QI QUAESTIONES INFORMATICAE

Volume 4 Number 2 May 1986

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Africa en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

QUAESTIONES INFORMATICAE

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Africa en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Subscriptions

Annual subscription are as follows:

	SA	US	UK
Individuals	R10	$ 7	£ 5
Institutions	R15	$14	£10

Circulation and Production

Mr C.S.M. Mueller
Department of Computer Science
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg, 2001

Quaestiones Informaticae is prepared by the Computer Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Finding All Primes Less Than Some Maximum

C.C.Handley
University of Natal
Durban 4001

The problem of finding all prime numbers less than some specified maximum is frequently set as an exercise to beginning programmers, particular students doing their first programming course at University. Several different solutions to this problem are presented and tested. Experimentally derived timings are presented for these algorithms implemented in different languages. From these times, we can draw the conclusions that using only primes as divisors is not significantly faster than using all odd numbers, and that, in certain circumstances, FORTRAN programs may well take longer than equivalent Pascal programs.

1. THE PROBLEM

Many generations of students have been set the problem of finding all prime numbers less than some given number (say 1000). This is often used as a lead-in to a discussion of problem-solving techniques, top-down analysis, step-wise refinement etc. Further discussion then brings in the concept of arrays and, by introducing the Sieve of Eratosthenes, leads on to sets as a data type. Other approaches, such as that of Wirth [1], introduce the concept of storing previously calculated values, rather than recalculating them. In all of these discussions, there is the implicit assumption that some solutions are inherently "better" or at least faster than others. This study presents various solutions for "The Prime Number Problem", and shows how speed of execution varies with algorithm and implementation language.

2. THE ALGORITHMS

The algorithms can be classified into two groups, those that examine each candidate prime and determine its primeness in some way (the constructive approach) and those that, having found a prime, remove all its multiples from further consideration (the sieve approach).

2.1 Constructive algorithms

The simplest constructive approach can be specified as:-

```
for x from 1 to max
   test primeness of x
   if x is prime then record x.
```

A moment's analysis convinces us that, if we treat 2 as a special case, we can halve the computational effort by considering only odd numbers as prime candidates. If we refine the test for primeness at the same time, this leads to:-

```
for x from 3 step 2 until max
   factor ← 3
   while x is not divisible by factor and factor < some limit
      factor ← factor + 2
   if x is prime then record x.
```

Note that only odd factors need be considered since we are only dealing with odd candidates. We now need to refine the stopping criterion (factor < some limit) and the test for divisibility. Certainly the limit on factor could be x-1, but no advantage is gained by continuing the search for factors beyond $\sqrt{x}$. The test for divisibility can be performed by testing the remainder for zero.

Thus the final refinement is:-

```
begin {Algorithm 1}
   record 2
   for x from 3 step 2 until max
      factor ← 3
      while x mod factor ≠ 0 and factor < √x do factor ← factor + 2
      if factor > √x then record x.
end
```

This algorithm can be translated into any programming language supporting simple integer arithmetic, since it makes no assumptions about any storage structures, and the **mod** and square root functions can be replaced by equivalents using only integer multiplication and division.

An obvious refinement follows from the realisation that the only useful candidates for factors are the prime numbers themselves. This implies that the operation "record x" must not only write x on to some peripheral device, but must also store it so that its value can be retrieved, which leads to the concept of an array to hold all the primes found so far. If we wish to retain the same limit (factor < $\sqrt{x}$) we need to assume that there will always be enough primes to serve as candidate factors. This is indeed so, as a deep result in number theory shows. This leads to:-

```
begin {Algorithm 2}
   record 2
   for x from 3 step 2 until max
      k ← 1
      while x mod prime_k ≠ 0 and prime_k < √x do k ← k + 1
      if prime_k > √x then record x.
end
```

This approach can be modified further by noting (see [1]) that the stopping criterion can be written in terms of the index of the largest prime that needs to be considered. This limit can be made large enough to start with and increased as and when necessary. This leads to the following:-

```
begin {Algorithm 3}
   record 2; limit ← 1;
   for x from 3 step 2 until max
      if prime_limit ≤ √x then limit ← limit + 1
      k ← 1
      while x mod prime_k ≠ 0 and k ≤ limit do k ← k + 1
      if k > limit then record x.
end
```

This would appear to be as far as this algorithm can be stretched, but, following Wirth [1] (and through him, Dijkstra), it can be expressed somewhat differently. If we examine the last formulation, we notice that no multiplication operations appear, and that the only division operation appears in the test for primeness. Since there are (once again) many processors that do not have an explicit hardware divide instruction (and for those that do it is usually the most time consuming operation in the arithmetic repertoire), and since this operation is going to be repeated very frequently, it might pay us to consider alternative methods of achieving the same effect. Using repeated subtraction is not very attractive, since it is merely a high-level version of a series of machine or micro-code instructions, and therefore is not likely to produce any gains in efficiency. A better solution is to realise that all that is necessary is access to a table of multiples of the primes found so far. If x (the candidate prime) is not found anywhere in the table, then, by definition, x must be prime. Again some thought convinces us that all we need is a tabulation of multiples of primes ($M_k = m * prime_k$) such that for any value of x, $x \leq M_k < x + prime_k$ for k = 2,...,limit.

This leads to the following algorithm:-

```
begin   {Algorithm 4}
   record 2; limit ← 1
   for x from 3 step 2 until max
      if prime_limit ≤ √x
         then m_limit ← (prime_limit)^2; limit ← limit + 1
      k ← 1
      while x ≠ m_k and k < limit
         k ← k + 1
         if m_k < x then m_k ← m_k + primes_k
      if x ≠ m_k then record x
end
```

Note that this is very close to a sieve algorithm as discussed in the next section

2.2 Sieve algorithms

The basic idea of a sieve algorithm is simple. All candidate numbers are placed in the sieve, and the composite numbers are then removed, leaving the primes. Thus once a number is found to be prime (by not having been eliminated as composite) all its multiples can immediately be removed. This leads to the following:-

```
begin {Algorithm 5}
   fill sieve with all candidates
   x ← 1
   while x < √max
      x ← x + 1
      while x is not in sieve do x ← x + 1
      j ← x * x
      while j ≤ max
         eliminate j; j ← j + x
end
```

In the same way as for the constructive algorithm we can consider only odd candidates. This will certainly save space, and may save time as well depending on the cost of the mapping function that needs to be applied to access the sieve. The algorithm can be formulated as follows:-

```
begin {Algorithm 6}
   fill half-size sieve with all odd candidates
   x ← 1
   while x < √max
      x ← x + 2
      while x is not in sieve do x ← x + 2
      j ← x * x
      while j ≤ max
         eliminate j; j ← j + x+x
end
```

3 IMPLEMENTATION

All 6 algorithms were coded initially as self-contained Pascal procedures. (For each of the sieve algorithms, 3 procedures were written - using Boolean arrays, packed Boolean arrays and sets.) A driver program was also developed, which calls t™he procedure, passing to it a parameter specifying the value of max (the upper limit on the primes to be found). In the system used for testing (HP1000E with RTE6VM), the Pascal compiler allows independant compilation

of procedures, with the final choice of which procedure to incorporate left until linkage edit time. The same effect could have been achieved by incorporating the text of each desired algorithm into the source of the driver program and recompiling each time.

Each algorithm was first translated as faithfully as possible from the formulation earlier, and tested in that form. Thereafter, each procedure was examined for the known inefficiencies present in any high level language implementation, in particular array accesses (even without array bound checking this is a slow process). The final version of each procedure was then translated into FORTRAN 77. To facilitate cross-language comparisons, all Pascal options such as range checking were turned off (but only after successful testing!).

4. TESTING

The driver program was responsible for all initialisation and any book-keeping that needed to be done. The chosen procedure was called upon to find all prime numbers in the range [2,max], where max ranged from 1000 to 20 000 in steps of 1000. These limits were chosen because they were large enough to allow the elapsed time to be measured accurately, and because all the routines could then fit easily into the logical address space on the HP1000, i.e. there was no need for hardware or software intervention on any memory accesses. Testing was done at night when no other users were logged on and all other system activity was terminated during the session. This meant that times as measured were reasonably accurate, and were at least consistent. To ensure that the time taken to write results did not affect the time measured for the procedure (this could happen since the printing is handled as a separate task), results were stored in an array and printed at the end of the program.

N	Num	P1	F1	P2	F2	P3	F3	P4	F4	P5	F5	P6	F6
1000	168	0.28	0.33	0.29	0.34	0.30	0.35	0.14	0.25	0.06	0.06	0.03	0.03
2000	303	0.60	0.76	0.61	0.74	0.63	0.75	0.30	0.55	0.11	0.11	0.06	0.07
3000	430	0.95	1.24	0.95	1.18	0.98	1.19	0.48	0.85	0.17	0.16	0.09	0.12
4000	550	1.32	1.77	1.29	1.63	1.34	1.64	0.67	1.16	0.23	0.22	0.13	0.16
5000	669	1.71	2.35	1.65	2.10	1.71	2.12	0.87	1.48	0.29	0.28	0.15	0.20
6000	783	2.13	2.95	2.02	2.59	2.09	2.61	1.08	1.81	0.35	0.34	0.19	0.24
7000	900	2.54	3.59	2.38	3.09	2.46	3.12	1.30	2.15	0.41	0.40	0.22	0.29
8000	1007	2.97	4.24	2.76	3.61	2.85	3.64	1.52	2.49	0.47	0.46	0.26	0.32
9000	1117	3.41	4.92	3.14	4.12	3.25	4.16	1.74	2.83	0.53	0.52	0.29	0.37
10000	1229	3.87	5.63	3.54	4.67	3.65	4.70	1.99	3.19	0.60	0.58	0.33	0.41
11000	1335	4.34	6.36	3.93	5.21	4.05	5.25	2.22	3.56	0.66	0.64	0.37	0.46
12000	1438	4.81	7.11	4.33	5.77	4.47	5.81	2.47	3.92	0.72	0.70	0.40	0.51
13000	1547	5.29	7.87	4.74	6.34	4.89	6.39	2.72	4.30	0.78	0.76	0.43	0.55
14000	1652	5.79	8.67	5.16	6.92	5.31	6.98	2.99	4.68	0.84	0.82	0.47	0.59
15000	1754	6.29	9.47	5.58	7.50	5.74	7.55	3.25	5.05	0.91	0.88	0.51	0.63
16000	1862	6.80	10.29	5.99	8.09	6.17	8.15	3.50	5.43	0.97	0.93	0.54	0.68
17000	1960	7.32	11.12	6.41	8.67	6.59	8.73	3.77	5.81	1.03	0.99	0.58	0.72
18000	2064	7.84	11.98	6.83	9.27	7.03	9.33	4.04	6.20	1.10	1.05	0.62	0.77
19000	2158	8.36	12.81	7.25	9.86	7.45	9.93	4.30	6.58	1.16	1.12	0.65	0.82
20000	2262	8.90	13.71	7.69	10.48	7.90	10.56	4.59	6.98	1.22	1.18	0.68	0.87
Size		68	64	2344	2340	2367	2358	2407	2413	20K	20K	10K	10K

Table 1

Times for various algorithms to find all primes less than N.

Algorithms are labelled Pn (Pascal) or Fn (FORTRAN) where n refers to Algorithm number (see text).

5. RESULTS

The main results are shown in Table 1. The first two columns show respectively the value of the upper limit of the search and the number of primes in the range. The remaining columns are labelled 'Xn' where 'X' is 'P' (Pascal) or 'F' (FORTRAN), and 'n' is a digit representing the algorithm number. The numbers in these columns represent the times (in CPU seconds) taken by the specified algorithm. The line labelled "Size" is the total size of the procedure in 16-bit words as produced by the compiler. This size includes the code and all local work areas and arrays necessary to find all primes up to 20 000. These areas were made the exact size necessary to handle this range.

The three different implementations (in Pascal) of algorithms 5 and 6 were tested separately and the results are shown in Table 2. In this table, labels are of the form 'PnX' where 'X' is blank for the standard representation, 'P' for the packed array implementation and 'S' for the set implementation.

N	P5	P5P	P5S	P6	P6P	P6S
1000	0.05	0.09	52.83	0.03	0.05	7.29
2000	0.11	0.19	111.81	0.06	0.12	17.29
3000	0.17	0.28	173.30	0.09	0.17	28.07
4000	0.23	0.38	236.01	0.12	0.24	39.49
5000	0.29	0.48	299.68	0.16	0.30	51.13
6000	0.35	0.58	364.47	0.20	0.37	63.18
7000	0.41	0.68	429.82	0.23	0.42	75.34
8000	0.47	0.78	495.79	0.26	0.50	87.76
9000	0.53	0.88	562.08	0.29	0.56	100.34
10000	0.60	0.98	628.56	0.33	0.63	112.96
11000	0.66	1.08	695.70	0.36	0.69	125.85
12000	0.72	1.19	763.39	0.40	0.77	138.88
13000	0.78	1.29	831.68	0.43	0.83	152.06
14000	0.85	1.39	899.78	0.47	0.90	165.20
15000	0.90	1.49	968.21	0.51	0.96	178.52
16000	0.98	1.60	1036.65	0.54	1.04	191.78
17000	1.03	1.71	1105.32	0.58	1.11	205.27
18000	1.10	1.81	1174.30	0.61	1.17	218.76
19000	1.17	1.91	1243.40	0.65	1.25	232.37
20000	1.22	2.02	1312.64	0.68	1.31	246.00
Size	20078	1349	3877	10097	755	2028

Table 2

Times for different Pascal implementations of sieve algorithms.
The final character in the name is:
' ' : standard implementation (Boolean arrays),
'P' : implementation using packed Boolean arrays,
'S' : implementation using sets.

6. CONCLUSIONS

The obvious conclusion that can be drawn from these results is that sieve algorithms are considerably faster than constructive algorithms (about 1 order of magnitude). This result was to be expected, and is only of interest because of the magnitude of the difference. Other conclusions are more startling and more interesting.

Firstly, the two simple array approaches are only somewhat faster than the simple (Brute Force and Ignorance) approach, but at the cost of considerably more storage and on any objective combination of time and space cannot be considered an improvement. Wirth's approach (algorithm 4) is a considerable improvement resulting in savings of approximately 50%, with a

somewhat increased space usage with respect to the simple array approaches.

Secondly, it is interesting to note (from Table 2) that, (in this Pascal implementation at least), the time involved in accessing a packed Boolean array is more than compensated for by the savings in space. Furthermore, the algorithm involving only the odd numbers in the sieve (algorithm 6) is faster than the simple version although the mapping function involved in accessing the array is more involved. Thus this particular algorithm is both smaller and faster than the simpler version.

The most interesting conclusion that emerges from these results is that in all but one case the Pascal implementation is faster (usually considerably faster) than the FORTRAN implementation. The compilers used (Pascal 1000 and FTN7X/1000) are of approximately the same age and come from the same source so one would expect code of a similar quality. Part of the reason for the difference in speed is that Pascal treats **mod** as an operator, thus allowing the production of in-line code whereas FORTRAN treats it as a function with all the overhead associated with sub-program invocation and parameter passing. However this does not explain the results for algorithm 6. It is often stated that Pascal is all very well from a theoretical point of view, but is hopelessly inefficient and impractical in the "real world". Certainly, the error checks normally incorporated in a Pascal program do affect running times, however these examples show that it is possible (in one system anyway) for Pascal compilers to produce code that is as good as or better than that from the equivalent FORTRAN compiler. This is an aspect of algorithm timing that deserves more attention.

7. NOTE

Listings of the driver programs and all procedures/subroutines are available on request from the author. Please specify whether you want Pascal, FORTRAN or both.

REFERENCES

[1] Wirth, C. Systematic Programming - An Introduction, Prentice Hall, Englewood Cliffs (1973).