(x_n)$ such that

$$20.5. \quad |y_{n+1} - F(x_n)| < b^{-d-2}.$$

This is done by carrying out the required multiplications and additions to $d + m$ places, for sufficiently large m (m is independent of n).

(ii) Round y_{n+1} to exactly d places to the right of the radix point. Thus we obtain z_{n+1} such that

$$20.6. \quad |z_{n+1} - y_{n+1}| \leq \frac{1}{2} b^{-d}.$$

Since $F(x)$ satisfies the conditions of the lemma, we have by 20.3, 20.4, and the definition of c ,

$$20.7. \quad |F(x_n) - \alpha| \leq Bb^{-d-2-c} \leq b^{-d-2}.$$

From 20.5, 20.6, 20.7, and the fact $b \geq 2$ we obtain

$$20.8. \quad |z_{n+1} - \alpha| < b^{-d}.$$

(iii) Let $f(x)$ be a polynomial with integer coefficients such that $f(\alpha) = 0$, and $f'(\alpha) > 0$. Calculate $f(z_{n+1})$ and determine whether or not $f(z_{n+1}) > 0$, and hence whether or not $z_{n+1} > \alpha$. Now set

$$x_{n+1} = z_{n+1} - b^{-d} \quad \text{if } z_{n+1} > \alpha, \text{ and}$$

$$x_{n+1} = z_{n+1} \quad \text{otherwise.}$$

In either case the base b expansion of x_{n+1} terminates after d places to the right of the radix point. From 20.8 we see that these d places, together with those to the left of the radix point, are an initial segment of the base b expansion of α .

It remains to estimate $T(k)$, the number of steps required to designate the first k output digits. These k digits will all be designated by the $n^{\frac{t_n}{2}}$ iteration, where n is the least integer such that $2^n \geq k$. Thus there are constants C_1, C_2 such that on the $n^{\frac{t_n}{2}}$ iteration C_1 multiplications are performed, and the number of digits involved in each does not exceed $C_2 k$. According to theorem 17.1, these multiplications can be performed within $C_3 k 2^{5\sqrt{\log k}}$ steps for some constant C_3 . Since the number of steps required at least doubles with each successive iteration, and since the number of steps for a given iteration is bounded by a constant multiple of the number of steps required for multiplication in that iteration, there is a constant C_4 such that

$$T(k) \leq C_4 k 2^{5\sqrt{\log k}}.$$

The constant C_4 can be reduced to 1 by applying the speed-up theorem of Hartmanis and Stearns [6].

We now turn our attention to the problem of computing reciprocals. The input-output arrangements require special consideration in this case, since a radix expansion of the reciprocal of an integer may not terminate. One interesting model is to assume the input stands for a real number x between 0 and 1 and is presented to the machine with high order digits first, starting with the digit to the right of the radix point. The machine would then read in the digits of x and designate the digits of $\frac{1}{x}$ as fast as possible, starting with the high order digits. A little thought shows, however, that the machine could not necessarily designate any of the output digits exactly until it had access to the entire input string. For example, suppose the argument appearing on the input tape is in decimal notation, and begins with a decimal point followed by a long string of 3's:

$$x = 0.3333\dots$$

Then $\frac{1}{x}$ is approximately 3, but the machine cannot determine whether the first digit in the output should be a 2 or a 3 without examining the entire string of 3's to see if the digit following the string is greater than or is less than 3.

This suggests that we relax the requirement of an exact decimal expansion of the reciprocal, and be satisfied with an output which approximates the reciprocal. To do this, we can introduce an eleventh

decimal digit t , which stands for ten. We shall call a string of decimal digits including t a pseudo decimal expansion. Thus for example, $50 = 5t$, and $3 = 2.9t = 2.99t = 2.999t$, etc. Now when the machine is confronted with $.333\dots$ it proceeds to give out $2.999\dots$ until either a digit exceeding 3 occurs, in which case it proceeds as it would normally, or a digit less than 3 occurs, in which case it gives out a t followed by suitable decimal digits.

In general we can give the following definition.

20.9. Definition. The sequence $c_n c_{n-1} \dots c_0 \cdot d_1 d_2 d_3 \dots$ is a pseudo base b expansion for a number x provided the c_i 's and d_j 's are chosen from the set of digits $\{0, 1, \dots, b\}$ and

$$x = \sum_{i=0}^n c_i b^i + \sum_j d_j b^{-j}.$$

Before proving the theorem concerning reciprocals and pseudo expansions, it will be useful to prove a simpler proposition which avoids mention of the pseudo expansions.

20.10. Theorem: For every base $b \geq 2$ there is a multitape Turing machine such that for all d , when d digits representing the base b expansion for a number $A \neq 0$ (not necessarily an integer) appear on the input tape, the machine will calculate the d highest order digits (the highest order digit is by definition not zero) of $\frac{1}{A}$ within $d \cdot 2^{\sqrt[5]{\log d}}$ steps.

Proof. By applying Newton's method to find the zero of the function $y = \frac{1}{x} - A$ we obtain the iterative equation

$$20.11. \quad y_{n+1} = 2y_n - Ay_n^2.$$

If we set

$$20.12. \quad \epsilon_n = \frac{1}{A} - y_n,$$

then a simple calculation shows

$$20.13. \quad \epsilon_{n+1} = A\epsilon_n^2.$$

In carrying out the approximation of $\frac{1}{A}$, the machine does not use 20.11, because of the excessive time required to multiply all d digits of A . Instead, the machine considers initial segments of the representation of A whose length successively doubles. Say the n^{th} such segment consists of the 2^n highest order digits of A , and represents the number A_n . The machine calculates a sequence of successive approximations z_1, z_2, \dots to $\frac{1}{A}$ using the formula

$$20.14. \quad z_{n+1} = 2z_n - A_{n+1}z_n^2.$$

The right side of 20.14 is evaluated exactly, since the base b expansions of all terms involved terminate.

It remains to show how the machine uses the final approximation z_m to obtain the proper output. In all calculations, the machine ignores leading zeroes of the input and acts as though the radix point is immediately to the left of the highest order non-zero digit of A . We shall make this assumption in estimating the error. The machine

has no difficulty in locating the correct position for the radix point in $\frac{1}{A}$ after the proper string of output digits has been found.

Since A_{n+1} is a 2^{n+1} -digit approximation to A , we have

$$20.15. \quad \left| A - A_{n+1} \right| \leq b^{-2^{n+1}}.$$

Thus, setting

$$20.16. \quad \delta_n = \left| \frac{1}{A} - z_n \right|,$$

we have by observing 20.11 - 20.14,

$$\begin{aligned} 20.17. \quad \delta_{n+1} &\leq A \epsilon_n^2 + b^{-2^{n+1}} z_n^2 \\ &\leq \epsilon_n^2 + b^{-2^{n+1}+2} \end{aligned}$$

Hence the number of correct places approximately doubles with each iteration, so after m iterations, where m is approximately $\log d$, we have

$$\delta_m < b^{-d}.$$

The machine now obtains \widetilde{z}_m by rounding z_m to exactly $d - 1$ places.

(Notice that z_m has one digit to the left of the radix point). Thus

$$\left| \widetilde{z}_m - \frac{1}{A} \right| < b^{-d+1}.$$

The machine now compares $A \widetilde{z}_m$ with 1 to decide whether or not $\widetilde{z}_m > \frac{1}{A}$.

The final output B is determined by

$$B = \widetilde{z}_m - b^{-d+1} \quad \text{if } \widetilde{z}_m > \frac{1}{A}$$

and

$$B = \tilde{z}_n \quad \text{if } \tilde{z}_n \leq \frac{1}{A}$$

The estimation of the number of steps required to carry out the process is essentially the same as in the proof of theorem 17.1.

20.18. Theorem: For every base $b \geq 2$ there is a multitape Turing machine such that if a string of base b digits is written on the input tape, the first digit not zero, which represents a number A when the radix point is immediately to the left of the first digit, then the machine will give out $\frac{1}{A}$ in pseudo base b notation such that the first n digits are designated within $n 2^{5\sqrt{\log n}}$ steps, for all n .

Proof. The machine is similar to that of the previous theorem, but now the result of each iteration is transmitted immediately as output. Also the number A_n is now formed by adding b^{-2^n-3} to the number represented by the first $2^n + 3$ digits of the input A so that both the conditions

$$20.19. \quad \left| \frac{1}{A_n} - \frac{1}{A} \right| = \left| \frac{A - A_n}{A_n A} \right| \leq b^{-2^n-1},$$

and

$$20.20. \quad A_n > A$$

are satisfied. The sequence of successive approximations z_1, z_2, \dots to $\frac{1}{A}$ is again defined by 20.14, and the initial approximation z_1 is chosen to be close enough so that for all n ,

$$20.21. \quad \left| \frac{1}{A_n} - z_n \right| < b^{-2^n-1}.$$

Using z_n it is possible to determine a number w_n whose base b expansion terminates after 2^n places to the right of the radix point such that

$$20.22. \quad 0 \leq \frac{1}{A_n} - w_n \leq b^{-2^n}.$$

This is done by rounding z_n and comparing $A_n w_n$ with 1. Now it is easily proved by induction that the machine is able to designate additional digits and pseudo digits as output after each iteration so that after the n^{th} iteration the totality of digits designated forms a 2^n -place pseudo base b expansion for the number w_n . In fact, by 20.19, 20.20, and 20.22,

$$0 \leq \frac{1}{A} - w_n \leq b^{-2^n} + b^{-2^{n-1}}.$$

That is, w_n under-approximates $\frac{1}{A}$, but the error is not so big that it cannot be corrected by designating pseudo digits on the next two outputs.

The time estimates for the computation are the same as for the previous two theorems.

Of course, once it is possible to multiply and compute reciprocals in time $n^{5/\log n}$, it becomes possible to divide in the same time and so compute any rational function in the same time. The method can be extended to well-defined algebraic functions such as \sqrt{x} by applying Newton's method, but we shall not present these results in detail.

21. The effect on a computation of changing the radix base.

If one is to obtain a sensible classification of functions according to their minimum computation time, it is necessary to discuss how the computation time depends on the particular notation chosen for the arguments and values of the functions. In case the functions take integers into integers, the most interesting notations are the radix expansions, and so the problem becomes one of determining how the computation time depends on the choice of radix base. The first theorem in this section states that a multitape Turing machine can convert notation in one base to that of another in time $n^{6/\log n}$, so if the computation time of a function grows any faster than this then the choice of radix base has no effect on the time.

The same question of notation arises when one tries to classify real numbers according to their computational complexity. In this case, the choice of radix base can make an enormous difference in the computation time of a radix expansion because of the "ripple carry"; caused, for example, by a long string of 9's in decimal notation. In fact, theorem 21.18 below states that no matter how fast the computable function $T(n)$ grows, there is a real number x computable base 2 in time $n^{6/\log n}$, but not even computable base 3 in time $T(n)$. Obviously, computation time of the radix expansion is not always a good way to classify numbers.

Perhaps a more natural way to classify a number x as to complexity is to determine, given a number $\epsilon > 0$, how long it takes to produce a

rational number which approximates x to within ϵ . This idea is formalized as definition 21.6, and theorem 21.8 states that the resulting time functions do not depend heavily on the choice of radix base. We then prove theorem 21.12, which states that the base b approximability (21.6) of a number x and the ability to calculate a pseudo base b expansion for x lead to about the same time function, even though in the former case the Turing machine has available a preassigned error bound ϵ it must satisfy. This suggests that the minimum possible time needed to compute some pseudo base b expansion of x is a good way to classify x , and theorem 21.17 states that the resulting classification does not depend heavily on the choice of b .

In this section, "machine" means multitape Turing machine.

21.1 Theorem: For any pair of bases a and b there is a machine which converts a given integer N base a into its base b notation within $n 2^{6\sqrt{\log n}}$ steps, where n is the maximum of the lengths of the two notations for n .

Proof. The method is similar to the one used for multiplication. The computation is divided into successive iterations such that on the i^{th} iteration the parameters q and r are chosen to be q_i and r_i , as in the proof of 17.1. The r_i lowest order digits of the base a notation for N are divided into r pieces $\alpha_0, \alpha_1, \dots, \alpha_{r-1}$ of q digits each, so that

21.2.
$$N = \sum_{i=0}^{r-1} \alpha_i a^{qi}.$$

Next each α_i is converted to base b notation by calling the procedure again r times, and the numbers $a^q, a^{2q}, \dots, a^{r_q}$ are calculated using base b arithmetic. This computation is ~~more~~ easier because the base b notation for a^q is available from the previous iteration. Finally, 21.2 is evaluated using base b arithmetic, completing the i^{th} iteration.

To estimate the time required, we note that all multiplications on the i^{th} iteration can be performed by successive multiplications of pairs of q -digit numbers; and the total number of such q -digit multiplications is $O(r^2)$. Thus, if s_i is the total number of steps required for the conversion when $r = r_i$ and $q = q_i$, then by the inequality 17.7 and the fact the routine must call itself r_i times, we have

$$21.3. \quad s_i \leq D r_i^2 q_i 2^{4\sqrt{\log q_i}} + r_i s_{i-1}$$

for some constant D . We can now establish

$$21.4. \quad s_i \leq E q_{i+1} 2^{5\sqrt{\log q_{i+1}}}$$

for some constant E by induction on i . Certainly the inequality holds for $i = 1$, if E is chosen large enough. Assume now $i > 1$ and

$$s_{i-1} \leq q_i 2^{5\sqrt{\log q_i}}.$$

Then by applying 21.3 with $r_i = 2^{\lceil \sqrt{\log q_i} \rceil}$ (cf. 17.2) we have

$$\begin{aligned} 21.5. \quad s_i &\leq D r_i q_i 2^{5\sqrt{\log q_i}} + E r_i q_i 2^{5\sqrt{\log q_i}} \\ &\leq (D + E) q_{i+1} 2^{5\sqrt{\log q_i}}, \end{aligned}$$

where the second line uses the equation $q_{i+1} = r_i q_i$ given in 17.2.

Now in general,

$$\sqrt{x} < \sqrt{x + [\sqrt{x}]} - \frac{1}{4}$$

for sufficiently large x , so that

$$\sqrt{\log q_i} < \sqrt{\log q_i + [\sqrt{\log q_i}]} - \frac{1}{4} = \sqrt{\log q_{i+1}} - \frac{1}{4}$$

if q_i is sufficiently large. Therefore 21.4 follows from 21.5, provided E is large enough that

$$(D + E) 2^{-5/4} \leq E.$$

This completes the induction, and establishes 21.4. The time bound $n 2^{6\sqrt{\log n}}$ for the conversion follows from 21.4 by the same argument that proved theorem 17.1 from 17.7.

21.6. Definition.[†] A real number x is approximable base b in time $T(n)$ if there is a machine which, upon being given an integer n (say in base b notation) as input, will designate the base b expansion of a number x_n within $T(n)$ steps, where the base b expansion for x_n terminates after at most n places to the right of the radix point, and $|x - x_n| < b^{-n}$.

21.7. Terminology. The n^{th} place (base b) of a real number x is the n^{th} digit to the right of the radix point in the base b expansion of x . An n -place number is one whose base b expansion terminates after n places to the right of the radix point.

[†] This definition was suggested to the author by Michael Fischer.

21.8. Theorem: Suppose a and b are radix bases and $T(n)$ is a time function such that

$$T(n) \geq n 2^{6\sqrt{\log n}}.$$

If a real number x is approximable base a in time $T(n)$, then x is approximable base b in time $T(\{n \log_a b\} + 1)$, where $\{y\}$ is the least integer not smaller than y .

Proof. Suppose the machine M_1 approximates x base a in time $T(n)$ in the sense of definition 21.6. The machine M_2 then proceeds as follows. Upon being given n , M_2 calculates $m = \{n \log_a b\} + 1$ (for example, by finding the number of digits in b^n base a and possibly adding 1) and gives m as input to M_1 . The machine M_1 then designates a number x_m to m places base a such that

$$21.9. \quad |x - x_m| < a^{-m} \leq \frac{1}{a} b^{-n}.$$

This is done within $T(m)$ steps.

The output x_m can be written in the form $\frac{N}{a^m}$, for some integer N .

Now M_2 converts N to base b notation and calculates a^m base b . This requires $O(n 2^{6\sqrt{\log n}})$ steps by the proof of the previous theorem.

Finally, M_2 computes the quotient $\frac{N}{a^m}$ base b rounded exactly to n base b places, obtaining a number z_n such that

$$21.10. \quad |z_n - x_m| \leq \frac{1}{2b^n}.$$

A slight modification at the end of the proof of theorem 20.10 shows this can be done within $n 2^{5\sqrt{\log n}}$ steps. By 21.9 and 21.10 we have

$|x - z_n| < b^{-n}$, so that z_n is the desired approximation. Since $T(n) \geq n 2^{6\sqrt{\log n}}$, the total time required is $O(T(m)) = O(T(\{n \log_a b\} + 1))$ steps. The time can be reduced to $T(m)$ by the speed-up theorem in [6].

21.11. Definition.[†] An increasing function $T(n)$ on the positive integers is real-time countable if there is some multitape Turing machine which generates a sequence a_1, a_2, \dots in real time such that $a_i = 1$ if i is in the range of $T(n)$, and $a_i = 0$ otherwise.

21.12. Theorem: Suppose $T(n)$ is real-time countable, and let x be a real number. Then

(i) If a pseudo base b expansion of x (20.9) is computable in time $T(n)$, then x is approximable base b in time $T(n)$, and

(ii) If x is approximable base b in time $T(n)$, then a pseudo base b expansion of x is computable in time $T(n + 1)$.

Proof. (i) To obtain an n -place approximation to x from the first n digits of a pseudo expansion of x , add b^{-n} to the pseudo expansion and convert the first n digits to standard base b notation.

(ii) Now suppose x is approximable base b in time $T(n)$ by a machine M_1 . Let the sequence $a_1 \leq a_2 \leq a_3 \leq \dots$ be defined by the condition a_m is the largest integer such that $T(a_m) \leq 2^m$. The machine M_2 proceeds to find a pseudo base b expansion by successive iterations. By the end of the m^{th} iteration, the machine has designated an $(a_m - 1)$ -place pseudo notation for a number u_m such that

[†] This definition was introduced by Yamada in [16].

$$21.13. \quad 0 \leq x - u_m \leq b^{-a_m+1} + b^{-a_m}.$$

On the m^{th} iteration the number a_m is given to M_1 as input, thus obtaining an a_m -place approximation y to x which satisfies

$$21.14. \quad |x - y| < b^{-a_m}.$$

Next y is used to find an $(a_m - 1)$ -place approximation z to x satisfying

$$21.15. \quad 0 \leq x - z \leq b^{-a_m+1} + b^{-a_m}.$$

In fact, z is $(y - b^{-a_m})$ truncated to $a_m - 1$ places. Now if $z \leq u_{m-1}$ (where u_{m-1} is the previous output) then the new output consists of enough zeroes to bring the number of output places to $a_m - 1$; so $u_m = u_{m-1}$. Then $z \leq u_m \leq x$, so 21.13 follows from 21.15 in this case. On the other hand, if $z > u_{m-1}$, then the new output consists of the right digits and pseudo digits to make $u_m = z$ (so again 21.13 follows from 21.15). That such pseudo digits exist follows from the inequality

$$21.16. \quad u_{m-1} < z \leq u_{m-1} + b^{-a_{m-1}+1} + b^{-a_{m-1}}$$

and the facts that u_{m-1} has $a_{m-1} - 1$ base b places and z has $a_m - 1$ base b places.

The time required for the m^{th} iteration is $O(T(a_m)) = O(2^m)$ steps. Now given an integer n , let m be the smallest integer such that $T(n + 1) \leq 2^m$. Thus $n + 1 \leq a_m$, so that the first n pseudo digits are designated by the end of the m^{th} iteration. Thus, for some constant C

the total number of steps required to designate the first n pseudo digits does not exceed

$$\sum_{k=1}^m C 2^k < 2C 2^m < 4CT(n+1).$$

The constant $4C$ can be reduced to 1 by the speed-up theorem.

21.17. Theorem: Given bases a, b and a time function $T(n)$ satisfying

$$T(n) \geq n 2^{6\sqrt{\log n}},$$

if a pseudo base a expansion of a real number x is computable in time $T(n)$, then a pseudo base b expansion is computable in time $T(\{(n+1) \log_a b\} + 1)$.

Proof. We may assume $T(n)$ is real-time countable, because in fact the actual time used by the first machine is real-time countable and does not exceed $T(n)$. Thus theorem 21.17 is an immediate corollary of theorems 21.8 and 21.12.

Finally, to justify the use of the notions of approximability and pseudo expansion, we show there is no analog of theorems 21.8 and 21.17 for the case of ordinary radix notation.

21.18. Theorem: For every recursive function $T(n)$, there is a real number x such that the base 2 expansion of x is computable in time $n 2^{6\sqrt{\log n}}$, but the base 3 expansion of x is not computable in time $T(n)$.

Proof. We may assume $T(n)$ is increasing, real-time countable, and satisfies $T(n) > n$, for if $T(n)$ does not satisfy these conditions there is a larger function that does. Now define $U(n)$ by the primitive recursion

$$U(1) = 1$$

$$U(n+1) = (T(U(n) + 1))^3.$$

Then $U(n)$ is readily verified to be real-time countable. Further,

$$\lim_{n \rightarrow \infty} \frac{(T(U(n-1) + 1))^2}{U(n)} = 0,$$

so by Corollary 9.1 of Hartmanis and Stearns [6] there is a sequence

$\langle \epsilon_n \rangle$ with values in $\{-1, 1\}$ such that ϵ_n is computable in time $U(n)$,

but not in time $T(U(n-1) + 1)$. Assume $\epsilon_1 = 1$, and set

$$x = \sum_{n=1}^{\infty} \frac{\epsilon_n}{3^{U(n)}}.$$

If x were computable base 3 in time $T(n)$, then the digit in position $U(n-1) + 1$ would be available within $T(U(n-1) + 1)$ steps. But $\epsilon_n = 1$ if and only if this digit is 0, and $\epsilon_n = -1$ if and only if this digit is 2. Hence $\langle \epsilon_n \rangle$ would be computable in time $T(U(n-1) + 1)$, contrary to assumption. Thus x is not computable base 3 in time $T(n)$.

To see that x is computable base 2 in time $n 2^{6\sqrt{\log n}}$, we note that $U(n)$ is real-time countable, and hence x is approximable base 3 in real time. By theorem 21.8, x is approximable base 2 in time $n 2^{6\sqrt{\log n}}$. Hence by the lemma below, x is in fact computable base 2 in time $n 2^{6\sqrt{\log n}}$, as required by the theorem.

21.19. Lemma. Suppose A and B are positive integers such that B is odd and $B < 2^m$. Then the base 2 expansion of $\frac{A}{B}$ nowhere contains m consecutive 1's.

Proof. Suppose, to the contrary, the binary expansion of $\frac{A}{B}$ has m consecutive 1's. Then there are integers A_1, n such that

$$\frac{A}{B} = \frac{A_1}{2^n} + \frac{2^n - 1}{2^{m+n}} + \frac{\theta}{2^{m+n}},$$

where $0 \leq \theta < 1$. Thus

$$2^{m+n}A = A_1B2^m + 2^mB - (1 - \theta)B.$$

Hence 2^m divides the integer $(1 - \theta)B$, which is impossible because $0 < (1 - \theta) \leq 1$ and $0 < B < 2^m$.

BIBLIOGRAPHY

1. Atrubin, A. J. A one-dimensional real-time iterative multiplier. IEEE Trans. on Elec. Comp. EC-14, No. 3 (1965), 394-399.
2. Becvar, Jiri. Real-time and complexity problems in automata theory. To be published.
3. Cobham, A. The intrinsic computational difficulty of functions. Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science, North-Holland Publishing Co., Amsterdam, 24-30.
4. Cole, S. N. Real-time computation by iterative arrays of finite-state machines. Doctoral Thesis, and Report 8L-36, Computation Laboratory, Harvard University, 1964.
5. Fischer, P. C. Generation of primes by a one-dimensional real-time iterative array. J. Assoc. Comp. Mach. 12, No. 3 (1965), 388-394.
6. Hartmanis, J., and R. E. Stearns. On the computational complexity of algorithms. Trans. Amer. Math. Soc. 117 (1965), 285-306.
7. Hennie, F. C. On-line Turing machine computations. IEEE Trans. on Elec. Comp. EC-15, No. 1 (1966), 35-44.
8. Hennie, F. C. One-tape, off-line Turing machine computations. Info. and Control 8 (1965), 553-578.
9. Hennie, F. C., and R. E. Stearns. Two-tape simulation of multitape Turing machines. J. Assoc. Comp. Mach. (to appear).
10. Higman, G. Subgroups of finitely presented groups. Proc. Royal Soc. A, 262 (1961), 455-475.
11. McNaughton, R. The theory of automata, a survey. Advances in Computers, vol. 2, F. L. Alt ed., Academic Press, New York, 1961, 379-421.
12. Rabin, M. O. Real-time computation. Israel J. of Math., Dec., 1963, 203-211.
13. Shepherdson, J. C., and H. E. Sturgis. Computability of Recursive Functions. J. Assoc. Comp. Mach. 10 (1963), 217-255.

14. Toom, A. L. The complexity of a scheme of functional elements realizing the multiplication of integers. Sov. Math., Translations of Dokl. Akad. Nauk. SSSR, 4, No. 3 (1963), 714-716. (Original Vol. 150, No. 3 (1963), 496-498).
15. Winograd, S. On the time required to perform multiplication. Research Paper RC-1564, March 8, 1966, IBM Watson Research Center, Yorktown Heights, New York.
16. Yamada, H. Real-time computation and recursive functions not real-time computable. IRE Trans. on Elec. Comp. EC-11 (1962), 753-760.