The Aliquot Project:
An Application of Job Chaining in Number Theoretic Computing

M. C. Wunderlich
Northern Illinois University
DeKalb, Illinois 60115

INTRODUCTION. This paper is divided into two parts. Part 1 first presents an old and charming number theoretic conjecture which has been generally believed or at least respected by the mathematical community for over 70 years. A probabilistic argument is then detailed which supports the opposite of this conjecture. A vast amount of computing has been recently done at various universities all over the world in order to investigate the plausibility of this conjecture and it was largely as a result of these computations that the negative argument was formulated. Part II of the paper discusses these computations and describes some of the problems which were encountered in this project. Not only were many hours of machine time consumed but hundreds of man-hours were spent book-keeping and "terminal-watching" because of the non-homogeneous character of the project. Finally, a software system is described which, when fully implemented, will provide the user with controlled but automatic job submission permitting number theoretic projects such as this to be carried out with much less constant attention.

Part I. The mathematics.

An <u>aliquot sequence</u> (abbreviated AS) is a sequence of positive integers $n_0, n_1, \ldots$ for which $n_k = s(n_{k-1})$ where $s(n) = \sigma(n) - n$ and, as usual,

$$\sigma(n) = \sum_{d|n} d.$$

We repeatedly use the fact that σ is multiplicative and $\sigma(p^\gamma) = 1 + p + p^2 + \ldots + p^\gamma$. There are three kinds of aliquot sequences:

<u>terminating</u>: $n_k = 1$ for some k

<u>periodic</u>: $n_{k+t} = n_k$ for some t and for k sufficiently large (for example $s(6) = 3.4-6 = 6$)

<u>infinite</u>: $\lim_{k\to\infty} n_k = \infty$.

Catalan (1887) and Dickson (1913) have conjectured that infinite aliquot sequences do not exist. Recent computations by Guy, Selfridge, and Wunderlich show that of all the aliquot sequences for which $n_0 < 10{,}000$, all but 98 are known to terminate. In January, 1973, Richard Guy and John Selfridge conjectured that infinitely many aliquot sequences, perhaps almost all with n_0 even, are infinite. We present in this section the Guy-Selfridge argument.

Table 1: Example of a "long" terminating AS.

k	n(k)
0	2880
32	123 709593008
55	7447648
69	668429258
99	6677260
154	5108232 531623332
203	26799040
224	177 841798874
251	124124
325	36445367 869087816
393	277
394	1

The sequence beginning with $n_0 = 2880$ demonstrates the behavior of many aliquot sequences. The table above lists all the terms which are relative maxima or relative minima. Table 2 is a detailed look at a segment of the sequence beginning with 1074. The right hand column of the table is the unique factorization of $n(k)$ into primes. (In our notation, exponents are contained in parentheses and a period is used to denote multiplication whenever necessary. Thus 2(3)5.7.977 means $2^3 \cdot 5 \cdot 7 \cdot 977$.) The reader should note that each of the first 25 terms in the table contains a single power of 2 and no 3's whatsoever. On the other hand, all the other small primes appear about the "right" number of times. This pattern seems to coincide with terms which are steadily decreasing in magnitude. On the other hand, the last 19 terms in the segment contain a single power of 2, at least one power of 3 and a random representation of the other primes. This pattern is associated with terms which are steadily increasing but not as rapidly as the earlier terms are decreasing. We shall fully explain this phenomenon.

TABLE 2 -- A "Snapshot" of AS 1074*

K	N(K)	FACTORIZATION OF N(K)
310	436546562 900403622	2.218273281450201811
311	218273281 450201814	2.17.29.43.1249.9929.415133
312	149021921 791798186	2.22316761.3338789213
313	74510970 979217018	2.1747.21325406691247
314	37319461 709687494	2.2687767.6942465941
315	18659751 690304874	2.47.198507996705371
316	9925399 835268694	2.7(2)31.3267083553413
317	7952081 369012714	2.199.22861.873982063
318	4036505 399288086	2.2018252699644043
319	2018252 699644046	2.89.2503.4529963369
320	1044364 935545554	2.7.71.139.191.39574709
321	793826 522348846	2.7.56701894453489
322	567018 944534914	2.9176053.30896669
323	283509 592485626	2.13.6917.24749.63697
324	174559 350692374	2.331.3491.75532747
325	88145 963899562	2.67.173.293.12977287
326	47282 729665750	2.5(3)19.9954258877
327	45889 133432330	2.5.19.241521754907
328	41058 698334550	2.5(2)13.29.461.4724903
329	44205 596276330	2.5.4420559627633
330	35364 477021082	2.11.1607476228231
331	22504 667195270	2.5.13.173112824579
332	21119 764598890	2.5.139.15194075251
333	17169 305036150	2.5(2)1483.231548281
334	14787 136459234	2.7393568229617
335	7393 568229620	2(2)5.19(2)1024039921
336	8993 118602224	2(4)19.29582626981
337	9348 110126616	2(3)3.152407.2555687
338	14022 327675624	2(3)3.73.241.5647.5881
339	21673 548325656	2(3)3(2)23.107.199.614657
340	40460 999578344	2(3)3(2)561958327477
341	69120 874279866	2.3.67.267601.642533
342	71184 926630022	2.3.83.142941619739
343	72900 226067898	2.3.157.967.4651.17207
344	74020 746222150	2.3.5(2)7.13.2551.2125741
345	152002 020061626	2.3.7(2)977.18713.28279
346	202028 731286214	2.3.7.4810207887767
347	259751 225939514	2.3.79.149.193.19056173
348	272602 050924486	2.3.29.2857.548364877
349	291599 604752154	2.3.48599934125359
350	291599 604752166	2.3(2)17.65003.14659937
351	377374 531293738	2.3(2)11.1905931976231
352	514601 633582838	2.3(2)101.127.44059.50587
353	620322 290876682	2.3(2)41.840545109589
354	756490 598631738	2.3(3)463.30257203369
355	928230 485009862	2.3.14731.18671.562477
356	928459 246951482	2.3.9677.15990824411
357	928651 136960550	2.3.5(2)31.43.59921.77509
358	1504054 894174170	2.3.5.13.31.317.392444389
359	2521407 331366950	2.3.5(2)13.5477.236083513

* Computed by H. J. Godwin, see [6]

Let the set of all positive integers be partitioned into sets S_0, S_1, S_2 and S_3 as follows:

S_0; set of all odd integers

S_1; $n = 2k$ where $(k,6) = 1$

S_2; $n = 2.3.k$ where k is odd

S_3; $n = 2^2.k$.

If $C(n)$ is a condition on n, we will denote with $N_x\{C(n)\}$ the number of positive integers $n \le x$ for which $C(n)$ is true. Thus for $i = 0,1,2,3$, we define the function

$$(1) \qquad B_i(x) = \frac{N_x\{n \in S_i; s(n) \notin S_i\}}{N_x\{n \in S_i\}}$$

which, loosely stated, measures the probability that a term of an aliquot sequence whose order of magnitude is x will "break" out of the set S_i. We will call these functions "break probabilities".

We also define the function A_i to be the average order of the function $s(n)/n$ taken over the set of all $n \in S_i$. Formally, it is defined to be

$$(2) \qquad A_i = \lim_{x\to\infty} \frac{\sum\limits_{\substack{n<x\\ n\in S_i}} \frac{s(n)}{n}}{\sum\limits_{\substack{n<x\\ n\in S_i}} 1} = \frac{\sum\limits_{\substack{n<x\\ n\in S_i}} \frac{\sigma(n)}{n}}{\sum\limits_{\substack{n<x\\ n\in S_i}} 1} - 1.$$

This function measures the average growth of an aliquot sequence term lying in one of the sets S_i. That the limit exists is based on the following lemma whose proof we omit.

<u>Lemma 1</u>: If k is a residue mod p, then

$$\sum_{\substack{n<x\\ n\equiv k(p)}} \frac{\sigma(n)}{n} = Cx + O(\log x)$$

for some constant C.

For the remainder of this section, we adopt the usual notation $a|n$ for "a divides n". If p is a prime, $p^\alpha | n$ but $p^{\alpha+1} \nmid n$, we will write $p \| n$.

<u>Lemma 2</u>:

(a) $B_0(x) = O(1/\sqrt{x})$

(b) $B_1(x) \sim \pi^2/6 \log x \doteq 1.64492/\log x$

(c) $B_2(x) \sim \pi^2/24 \log x \doteq .41123/\log x$

Proof: (a) follows from the fact that if k is odd, $s(k)$ is even if and only if k is a square. ($\sigma(p^a) = 1 + p + p^2 + \ldots + p^a$). To prove (b), we let $(k,6) = 1$ and let $M = s(2k) = 3\sigma(k) - 2k$. Clearly $3\nmid M$. $2 \| M$ if and only if $2^2|\sigma(k)$ and $2^2 \nmid \sigma(k)$ if and only if

(i) k is a square (neglect this)
(ii) $k = Sp$ where S is a square relatively prime to 6 and p is a prime $\equiv 1 \pmod 4$.

It is well known that the number of primes $\le x$ which are congruent to $1 \bmod 4$ is asymptotic to $x/2\log x$. Using this we obtain

$$(3) \qquad N_x\{n = Sp,\ S \text{ square},\ (S,6) = 1,\ p \text{ prime},\ p \equiv 1(4)\} \sim \frac{\pi^2 x}{18 \log x}$$

so $N_x\{n \in S_1,\ s(n) \notin S_1\} \sim \dfrac{\pi^2 x}{36 \log x}$ and so from (1), $B_1(x) \sim \dfrac{\pi^2 x}{36 \log x} \Big/ \dfrac{x}{6} = \dfrac{\pi^2}{6 \log x}$. To prove (c), let k be odd and let

$$M = s(2.3^a.k) = 3\sigma(3^a)\sigma(k) - 2.3^a.k.$$

Again we discount the case where k is a square; thus we assume that $2|\sigma(k)$. Therefore $2^2 \nmid M$ whenever $\sigma(3a)$ is odd and $k = Sp$, S a square and $p \equiv 1 \pmod 4$. $\sigma(3^a)$ is odd whenever a is even, so we get using (3)

$$N_x\{n \in S_2, s(n) \notin S_2\} \sim \sim \frac{\pi^2 x}{18 \log x}\left(\frac{1}{2.3^2} + \frac{1}{2.3^4} + \ldots\right) \sim \sim \frac{\pi^2 x}{288 \log x}$$

and $N_x\{n \in S_2\} \sim x/12$.

Thus $B_2(x) \sim \dfrac{\pi^2 x}{288 \log x} \Big/ \dfrac{x}{12} = \dfrac{\pi^2}{24 \log x}$.

To obtain values for A_0, A_1, and A_2, we need average order results for $\sigma(n)/n$ taken over various sets. General theorems of this sort can be obtained, but we need only two special results whose proofs we can sketch.

<u>Lemma 3</u>:

(a) $$\sum_{\substack{n\le x\\ n \text{ odd}}} \frac{\sigma(n)}{n} \sim \frac{\pi^2 x}{16}$$

(b) $$\sum_{\substack{n\le x\\ (n,6)=1}} \frac{\sigma(n)}{n} \sim \frac{\pi^2 x}{27}$$

<u>Sketch of proof</u>: We use a result of Hardy and Wright to write

$$(4) \qquad \frac{\pi^2 x}{6} + O(\log x) = \sum_{n\le x} \frac{\sigma(n)}{n} =$$

$$= \sum_{\substack{k>0 \\ 2^k<x}} \; \sum_{\substack{n\le x/2^k \\ n \text{ odd}}} \frac{\sigma(2^k n)}{2^k n} .$$

We then use lemma 1 to assert that a C exists for which

$$\sum_{\substack{n\le x \\ n \text{ odd}}} \frac{\sigma(n)}{n} = Cx + O(\log x)$$

and we substitute this for the right hand sum of (4). [Note that $\sigma(n)$ and $\sigma(n)/n$ are multiplicative.] One can thus solve for C and get (a) with the error $O(\log^2 x)$. This result is then used in a similar way to obtain (b).

Lemma 4: $A_0 = \frac{\pi^2}{8} - 1$, $A_1 = \frac{\pi^2}{6} - 1$, and

$$A_2 = \frac{11\pi^2}{48} - 1.$$

Proof: The value of A_0 follows directly from (2) and lemma 3. To obtain A_1, we estimate

$$\sum_{\substack{n\le x \\ n\equiv \pm 2 (12)}} \frac{\sigma(n)}{n} = \sum_{\substack{n\le x/2 \\ n\equiv \pm 1 (6)}} \frac{\sigma(n)}{n} \cdot \frac{\sigma(2)}{2} \sim$$

$$\sim \frac{3}{2} \sim \frac{\pi^2 x}{2.27} = \frac{\pi^2 x}{36}$$

and the result follows from (2). To estimate A_2, we write

$$\sum_{\substack{n\le x \\ n\equiv 6 (2)}} \frac{\sigma(n)}{n} = \frac{3}{2}\left(\sum_{\substack{n<x/2 \\ n \text{ odd}}} \frac{\sigma(n)}{n} - \sum_{\substack{n<x/2 \\ n\ \pm 1 (6)}} \frac{\sigma(n)}{n} \right) \sim$$

$$\sim \frac{11\pi^2}{12.48}$$

and the result follows from (2).

One could also compute $A_3 = \frac{11\pi^2}{48} - 1$, but the result would be misleading. The actual "average growth" of terms n such that $2^2|n$ would not reflect the average order of $s(n)/n$ since the break probability would differ for each class of integers N_a for which $2^a||n$. The following computation, however, indicates that the general tendency for numbers in S_3 would be to grow upwards. We let $n = 2^a k$ for $a \ge 2$ and k odd and write

$$\frac{\sigma(2^a k)}{2^a k} = \frac{\sigma(2^a)}{2^a} \cdot \frac{\sigma(k)}{k} > \frac{7}{4}\ \frac{\sigma(k)}{k} ,$$

but the average order of $\frac{\sigma(k)}{k}$ over k odd is $\frac{\pi^2}{8}$ and

$$\frac{7}{4} \cdot \frac{\pi^2}{8} \doteq 2.15897 > 2.$$

The results thus far can be summarized in the following table:

j	$B_j(x)$	A_n
0	$1/\sqrt{x}$	$\pi^2/8-1$: .23369
1	$\pi^2/6 \log x$	$\pi^2/6-1$: .64493
2	$\pi^2/24 \log x$	$11\pi^2/48-1$: 1.26178
3	≤ 1	> 1

Since for all n, $s(n) \in S_0$ only if n is a square or twice a square, aliquot sequences with large terms will be dominated by terms in S_1, S_2, and S_3. The Guy-Selfridge argument is based on the "expected-value" behavior of an aliquot sequence dominated by terms in S_1 and S_2. Terms in S_3 can only help matters.

Now, suppose ℓ_1 consecutive terms occur of type 1, after which n has been reduced to n . The geometric mean of the size of the term is $n^{(1+\alpha_1)/2}$. The "mean probability" of this type breaking is

$$\frac{\pi^2}{6 \log n^{(1+\alpha_1)/2}} = \frac{\pi^2}{3(1+\alpha_1) \log n}$$

so the expected length of this string, ℓ_1, is

$$\ell_1 = \frac{3(1+\alpha_1)\log n}{\pi^2} .$$

But, on average, $s(n) = nA_1$ where $A_1 = \frac{\pi^2}{6} - 1$. Thus

$$n\left(\frac{\pi^2}{6} - 1\right)^{\ell_1} = n^{\alpha_1}$$

$$\log n + \frac{3(1+\alpha)\log n}{\pi^2} \log\left(\frac{\pi^2}{6} - 1\right) = \alpha_1 \log n$$

$$\pi^2 + 3(1+\alpha_1)\log\left(\frac{\pi^2}{6} - 1\right) = \pi^2\alpha_1$$

$$\alpha_1 = \frac{\pi^2 + 3\log\left(\frac{\pi^2}{6} - 1\right)}{\pi^2 - 3\log\left(\frac{\pi^2}{6} - 1\right)} = .76479$$

$$\ell_1 = \frac{3(1+\alpha_1)}{\pi^2} \log n = .53643 \log n.$$

For type 2 sequence,

$$\ell_2 = \frac{12(1+\alpha_2)\log n}{\pi^2}$$

and on average $s(n) = nA_2$ where $A_2 = \frac{11\pi^2}{48} - 1$

so in a string of ℓ_2,

$$n\left(\frac{11\pi^2}{48} - 1\right)^{\ell_2} = n^{\alpha_2}$$

or

$$\log n + \frac{12(1+\alpha_2)\log n}{\pi^2} \log\left(\frac{11\pi^2}{48} - 1\right) =$$

$$= \alpha_2 \log n,$$

$$\pi^2 + 12(1+\alpha_2)\log\left(\frac{11\pi^2}{48} - 1\right) = \pi^2\alpha_2,$$

$$\alpha_2 = \frac{\pi^2 + 12\log\left(\frac{11\pi^2}{48} - 1\right)}{\pi^2 - 12\log\left(\frac{11\pi^2}{48} - 1\right)} = 1.78831,$$

$$\ell_2 = \frac{12(1+\alpha_2)}{\pi^2} \log n = 3.39018 \log n.$$

If an aliquot sequence began with terms in S_2, continued for ℓ_1 terms, broke into terms in S_1, continued for ℓ_2 terms, the size would be

$$(n^{\alpha_2})^{\alpha_1} = n^{1.36768}.$$

The total number of terms would be approximately

$$3.39 \log n + .536 \log(n^{1.79}) = 4.35 \log n.$$

Thus the expected behaviour of an aliquot sequence dominated by the type 1 and 2 terms is for geometrical growth. To repeat, type 3 terms can only help, the type 0 "down driver" is entered with prob. $O(1/\sqrt{x})$ and hence can be neglected.

The above argument is an over-simplified analysis of a very complicated two dimensional Markov process. The set of integers should be partitioned into a larger collection of sets, each associated with its own "driver". For a discussion of which drivers should be included, see Guy and Selfridge [7]. A break matrix can be constructed giving the probability of a sequence going from one driver to another - these probabilities are, of course, functions of x, the magnitude of the term. Each driver has its own growth distribution describing how rapidly terms are increasing or decreasing in each category. In order for such a model to be convincing, one should compare it with driver statistics collected from a large number of computed aliquot sequences. We will now turn our attention to the problems associated with the computing of aliquot sequences.

Part II. <u>The Computation</u>. The following algorithm replaces the positive integer N with $\sigma(N) - N$, the next term in the aliquot sequence.

<u>Algorithm A</u>

1. (Initialize) Set $S \leftarrow 1$; set $M \leftarrow N$.
2. Perform steps 2.1 through 2.4 while M is composite.
   - 2.1 Search for p, the smallest prime such that $p|M$. If no such p can be found, the program fails.
   - 2.2 Let $M \leftarrow M/p$ and $F \leftarrow 1 + p$.
   - 2.3 Do while $p|M$; $F \leftarrow 1 + Fp$; $M \leftarrow M/p$; End;
   - 2.4 $S \leftarrow S \cdot F$;
3. If $M > 1$, set $S \leftarrow S \cdot (1 + M)$
4. Let $N \leftarrow S - N$;

For each $p^\gamma||N$, step 2.3 accumulates the value $\sigma(p^\gamma) = 1 + p + p^2 + \ldots + p^\gamma$ and step 2.4 accumulates the product of these values for all $p|N$. In practice, a program would test for $n_k = n_{k-1}$, $n_k = n_{k-2}$ and perhaps even $n_k = n_{k-4}$ since many aliquot sequences are known of period 1, 2 and 4. There is also a well known sequence of period 43 which a sophisticated program can look for. Also, since this program is designed to collect driver statistics, steps 2.1/2.3 must be elaborated upon in order to determine which driver is in effect. These are all easy problems to solve, however, and we omit discussing them in this narrative. The purpose of this paper is to discuss some computational problems which are unique to number theoretic computing and to suggest some novel ways to solve them.

The main difficulty arises in steps 2 and 2.1, the only two places where failure can occur. The condition in step 2 is tested by using Fermat's test for compositeness. If p is a prime, then $a^{p-1} \equiv 1 \mod p$ for any integer a. Thus, if $a^{N-1} \not\equiv 1 \mod N$, N is surely composite. Furthermore, if N is composite, it is very improbable that $a^{N-1} \equiv 1 \mod N$. The exponent a^{N-1} can be computed in $O(\ell og\ N)$ operations so we have a very inexpensive method for verifying compositeness. If $a^{N-1} \equiv 1 \mod N$ however, we should really not proceed until we have proved N prime.

Proofs of primality for large numbers N can be long, tedious and fraught with danger. A large collection of theorems related to primality proofs can be read in [1]. A complete algorithm for efficient prime testing appears in [14] and we have a working program which is essentially based on that algorithm. Primality proofs are produced by that program by obtaining partial or complete factorizations of $N - 1$ and $N + 1$. In most cases, when N is less than 40 digits, these factors can be found by the simple divide and factor algorithm [9] which attempts to divide $N \pm 1$ by the primes 2, 3, 5,... until enough factors are collected to obtain a proof. The entire proof usually takes less than 12 seconds on a 360/67 computer and so the program can be easily invoked as a sub-procedure of the main aliquot generating program. However, there are cases in which $N + 1$ and $N - 1$ only have a few small divisors--insufficient for a proof. If after dividing out these factors, one or more of the resulting cofactors fails Fermat's test for compositeness, then the program continues recursively by attempting a primality proof of the cofactor. If both cofactors are composite, a more sophisticated proof can often be obtained by using not only the factors of $N \pm 1$ but also a bound b for which $p|N \pm 1 \Rightarrow p > b$. The program can compute the smallest such bound b which would be sufficient to provide a proof if no additional factors were discovered. If this value is too high, the routine must send back to the main program the message, "We're sorry. Your number is most assuredly prime but we cannot produce a proof of such without possibly investing n minutes of computer time," Where n may be anywhere from one or two to several hundred.

The main program now has basically two options. It can proceed with its computation after noting that the number in question has been merely proved "pseudoprime." Then at the end of the semester when more computer time is available or when the economy improves and a new computer is obtained at our university these difficult primality proofs can be "cleaned up" and the PSP's denoting pseudoprimes can be removed from the aliquot sequence output. Of course, there is that minute possibility that a pseudoprime may turn out to be composite and all the subsequent terms in the aliquot sequence would be incorrect. Thus the other more conservative option would be for the main program to halt all operations until a complete primality proof is obtained. We adopted the former philosophy in our computations. In fact, we employed a very crude primality proving program for the bulk of the aliquot project while we were developing the more sophisticated program described above. By now, all PSP's have been removed from our output and not a single composite pseudomprime was discovered.

The other difficulty arises in step 2.1 of the algorithm. We have already assertained that M is composite but all of our efforts to find a factor of M have proved unsuccessful and we are faced with the possibility of exceeding our estimated computer time for the job. At this point, there is really nothing the main program can do but terminate the job. There is no such thing as a pseudo-factorization which would enable the program to carry on. Since both difficulites in this algorithm involve the factorization of large numbers, we shall digress for a moment to discuss the general question, "How long does it take to factor a number."

It depends very strongly on the method you wish to use, and having chosen the method it may depend on the number you are factoring. In order to completely factor n using the divide and factor method discussed earlier, one must divide by the primes which are less than the second largest prime dividing n, which we will denote by $F_2(n)$. D. E. Knuth and L. T. Pardo [9, 10] have recently analysed the distribution of the values $F_2(n)$ for $n \leq x$ and their results show that for about half of the integers $n \leq x$ $F_2(n) \leq x^{.21172}$. Thus, for about half the 36 digit numbers we factor, a complete factorization can be gotten by dividing up to 42 million. On our computer, this would take about 7.3 minutes of computer time. To put it another way, we can factor a 36 digit number up to 1,000,000 in about 10 seconds. Again, using the Knuth-Pardo tables, we completely factor about 37% of the 36 digit numbers by dividing up to 1,000,000. However, for 25% of the 36 digit numbers,

$F_2(n) > n^{.29153} \doteq 31 \times 10^9$. To factor this high would take about 90 hours of machine time. There are two other recently discovered methods of factoring which shorten this time considerably. Pollard's Monte Carlo method requires roughly $\sqrt{p}$ operations where p is the second largest prime dividing n, but each operation is at least 100 times as costly as the single division counted in the divide and factor algorithm. Never-the-less, in the example cited just above, only 176823 operations would be required. This should be possible in a few minutes with a good program. Even with the Pollard Monte Carlo method, one 36 digit number in 20 will take over 4 hours of computer time. (Another interesting method was discovered earlier by Pollard which requires p operations where p is the largest prime dividing $q - 1$ where $q|n$. [12,13]) For two good discussions of factorization, see [8,10].

There are other methods of factoring which do not depend at all on the particular distribution of the factors but rather depend only on the size of the number. The best example of this type of factorizer is the continued fraction method developed by John Brillhart [11]. Although performance characteristics have not been theoretically obtained for this method, our experience with the program suggests the timing formula

$$\text{TIME} = .0003324\ N^{.1574} \tag{4}$$

where TIME is the estimated time in 360/67 computer minutes to factor the number N. This formula yields the following values.

N	TIME (minutes)
10^{16}	.110
10^{20}	.467
10^{24}	1.992
10^{28}	8.489
10^{32}	36.18
10^{36}	154.2

The formula was obtained by doing a linear regression analysis of log T versus log N on 100 actual factorizations using the program in which N was factored in T minutes. The 100 observations ranged in size from 10^{16} to 10^{36}. The fit was quite good producing a correlation coefficient of 0.968. For each of these observations the ratio ACTUAL TIME/PREDICTED TIME was computed. The largest and second largest of these ratios were 2.84 and 2.01 respectively and the smallest ratio was .492. The computed mean was 1.05 and the standard deviation was .344. Thus the timing estimate will almost always predict the actual time to within a factor of 2.

The point we are making is that any general factoring subroutine presents the calling program with a hopeless problem. There is absolutely no way the main program can know how much computer time it will take to factor a number. It may take less than 10 seconds or over one hour. Clearly the 10 second jobs can be handled as a normal subroutine but what happens when a main program with a 5 minute time estimate calls a factoring subprogram which decides it requires over an hour to do its task? It must send back the message, "We cannot factor your number in the alloted time. You had better terminate your job."

Thus, the computation of a single sequence up to the limits of our computational power was a tedious and time consuming project. The early elements of the sequence were computed very rapidly, but as soon as hard-to-factor number was encountered, the program simply "timed out" and the output obtained was filed away. The difficult number was then submitted to a variety of programs for factorization and ultimately, if no other method succeeded, it was submitted to the continued fraction program. This usually required an over-night run so that in the morning after we collected our factor from the output bin, we had to compute the next term of the aliquot sequence on a desk computer (the Ollivetti 101) before we could rekindle our aliquot program. (In the later stages of the project, our aliquot program could receive a "hint list" of large factors which it would always try before giving up. This feature also provided us with a relatively fast procedure for recalculating a long sequence in order to provide contiguous output.) Since we had a very large number of sequences to compute, we generally had four or five going at any one time. The bookkeeping was very tedious and the probability of error was disturbingly high. At one point in the project, we were all very excited at the prospect of the sequence beginning with 4488 exceeding 1000 terms in length. When we made a recalculation of the sequence using our Hint feature, we discovered that many months (and terms) earlier a mistake was made on the desk computer and we had been computing a different and thoroughly uninteresting sequence ever since. In fact, 4488 terminates with the 459th term. Mistakes such as these were very rare--a credit to precision bookkeeping and data handling, not to versatile and efficient software.

Our new FACTOR program, which is in the design stage at the time of this writing (March, 1976), will automate this entire process. It's novelity relies on the fact that under the IBM OS operating system using HASP any program can send data to the internal reader, a special output channel provided by the system. Such data is immediately processed by HASP and introduced into the input stream as a job which will be queued into execution along with any other jobs that are currently awaiting execution. Thus a program has the capability of submitting another program for execution into the job queue. The program FACTOR has five parameters and could be invoked from an assembly language program using the following macro call.

```
CALL FACTOR,(NUMBER, TIME, AFACT, #FACT, DSN)
```

The parameters are used as follows:

- NUMBER - The number which is to be factored
- TIME - The maximum amount of computer time which FACTOR can use in factoring the number.
- AFACT - A pointer to the address in the main program where the factors are to be stored.
- #FACT - On input, this is the maximum number of factors which can be stored at AFACT. After execution, it contains the number of factors.
- DSN - A pointer to a character string which is the name of a catalogued data set located on an I/O device such as a magnetic disk.

The program also utilizes an external file called MEMORY. Its use will be described in detail later but for the moment, it will suffice to say that whenever a "major effort" is required to obtain a factorization, such as the use of Brillhart's continued fraction program, the number and its factorization is placed on the file MEMORY. We now describe the operation of FACTOR.

1. (Has the number already been factored?) The program first searches the file MEMORY to see if the number and its factorization has already been obtained. If so, it returns the factors to the main program, deletes them from the MEMORY file and returns.

2. (The number is not on MEMORY) The program attempts to factor the number in the time alloted. It can succeed in two ways.

   a) It obtains a complete factorizations and it has factored sufficiently far to guarantee that all the factors are indeed prime including the largest one. In this case, it returns the factors and returns to the main program with a condition code of 10.

   b) It obtains a complete factorization but the largest factor has only been tested for being pseudoprime. That is, it failed Fermat's test for compositeness. In this case, the factors are returned and the program returns after setting the condition code to 4. Thus we make it the responsibility of the main program to do the prime testing.

3. (The program fails) The TIME allotted by the main program wasn't sufficient to factor the number. In this event, the factors obtained together with the composite cofactor are returned and preparation is made to return to the main

program with a condition code of 8. First, however, an estimate is made as to how much additional computer time may be required to complete the factorization and if this amount of time is not excessive a job is introduced into the job stream by writing to the internal reader. This job, which we will call FACTOR2, will not execute immediately, of course, but will ultimately be queued into execution by the operating system.

4. (FACTOR2 executes). This program can now use all the high powered and time consuming methods it needs in order to factor the troublesome number. If it succeeds it performs the following two operations before terminating.

a) The factorization together with the original small factors (if any) found by the initial execution of the program are placed on the MEMORY file so that subsequent execution of the main program will obtain the factorization immediately when it re-invokes the FACTOR program.

b) The data set whose name is DSN is written to the internal reader as an executable job. This is the data set name originally communicated to FACTOR by the original call statement in the main program. If the main program is contained in the data set DSN, control is effectively passed back to the main program through the job submission process.

If the factoring is not successful, the procedure is essentially the same. The partial factorization is put on the MEMORY file and it is marked as being essentially unfactorable. Then DSN is submitted as described as above. When FACTOR is again invoked, the partial factorization can be returned with a condition code of 12, telling the main program that a complete factorization is not feasible with present technology.

It is now evident how a main program can be designed around this factorization system to generate an aliquot sequence. The input for the program would reside on a data set. Whenever the program terminates either by exceeding execution time or attempting to factor a difficult number, restart information is written to the input data set. If the aliquot program exceeds time itself, it must first submit the data set DSN to the internal readers so that the program will re-execute. The output generated by the program must be directed to an output data set rather than being sent to the system printer. Thus a clean copy of the output can be obtained after the entire chain of jobs is completed.

This has been an oversimplified description of the factorization system which we really want to implement. We will conclude this paper by briefly describing some additional features which this system will ultimately have.

a) An additional parameter BOUND should be provided. This tells the factor program that although a complete factorization would be nice, it would be acceptable to provide a set of factors the largest of which has no factor less than BOUND. This would be important for prime testing. On output, in case of failure, BOUND could be set by FACTOR to the extent to which factoring has been done.

b) The factoring program will automatically collect performance statistics and store them on another auxiliary file. It is very important that we continually sharpen our time and core estimates regarding factoring.

c) A parameter MAXTIME could tell FACTOR how large a job it is permitted to submit to the external reader. The user may decide that certain number theoretic projects aren't important enough to consume hours of computer time every night.

d) The main program may wish to decide after receiving partial factorization information whether or not FACTOR2 should be submitted. There are a variety of ways to implement this feature. A special macro PAUSE can be provided which calls an alternate entry point in FACTOR which submits the job FACTOR2 and then returns normally. If the main program ends with a normal RETURN, the job will not be submitted.

e) The program FACTOR2 will itself be a chain of separate jobs. We already have a very successful version of Brillhart's continued fraction program which submits a sequence of small jobs each of which generates a collection of factored quadratic residues. When a large enough collection of factored residues are generated, a 1 minute program is submitted which completes the factorization process by doing a Gaussian elimination on a very large (360,000 bytes) 0-1 matrix. One of our 36 digit factorizations took from Wednesday afternoon to Saturday morning to complete its work, going through a chain of 10 twenty minute jobs.

f) When this system becomes heavily used, a queuing system will be designed. A file FQUEUE will be maintained which contains all the numbers which are awaiting factorization. The file will be sorted according to the size of the number so that small jobs will be executed first. When the in-stream FACTOR program submits FACTOR2, this merely inserts the number together with any small factors which have been found and the return DSN onto the file FQUEUE and sorts it. A separate chain of factoring programs will be removing numbers from the top of the file and factoring them as time permits. Provision will also be made for entering numbers onto FQUEUE from a computer terminal. In this case, the character string DSN will serve as a label rather than a return program so that the factored number can be identified.

REFERENCES

1. J. BRILLHART, D. H. LEHMER, J. L. SELFRIDGE, "Primality testing and new factorizations of $2^n \pm 1$," Math. Comp. 29 (1975).

2. E. CATALAN, Bull. Soc. Math. France, v. 16, 1887-88 pp. 128-129.

3. L. E. DICKSON, "Theorems and tables on the sums of divisors of a number," Quart. J. Math. v. 44, 1913, pp. 264-296.

4. R. K. GUY and J. L. SELFRIDGE, "Interim report on aliquot series," Proc. Manitoba Conf. Numerical Math., Winnipeg, 1971, pp. 557-580.

5. R. K. GUY, D. H. LEHMER, J. L. SELFRIDGE and M. C. WUNDERLICH, "Second report on a aliquot sequences," Proc. 3rd Manitoba Conf. Numerical Math., Winnipeg, 1973, pp. 57-368.

6. R. K. GUY and J. L. SELFRIDGE, Combined Report on Aliquot Sequences, University of Calgary Math. Research Report No. 225, May 1974.

7. R. K. GUY and J. L. SELFRIDGE, "What drives an aliquot sequence," Math. of Comp. v. 29, n. 129, 1975, pp. 101-107.

8. R. K. GUY, "How to factor a number," Proc. Fifth Manitoba Conf. on Numerical Math. 1975.

9. D. E. KNUTH and L. T. PARDO, "Analysis of a simple factorization algorithm", Submitted for Publication.

10. D. E. KNUTH, "The art of computer programming," Vol. II, Revised edition. Addison-Wesley, to appear.

11. M. A. MORRISON and J. BRILLHART, "A method of factoring and the factorization of F_7," Math. of Comp. v. 29, n. 129, 1975, pp. 183-206.

12. J. M. POLLARD, "Theorems on factorization and primality testing," Proc. Cambridge Philos. Soc. 76 (1974) 521-528.

13. ____________, "A Monte Carlo method for factorization," Nordisk Tidskr. Informationsbehandling (BIT) 15 (1975).

14. J. L. SELFRIDGE and M. C. WUNDERLICH, "An efficient algorithm for testing large numbers for primality," Proc. Fourth Manitoba Conf. on Numerical Mathematics, 1974, pp. 109-120.