A method for the compact and efficient encoding of ordinal primes

W. Freeman
Department of Computer Science
University of York
York YO10 5DD   U.K.

wf@cs.york.ac.uk

28 April (revised 10 June) 2000

Submitted as a YCS Report 26 March 2003

Indexing terms: ordinal prime numbers, compact encoding, Ada aggregates.

Consider the following two questions. (1) Given *v*, what is the *v*-th prime? (2) Given *p*, then, if *p* is the *v*-th prime, what is *v*? The method described allows these questions to be answered efficiently in real time, and compactly in space, for a range of primes of significant size. We call primes in the range in which the system can deliver their order in this way, *ordinal* primes.

(Note that we are *not* concerned here with simple primality, prime factorisation, etc., which can be determined by investigating the prime in question only.)

Practicalities and implementation issues are addressed. The method has been implemented in Ada95 and tested using GNAT. Some Ada95 code is included, which was used to give fast and compact access to ordinal primes up to 20999999 (the 1329943-th prime).

A method for the compact and efficient encoding of ordinal primes.

Introduction.

By *ordinal primes*, we mean primes recorded in such a way that we can tell (1) what is the v-th prime, given v; and (2) what is v, given prime p and supposing it to be the v-th prime. We shall call the ordinal numbers of such primes, *primals*. As a guide to the magnitudes concerned, it was required to deal with ordinal primes at least 19999999 (primal 1270607). But, the bigger the better.

We are required to answer questions (1) and (2), within the range implemented, compactly in space and efficiently in real time. Clearly, the table concerned must be pre-computed, with the possibilities then that (a) it could be an aggregate, statically compiled in the program, (b) it could be an array loaded in one go, at run-time, from a LAN fileserver, or (c) it could be a set of buffers loaded at run-time on demand. The structure of a wider system of practically usable software, of which this was a part, gave a very strong preference for (a) over (b) or (c), while the elapsed time on demand would always make (c) the least preferable whenever real time is important. This note is concerned in particular with exploring how far option (a) could be developed. Clearly, compactness and efficiency are always important, but are most important for (a). The method developed here is, though, generally applicable, including to cases (b) and (c).

(Note that we are here concerned with ordinal primes and primals, and *not* with simple primality, prime factorisation, etc., since these latter can be determined by investigating the prime in question only.)

Table 1. Primals and ordinal primes.

v	p_v	v	p_v	v	p_v	v	p_v
0	1	1000	7919	1000000	15485863	1270608	2000003
1	2	1001	7927	1000001	15485867	1270609	2000029
2	3	1002	7933	1000002	15485917	1270610	2000039
3	5	1003	7937	1000003	15485927	1270611	2000081
4	7	1004	7949	1000004	15485933	1270612	2000083
5	11	1005	7951	1000005	15485941	1270613	2000093
6	13	1006	7963	1000006	15485959	1270614	2000107
7	17	1007	7993	1000007	15485989	1270615	2000113
8	19	1008	8009	1000008	15485993	1270616	2000143
9	23	1009	8011	1000009	15486013	1270617	2000147

We could set up a simple bit map of all the natural numbers up to some limit forced by the resources available. This could have ‘0’ for composite and ‘1’ for non-composite (i.e. unit or prime[†]). Such a table, up to the 20 millionth natural number (19999999) would occupy 2500000 8-bit bytes, or 625000 32-bit words. It would flag primes up to 19999999, which is the 1270607-th prime. This table would be pre-computed, as also would a table of the running count of the number of primes so far (primals). On the assumption that as much space as is feasible should be allocated to the primality bit-map, this running count would be held only at regular intervals in the bit-map. Using these data to answer either of the questions (1) and (2) above could be efficient in time, but the bit-map would not be very compact.

† To allow a simple dichotomy, composite/prime, the unit is counted as the zeroth prime, throughout.

There is an obvious way of making it more compact: at the very least, the bit-map could be halved in size by storing the primality of odd numbers only. The prime 2 (or its primal, 1) would first be dealt with as a special case; thereafter, the method would proceed as before. A further improvement would be to take out all numbers divisible by 2 or by 3, and map only those that remain. The primes 2 and 3 (or their primals, 1 and 2) would first be dealt with as special cases. The map would now contain two bit positions for every six natural numbers (indicating the primality of those with remainders 1 or 5 on division by 6), so it would now be only one third of its original size.

How can this method be generalised? And, when does further such effort and complication cease to be cost-effective? We now address these questions.

The general method.

We define $\pi(s)$, the *primorial* function of *s*, in the usual way, as the product of the first *s* primes

$$\pi(s) \;\triangleq\; \prod_{i=1}^{s} p_i$$

where p_i is the *i*-th prime. See sequence M1691 in Sloane and Plouffe (1995), and Table 2 here. We define $\mathbf{Z}_m$, the *ring of integers* modulo *m*, in the usual way, as the set of remainders from integer division by *m*

$$\mathbf{Z}_m \;\triangleq\; \{r\colon 0 \le r \le m-1\} \;=\; \{0, 1, \ldots, m-1\}$$

and similarly Φ_m, the *reduced set of residues* modulo *m*, as the subset of $\mathbf{Z}_m$ that contains those elements that are relatively prime to *m*

$$\Phi_m \;\triangleq\; \{r\colon 0 \le r \le m-1 \wedge \gcd(r, m)=1\}$$

and recall that $\#\Phi_m = \phi(m)$, where $\phi(m)$ is *Euler's totient function* defined in terms of the canonical prime factorisation of *m* as

$$\phi(m) \;=\; \phi\Big(\prod_i p_i^{\alpha_i}\Big) \;\triangleq\; \prod_{\substack{i \\ \alpha_i \ne 0}} p_i^{\alpha_i - 1}(p_i - 1)$$

where p_i is the *i*-th prime number. See sequence M0299 in Sloane and Plouffe (1995), and Table 2 here. For number-theoretic matters discussed here, see e.g. Burn (1997).

Any natural number *n* can be expressed uniquely as $q.\pi(s)+r$, where $r \in \mathbf{Z}_{\pi(s)}$, and given fixed *s*. That is, $q = n$ div $\pi(s)$ and $r = n$ mod $\pi(s)$, where 'div' and 'mod' are integer division and remainder in the sense usual in computer science. Now, it is a necessary (but not sufficient) condition for *n* to be prime, that either $n \le p_s$ or $r \in \Phi_{\pi(s)}$. So, leaving cases $n \le p_s$ to be dealt with separately, and regarding cases $n > p_s$ as *mapped* numbers, we can say of a mapped *n* that if $r \notin \Phi_{\pi(s)}$ then *n* is certainly composite, while if $r \in \Phi_{\pi(s)}$ then we need to refer to the appropriate one of a number of bits (actually, $\phi(\pi(s))$ bits) that have been pre-computed to decide its primality.

The set of $\pi(s)$ consecutive numbers, from $q.\pi(s)$ to $(q+1).\pi(s)-1$, will be called the *q*-th *Z-block*. The string of $\phi(\pi(s))$ bits, from the $q.\phi(\pi(s))$-th to the $(q+1).\phi(\pi(s))-1$-th of the primality bit-map, whose values encode the primality of the numbers in the corresponding Z-block, will be called the *q*-th *Φ-block*. Accordingly, we define the *compaction factor*, κ_s, of the encoding, for given *s*, as the ratio between the size of a Φ-block (#Φ) and the size of a Z-block (#Z)

$$\kappa_s \triangleq \frac{\#\Phi}{\#Z} = \frac{\phi(\pi(s))}{\pi(s)}$$

Before drawing up Table 2, we incorporate one simplification of notation. First, we note that $\pi(s)$ is, by construction, always square-free; then, defining

$$\pi'(s) \triangleq \prod_{i=1}^{s} (p_i - 1)$$

we always have $\phi(\pi(s)) = \pi'(s)$.

Table 2. Z-block sizes, Φ-block sizes, and compaction factors.

s	p_s	$\#Z=\pi(s)$	$\Phi_{\pi(s)}$	$\#\Phi=\pi'(s)$	$\kappa_s = \frac{\#\Phi}{\#Z}$	$\frac{\kappa_{s-1}-\kappa_s}{\kappa_{s-1}}$
0	1	1	{0}	1	1.000	
1	2	2	{1}	1	0.500	50.0%
2	3	6	{1, 5}	2	0.333	16.7%
3	5	30	{1, 7, 11, 13, 17, 19, 23, 29}	8	0.266	6.7%
4	7	210	{1, 11, . . ., 199, 209}	48	0.228	3.8%
5	11	2310	{1, 13, . . ., 2297, 2309}	480	0.207	2.1%
6	13	30030	{1, 17, . . ., 30013, 30029}	5760	0.191	1.6%
7	17	510510	{1, 19, . . ., 510491, 510509}	92160	0.180	1.1%

Table 3. Φ-block sizes that are multiples of a byte.

s	p_s	$\pi(s)$	p_s-1	$\pi'(s)$	$\frac{t.\pi'(s)}{8}$	t	$\#\Phi=t.\pi'(s)$	$\#Z=t.\pi(s)$
0	1	1	0	1	1	8	(8)	(8)
1	2	2	1	1	1	8	8	16
2	3	6	2	2	1	4	8	24
3	5	30	4	8	1	1	8	30
4	7	210	6	48	6	1	48	210
5	11	2310	10	480	60	1	480	2310
6	13	30030	12	5760	720	1	5760	30030
7	17	510510	16	92160	11520	1	92160	510510

Table 4. Φ-block sizes that are multiples of a word.

s	p_s	$\pi(s)$	p_s-1	$\pi'(s)$	$\frac{t.\pi'(s)}{32}$	t	$\#\Phi=t.\pi'(s)$	$\#Z=t.\pi(s)$
0	1	1	0	1	1	32	(32)	(32)
1	2	2	1	1	1	32	32	64
2	3	6	2	2	1	16	32	96
3	5	30	4	8	1	4	32	120
4	7	210	6	48	3	2	96	420
5	11	2310	10	480	15	1	480	2310
6	13	30030	12	5760	130	1	5760	30030
7	17	510510	16	92160	2880	1	92160	510510

We suppose that Φ-blocks have been computed and recorded for all values of q from 0 up to a maximum value, $q_{\max} = Q - 1$, so that the primality bit-map contains $Q.\#\Phi$ bits altogether, recording the primality of those natural numbers from 0 to $Q.\#Z - 1$ which, when reduced mod $\#Z$, are relatively prime to $\#Z$. Similarly, the cumulative count of primes will have been computed and recorded in a separate table, at intervals of h bits in the bit-map. There will be $\frac{Q.\#\Phi}{h}$ numbers in this count table (where this ratio is assumed to be a whole number).

Clearly, in practice, it would be more efficient if $\#\Phi$ were a round number of bytes. Further, even if some table look-up (etc.) operations are carried out byte-wise, there will be aspects of the representation that would be better carried out word-wise. So we explore the consequences, first, of requiring that $\#\Phi$ be a multiple of 8 (see Table 3); and, second, of requiring it to be a multiple of 32 (see Table 4). (Extension to other powers of 2 is obvious.) If each Φ-block is to be made larger by the smallest necessary positive integral factor t that will ensure the required divisibility by 8 or by 32, then the Z-block must be made larger by the same factor. We could, in general, say that the new Φ-block and the new Z-block each represent t consecutive copies of the original (i.e. $t = 1$) Φ-block and Z-block, respectively: but it would be less awkward to explain and to code if we could say that $\Phi_{t.\pi'(s)}$ was the reduced set of residues of the ring $\mathbf{Z}_{t.\pi(s))}$; and, so, that each new Φ-block was the residue primality map of the corresponding new Z-block. It is easy to show that this is so for s at least 1. (When $s = 0$, t-fold replication remains a valid explanation.) If $\gamma(m)$ is the greatest square-free divisor of m, we have in general $\gamma(m)|n \Rightarrow \phi(m.n) = m.\phi(n)$; and so, in particular,

$$\gamma(t) \mid \pi(s) \quad \Rightarrow \quad \phi(t.\pi(s)) \; = \; t.\phi(\pi(s))$$

Consequently, if t contains no prime divisors that are not also divisors of $\pi(s)$, then $\phi(t.\pi(s)) = t.\phi(\pi(s)) = t.\pi'(s)$ as required. Since t is here always a power of 2, and $\pi(s)$ is even for s at least 1, the result follows for s at least 1.

The algorithms.

In what follows, $\rho(u)$ is the value of the u-th smallest element of the reduced set of residues modulo $\pi(s)$, so $u \in \mathbf{Z}_{\pi(s)}$ and $\rho(u) \in \Phi_{\pi(s)}$. Also, $\rho^{-1}(r)$ is the partial inverse of ρ defined over $\Phi_{\pi}(s)$, so that $\rho^{-1}(\rho(u)) = u$.

The procedure for pre-computing the tables is as follows.

(1) Choose a small positive integer, s. This will determine that the first s primes are to be treated as special cases. Let $\#Z = \pi(s)$, and $\#\Phi = \pi'(s)$.

(2) Choose a value for the eventual size of the entire primality bit-map that is to be available to any main program using the tables at run-time. Let this size be $Q.\#\Phi$ bits. Each successive Φ-block will be used to record the actual primality of those $\#\Phi$ numbers whose primality is in question, among each successive Z-block of natural numbers.

(3) Choose an interval, h, so that a cumulative count of ‘1’ bits will be kept prior to every $h.\#\Phi$-th bit position in the bit-map.

(4) Find the primality of all numbers n, $0 \le n \le Q.\#Z - 1$. For each n such that n mod $\#Z$ is relatively prime to $\#Z$, use one bit in the bit-map to record its primality (‘1’ for prime). For every $h.\#\Phi$ such bits, use one word in the count table to record

the cumulative count of such ‘1’ bits that applies prior to the start of the interval.

To find the v-th prime, proceed as follows.

(1) If $v \le s$, deal with this specially (in an obvious way). Otherwise, continue.

(2) Search the count table (in logarithmic time) for the greatest count, c, such that $c < v$. Suppose that this c is found at entry i in the count table: then the interval number in the bit-map is i, and the zeroth bit position in that interval is the $h.i.\#\Phi$-th bit position in the whole map.

(3) Scan through the i-th interval of the bit-map, accumulating k, the count of ‘1’ bits, starting with $k = c$, until k first reaches the value n at (say) the j-th bit position within the interval.

(4) Then we have $p_v = (h.i + j \text{ div } \#\Phi).\#Z + \rho(j \text{ mod } \#\Phi)$. So the answer returned is $(h.i + j \text{ div } \#\Phi).\#Z + \rho(j \text{ mod } \#\Phi)$.

To find primal v, given p and that p is the v-th prime, proceed as follows.

(1) If $p \le p_s$, deal with this specially (in an obvious way). Otherwise, continue.

(2) Let $i = p \text{ div } (h.\#Z)$. Take the i-th count from the count table, and let it be c.

(3) Scan through the i-th interval of the primality bit-map, accumulating k, the count of ‘1’ bits, starting with $k = c$, until the next bit position would be the zeroth bit of the $p \text{ div } \#Z$-th Φ-block of the entire map. Retain the resulting value of k.

(4) Let $u = \rho^{-1}(p \text{ mod } \#Z)$. If u is not defined, there must have been an error in the value of p supplied, since it has been found to be composite. Otherwise, let b be the u-th bit of the $(p \text{ div } \#Z)$-th Φ-block of the primality bit-map.

If $b = 0$, there must have been an error in the value of p supplied, since it has been found to be composite. Otherwise, increase k by the number of ‘1’ bits from the zeroth bit position to the u-th bit position (inclusive) within the $(p \text{ div } \#Z)$-th Φ-block of the primality bit-map. So p is p_k, and so the answer returned is k.

Implementation choices and practicalities.

The original implementation, in Ada83 compiled under the York Ada compiler, used $s = 2$ and (effectively) $t = 1$, and held the bit-map in the lower 31 bits of each 32-bit signed integer, giving $\kappa = \frac{32}{31} \times \frac{2}{6} = 0.344$. This arrangement was forced by the lack of unsigned 32-bit or 8-bit types and bitwise operators in Ada83, and by the fact that the York compiler generated Boolean vectors with — astoundingly — one byte, rather than one bit, per element. There were sufficient complications with $s = 2$, without going to $s = 3$, especially since $s = 2$ enabled a primality table of adequate size to be compiled.

In re-coding for Ada95, it was decided that the bit-map would be expressed as a sequence of 32-bit unsigned numbers (using package ada.interfaces). It was hoped that this would enable a larger table to be used. Each 32-bit number (containing four juxtaposed bytes) was output (in decimal, for compactness) by the pre-computation program and incorporated into the ordinal prime package. It was decided initially to use $s = 3$ and $t = 1$, giving $\#Z = 30$, $\#\Phi = 8$ and $\kappa = 0.266$.

The Ada95 was compiled using GNAT 3.11p. The size of aggregate that this compiler could cope with turned out to be much smaller than that possible with the York compiler for Ada83, for a given ceiling on the virtual memory available during compilation. If the bit-map table were merely to be declared statically, and then the aggregate loaded dynamically, there would be no problem; but on this occasion a static aggregate solution was sought, so as to be accessible simply through the Unix execution path. (That does not mean that the general idea of the compact encoding of ordinal primes, as described from the start of this note, is restricted to such an implementation: it is of course usable generally.)

Each of the two tables (bit-map and count) was held on its own in a package specification (see below), but, even if these were forcibly pre-compiled individually, that would be to no avail because the virtual memory limit would be exceeded (only) when there was a demand for code generation and linking, during the first compilation of a main program employing the package. It was decided to explore the setting $s = 4$ and $t = 2$, giving $\#Z = 420$, $\#\Phi = 96$ and $\kappa = 0.228$. (The code given below is for this case.) Then, a bit-map of 150000 words would encode up to 20999999, the 1329943-th prime.

It will be appreciated, from Table 2, that there would be a small further improvement with $s = 5$; while, with $s = 6$ and $s = 7$, the auxiliary tables would have grown to a size that countervailed any further compaction of the bit-map table. (For an example showing the necessary auxiliary tables, see the code given below for $s = 4$ and $t = 2$.)

Actual figures for sizes of virtual memory available, speeds of processors and times required for compilation are not given above, nor in Figure 5, since they became out of date (and were not of historical interest) even during the writing of this note.

Table 5. Implementation characteristics.

Language	Compiler	Words in bit map	Bits used per word	s	t	$\#\Phi$	$\#Z$	κ	$v_{\max}$	$p_{\max}$
Ada 83	York	150000	31	2	–	(64)	(186)	0.344	907101	13949989
Ada 83	York	450000	31	2	–	(64)	(186)	0.344	2539186	41849999
Ada 95	GNAT	50000	32	4	2	96	420	0.228	476606	6999299
Ada 95	GNAT	150000	32	4	2	96	420	0.228	1329943	20999999

References.

Sloane N J A and Plouffe S (1995) *The encyclopaedia of integer sequences*. San Diego CA: Academic Press.

Burn R P (1997) *A pathway into number theory*. Second edition. Cambridge: Cambridge University Press.

Ada code.

The following will be found in the subsequent pages.

[1] Common package specification PRIMA.

[2] Brief description of main program PRIME_GEN that generated primality bit-map and cumulative count tables, for use as aggregates in package specifications PRIMB and PRIMC.

[3] Package specification PRIMB that defined the primality bit-map.

[4] Package specification PRIMC that defined the cumulative count table.

[5] Package specification PRIME that provided primal and ordinal prime functions.

[6] Package body PRIME that provided primal and ordinal prime functions.

```
-------------------------------------[ 1 ]------------------------------------
--
-- common declarations for
--
-- (1) prime-table-generating main program PRIME_GEN
--
-- (2) packages PRIMB, PRIMC, PRIME

package PRIMA is

   WORDS_PER_SET: constant POSITIVE := 3;  -- that is 12 bytes
                                           -- or 96 bits
   subtype WORD_INDEX is NATURAL range 0 .. WORDS_PER_SET - 1;

   S_SIZE: constant NATURAL := 50000; -- number of SETS allocated
                                      -- to primality bit-map table

   subtype S_INDEX is NATURAL range 0 .. S_SIZE - 1;

   P_SIZE: constant NATURAL := S_SIZE * WORDS_PER_SET;
   subtype P_INDEX is NATURAL range 0 .. P_SIZE - 1;

   BITS_PER_BYTE: constant POSITIVE := 8;
   subtype BIT_INDEX is NATURAL range 0 .. BITS_PER_BYTE - 1;

   BYTES_PER_WORD: constant POSITIVE := 4;
   subtype BYTE_INDEX is NATURAL range 0 .. BYTES_PER_WORD - 1;

   RING_SIZE: constant POSITIVE := 2 * (2*3*5*7);        -- equals 420
   subtype RING_INDEX is NATURAL range 0 .. RING_SIZE - 1;

   LAST_MAPPED_NUMBER: constant POSITIVE := S_SIZE * RING_SIZE;

   RESIDUES_PER_SET: constant POSITIVE := 2 * (1*2*4*6);  -- equals 96
   subtype RESIDUE_INDEX is NATURAL range 0 .. RESIDUES_PER_SET - 1;

 -- It is important that RESIDUES_PER_SET = phi (RING_SIZE) = phi (420)
 --  = 2(2-1)(3-1)(5-1)(7-1) = 2(1*2*4*6) = 96 = bits per set
 --  = BITS_PER_BYTE * BYTES_PER_WORD * WORDS_PER_SET

 -- Reduced Set of Residues modulo 420:

   R_S_R: constant array (RESIDUE_INDEX) of RING_INDEX :=
     (  1,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,  47,
       53,  59,  61,  67,  71,  73,  79,  83,  89,  97, 101, 103,
      107, 109, 113, 121, 127, 131, 137, 139, 143, 149, 151, 157,
      163, 167, 169, 173, 179, 181, 187, 191, 193, 197, 199, 209,
      211, 221, 223, 227, 229, 233, 239, 241, 247, 251, 253, 257,
      263, 269, 271, 277, 281, 283, 289, 293, 299, 307, 311, 313,
      317, 319, 323, 331, 337, 341, 347, 349, 353, 359, 361, 367,
      373, 377, 379, 383, 389, 391, 397, 401, 403, 407, 409, 419);

   VAL_OF_SMALLEST_NON_COMPOSITE: constant POSITIVE := 1;
   VAL_OF_SMALLEST_PRIME: constant POSITIVE := 2;
   POS_OF_SMALLEST_NON_COMPOSITE: constant NATURAL := 0;
   POS_OF_SMALLEST_PRIME: constant NATURAL := 1;
   GREATEST_PRIME_DIVISOR_OF_RING_SIZE: constant POSITIVE := 7;
   NUMBER_OF_PRIME_DIVISORS_OF_RING_SIZE: constant POSITIVE := 4;
                                                 -- 2,3,5,7 -- nb not 1

   subtype UNMAPPED_POS_DOMAIN is NATURAL
```

```
                         range POS_OF_SMALLEST_NON_COMPOSITE
                            .. NUMBER_OF_PRIME_DIVISORS_OF_RING_SIZE;
                                     -- contains 0, 1, 2, 3, 4

   subtype UNMAPPED_VAL_DOMAIN is NATURAL
                         range VAL_OF_SMALLEST_NON_COMPOSITE
                            .. GREATEST_PRIME_DIVISOR_OF_RING_SIZE;
            -- contains 1, 2, 3, 5, 7:  i.e. 0th, 1st, 2nd, 3rd, 4th primes

   NUMBER_OF_UNMAPPED_PRIMES: constant POSITIVE :=
                              NUMBER_OF_PRIME_DIVISORS_OF_RING_SIZE;
                                    -- counts 2, 3, 5, 7 (but not 1)

   COUNT_INTERVAL: constant POSITIVE := 5;  -- one count per this many sets
   C_DIVISIBILITY_CHECK: constant NATURAL := (-(S_SIZE mod COUNT_INTERVAL));
   C_SIZE: constant NATURAL := S_SIZE / COUNT_INTERVAL;
   subtype C_INDEX is NATURAL range 0 .. C_SIZE - 1;

end PRIMA;

------------------------------------[ 2 ]-------------------------------------

with PRIMA;
package PRIMGEN is  -- Prime-table-generating main program (not shown here)
                    -- which outputs packages PRIMB.ads and PRIMC.ads,
                    -- as shown below.

 --- etc.

end PRIMGEN;

------------------------------------[ 3 ]-------------------------------------

with INTERFACES; use INTERFACES;
with PRIMA;
package PRIMB is
PRIMES_TABLE: constant array (PRIMA.P_INDEX) of UNSIGNED_32 := (

2147483631,2111749983,3617315763,4009081626,4126078778,2910688479,1022471854,
3017533693,1664527843,536281777,1337654858,2121266222,980116948,2759808102,

 --------------- etc. (150000 numbers altogether) ------------------

34701324,1224769543,26217250,839165992,2323783699,1101009760,268837128,
1153466432,807451140,3223420936,88629281
);
end PRIMB;

------------------------------------[ 4 ]-------------------------------------

with PRIMA;
package PRIMC is
PRIMES_COUNT: constant array (PRIMA.C_INDEX) of NATURAL := (

0,313,570,815,1047,1280,1500,1716,1935,2146,2356,
2576,2777,2985,3190,3385,3594,3791,3994,4191,4388,4585,

 --------------- etc. (10000 numbers altogether) ------------------

1327585,1327706,1327821,1327945,1328069,1328179,1328299,1328425,1328561,
1328686,1328820,1328957,1329074,1329202,1329328,1329452,1329582,1329707,
1329822
);
end PRIMC;

------------------------------------[ 5 ]-------------------------------------

package PRIME is

   function PRIME_POS (P_VAL: in NATURAL) return NATURAL;
   function PRIME_VAL (P_POS: in NATURAL) return POSITIVE;

end PRIME;

-------------------------------------[ 6 ]-------------------------------------

with INTERFACES; use INTERFACES;

with PRIMA; use PRIMA;
with PRIMB; use PRIMB;
with PRIMC; use PRIMC;

package body PRIME is

   -- To save space, the mechanism by which procedure ERROR raises an
   -- exception is omitted here

   BYTE_MASK: constant UNSIGNED_32 := 2#00000000000000000000000011111111#;

   LEFT_MASK: constant array (UNSIGNED_8) of UNSIGNED_8 := (
            2#10000000# => 2#10000000#,
            2#01000000# => 2#11000000#,
            2#00100000# => 2#11100000#,
            2#00010000# => 2#11110000#,
            2#00001000# => 2#11111000#,
            2#00000100# => 2#11111100#,
            2#00000010# => 2#11111110#,
            2#00000001# => 2#11111111#,   others => 0);

   subtype BYTE_WEIGHT_RANGE is NATURAL range 0 .. BITS_PER_BYTE;

   BYTE_WEIGHT: array (UNSIGNED_8) of BYTE_WEIGHT_RANGE := (
            0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
            1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
            1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
            2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
            1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
            2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
            2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
            3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
            1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
            2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
            2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
            3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
            2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
            3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
            3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
            4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8   );

 -- N.b. in RESIDUE_WORD_POS_IN_SET and RESIDUE_BYTE_POS_IN_WORD, a zero
 -- value is used as ‘don’t care’, since these cases will be dealt with
 -- by a zero value in RESIDUE_BIT_POS_IN_BYTE, which means ‘not a prime’.

   RESIDUE_WORD_POS_IN_SET: constant array (RING_INDEX) of WORD_INDEX := (

             1 .. 139 => 0,
           143 .. 277 => 1,
           281 .. 419 => 2,   others => 0);

   RESIDUE_BYTE_POS_IN_WORD: constant array (RING_INDEX) of BYTE_INDEX := (

             1 ..  31 | 143 .. 173 | 281 .. 313 => 0,
            37 ..  67 | 179 .. 209 | 317 .. 349 => 1,
            71 .. 103 | 211 .. 241 | 353 .. 383 => 2,
           107 .. 139 | 247 .. 277 | 389 .. 419 => 3,   others => 0);

   RESIDUE_BIT_POS_IN_BYTE: constant array (RING_INDEX) of UNSIGNED_8 := (

             1 |  37 |  71 | 107 | 143 | 179
         | 211 | 247 | 281 | 317 | 353 | 389 => 2#10000000#,
         11 |  41 |  73 | 109 | 149 | 181
      | 221 | 251 | 283 | 319 | 359 | 391 => 2#01000000#,

         13 |  43 |  79 | 113 | 151 | 187
      | 223 | 253 | 289 | 323 | 361 | 397 => 2#00100000#,

         17 |  47 |  83 | 121 | 157 | 191
      | 227 | 257 | 293 | 331 | 367 | 401 => 2#00010000#,

         19 |  53 |  89 | 127 | 163 | 193
      | 229 | 263 | 299 | 337 | 373 | 403 => 2#00001000#,

         23 |  59 |  97 | 131 | 167 | 197
      | 233 | 269 | 307 | 341 | 377 | 407 => 2#00000100#,

         29 |  61 | 101 | 137 | 169 | 199
      | 239 | 271 | 311 | 347 | 379 | 409 => 2#00000010#,

         31 |  67 | 103 | 139 | 173 | 209
      | 241 | 277 | 313 | 349 | 383 | 419 => 2#00000001#,   others => 0);


-- PRIMES_COUNT (I) contains the count of primes indicated by bits in
-- PRIMES_TABLE (0 .. I * COUNT_INTERVAL * WORDS_PER_SET - 1)
--
-- e.g. with COUNT_INTERVAL = 5, and WORDS_PER_SET = 3,
-- the array PRIMES_COUNT would contain
--
--  PRIMES_COUNT (0) =    0  =  #ones in PRIMES_TABLE (0 .. -1)
--  PRIMES_COUNT (1) =  313  =  #ones in PRIMES_TABLE (0 .. 14)
--  PRIMES_COUNT (2) =  570  =  #ones in PRIMES_TABLE (0 .. 29)
--  PRIMES_COUNT (3) =  815  =  #ones in PRIMES_TABLE (0 .. 44)
--
-- etc.


function PRIME_POS (P_VAL: in NATURAL) return NATURAL is

            -- if P_VAL is P_POS-th prime then return P_POS else ERROR

   PVST: constant NATURAL := P_VAL / RING_SIZE;      -- P_VAL's set number
                                                     --  in primes map
   PVIR: constant RING_INDEX := P_VAL mod RING_SIZE; -- P_VAL's element
                                                     --  number in ring
   PVWS: constant NATURAL := RESIDUE_WORD_POS_IN_SET (PVIR); -- P_VAL's
                                                     -- word number in set
   PVBW: constant NATURAL := RESIDUE_BYTE_POS_IN_WORD (PVIR); -- P_VAL's
                                                     -- byte number in word

   PVIT: constant NATURAL := PVST / COUNT_INTERVAL;  -- P_VAL's number
                                                     --  in count table
   PVFS: constant NATURAL := PVIT * COUNT_INTERVAL;  -- P_VAL's interval's
                                                     --  first set's no
                                                     --  in primes map

   WORD: UNSIGNED_32 := 0;  -- for shifting bytes in a word
                            --  taken from primes map
   BYTE: UNSIGNED_8 := 0;   -- for holding a byte whose 1 bits indicate
                            --  primality in the R_S_R
   RPIB: UNSIGNED_8 := 0;   -- residue position in byte
                            --  for holding a byte whose 1 bit indicates
                            --  the position of P_VAL in the relevant
                            --  byte of the R_S_R

   K: NATURAL := PRIMES_COUNT (PVIT); -- counts 1 bits (primes)

begin
   if P_VAL in UNMAPPED_VAL_DOMAIN then
      case P_VAL is
         when 1 => return 0;
```

```
            when 2 => return 1;
            when 3 => return 2;
            when 5 => return 3;
            when 7 => return 4;
            when others => null;
         end case;
      elsif PVST in S_INDEX then
         for SET_POS in PVFS .. PVST loop
            for WORD_POS in WORD_INDEX loop
               WORD := PRIMES_TABLE (SET_POS * WORDS_PER_SET + WORD_POS);
               for BYTE_POS in BYTE_INDEX loop
                  WORD := ROTATE_LEFT (WORD, BITS_PER_BYTE);
                  BYTE := UNSIGNED_8 (WORD and BYTE_MASK);
                  if SET_POS = PVST and then WORD_POS = PVWS
                                    and then BYTE_POS = PVBW then
                     RPIB := RESIDUE_BIT_POS_IN_BYTE (PVIR);
                     BYTE := BYTE and LEFT_MASK (RPIB);
                     K := K + BYTE_WEIGHT (BYTE);
                     if (RPIB and BYTE) = 0 then
                        ERROR ("primal position requested of a non-prime");
                     else
                        return NUMBER_OF_UNMAPPED_PRIMES + K;
                     end if;
                  end if;
               K := K + BYTE_WEIGHT (BYTE);
               end loop;
            end loop;
         end loop;
      end if;
      ERROR ("primal position requested of too large a number");
      return 0;
   end PRIME_POS;


   function PRIME_VAL (P_POS: in NATURAL) return POSITIVE is -- P_POS-th prime

      M: constant INTEGER := P_POS - NUMBER_OF_UNMAPPED_PRIMES;
                                     -- required mapped count of ‘1’ bits

      WORD: UNSIGNED_32 := 0;        -- for shifting bytes in a word
                                     --  taken from PRIMES_TABLE
      BYTE: UNSIGNED_8 := 0;         -- for holding a byte whose 1 bits
                                     --  indicate primality in the R_S_R
      PEIR: RING_INDEX := 0;         -- position of element in ring

      ILCT: constant POSITIVE := C_INDEX’LAST;  -- index of last count
                                                -- in count table
      I: C_INDEX := ILCT / 2;        -- current index for binary tree search
      D: C_INDEX := (ILCT + 1) / 2;  -- current difference
                                     --  for binary tree search
      K: NATURAL := 0;               -- counts 1 bits (mapped primes)

      SET_POS_FIRST, SET_POS_LAST: S_INDEX := 0; -- relevant interval in map
                                     --  expressed in set numbers
   begin
      if P_POS in UNMAPPED_POS_DOMAIN then
         case P_POS is
            when 0 => return 1;
            when 1 => return 2;
            when 2 => return 3;
            when 3 => return 5;
            when 4 => return 7;
            when others => null;
         end case;
      else
         loop  -- perform a binary tree search of the count table
            D := (D + 1) / 2;
            if M in 1 .. PRIMES_COUNT (I) then
               if I - D in C_INDEX then
                  I := I - D;
               end if;
            elsif M in PRIMES_COUNT (I) + 1 .. PRIMES_COUNT (I+1) then
               exit;
            elsif M in PRIMES_COUNT (I+1) + 1 .. PRIMES_COUNT (ILCT) then
               if I + D in C_INDEX then
                  I := I + D;
               end if;
            elsif PRIMES_COUNT (ILCT) < M then
               I := ILCT;
               exit;
            end if;
         end loop;  -- I now holds the relevant interval number
         K := PRIMES_COUNT (I);
         SET_POS_FIRST := I * COUNT_INTERVAL;
         SET_POS_LAST := (I + 1) * COUNT_INTERVAL - 1;
         for SET_POS in SET_POS_FIRST .. SET_POS_LAST loop
            for WORD_POS in WORD_INDEX loop
               WORD := PRIMES_TABLE (SET_POS * WORDS_PER_SET + WORD_POS);
               for BYTE_POS in BYTE_INDEX loop
                  WORD := ROTATE_LEFT (WORD, BITS_PER_BYTE);
                  BYTE := UNSIGNED_8 (WORD and BYTE_MASK);
                  if M <= K + BYTE_WEIGHT (BYTE) then
                     for BIT_POS in BIT_INDEX loop
                        BYTE := ROTATE_LEFT (BYTE, 1);
                        K := K + INTEGER (BYTE and 1);
                        if K = M then
                           PEIR :=
         (WORD_POS * BYTES_PER_WORD + BYTE_POS) * BITS_PER_BYTE + BIT_POS;
                           return SET_POS * RING_SIZE + R_S_R (PEIR);
                        end if;
                     end loop;
                  end if;
                  K := K + BYTE_WEIGHT (BYTE);
               end loop;
            end loop;
         end loop;
      end if;
      ERROR ("n-th prime requested for too large n");
      return 1;
   end PRIME_VAL;


begin

   null;

end PRIME;
```