Exercise 168, September 17, 2010

The Factorization of F_7

In the seventeenth century, the French jurist and amateur mathematician Pierre de Fermat investigated numbers of the form $2^{2^n} + 1$, now known as Fermat numbers: 3, 5, 17, 257, 65537, 4294967297, 18446744073709551617, ... (Sloane's A000215[1]). Fermat noted that the first five numbers in the sequence, F_0 to F_4, were prime, and conjectured that all the remaining numbers in the sequence were prime. He was wrong, as Leonhard Euler proved in 1732 with the factorization of $F_5 = 641 \cdot 6700417$; then Thomas Clausen[2] factored $F_6 = 274177 \cdot 67280421310721$ in 1855. Those first five numbers are the only members of the sequence that are known to be prime, and it now conjectured that all the remaining Fermat numbers are composite. There is something of a cottage industry among mathematicians and programmers to find new factors of Fermat numbers; you can find the current status of that friendly competition at ProthSearch[3].

On September 13th, 1970, Michael A. Morrison and John Brillhart factored the seventh Fermat number: $F_7 = 2^{2^7} + 1 = 2^{128} + 1 = 340282366920938463463374607431768211457 = 59649589127497217 \cdot 5704689200685129054721$. Their feat[4] inaugurated the modern culture of factorization by computer, and their now-deprecated method was the direct predecessor of the quadratic sieve that is today the most powerful factorization method[5] available for personal computers, able to factor numbers up to about a hundred digits. On the fortieth

[1]http://www.research.att.com/ njas/sequences/A000215

[2]Robert E. Bradley, Lawrence A. D'Antonio and Charles Edward Sandifer, *Euler at 300: an appreciation*, The Mathematical Association of America, July 2007, ISBN 0883855658, page 224

[3]http://www.prothsearch.net/fermat.html

[4]Michael A. Morrison and John Brillhart, "The Factorization of F_7," *Bulletin of the American Mathematical Society*, Volume 77, Number 2, March 1971, Page 264, available at http://projecteuclid.org/euclid.bams/1183532753

[5]the most powerful factoring algorithm known is the number field sieve, which can factor numbers up to about 150 digits, but requires a large cluster of computers

anniversary of their achievement we recreate their computation in this exercise. We are following their own description[6].

The math behind the method dates to Fermat, who noted that a factor of N could be found when $x^2 - N$ was a perfect square; thus, Fermat's factorization method started with $x = \lfloor\sqrt{N}\rfloor$ and decreased x at each step until it reached a solution, as we did in a previous exercise[7]. Early in the twentieth century, Maurice Kratchik, a Russian mathematician and number theorist living in Liège, Belgium, observed that Fermat's formula could be written as $x^2 \equiv y^2 \pmod{N}$. Then, in 1931, Derrick H. Lehmer and R. E. Powers discovered[8] that the convergents of the continued fraction representation of the square root of N provided a suitable x and y, but their method was unusable because the calculations were too tedious and time-consuming for manual calculation, and the method frequently failed to find a factorization. In 1965, Brillhart realized that the tedious calculations were fit for computer calculation, and he and Morrison spent the summer of 1970 developing their method.

The continued fraction method works in two stages, a first stage that computes successive convergents of the continued fraction representing the square root of the number being factored, and a second stage that performs linear algebra on the factors of the convergents and computes a factor of the original number. Morrison and Brillhart, using an IBM 360/91 mainframe computer at UCLA, coding in PL/1 with assembly-language libraries for large-integer processing, divided the work into programs RESIDUE and ANSWER to make it fit into the machine.

We saw continued fractions previously, in the exercise[9] on the golden ratio. Square roots can be represented by continued fractions, and are always periodic; we'll refer to the article by Morrison and Brillhart for the math and skip directly to their algorithm to expand $\sqrt{N}$, or $\sqrt{kN}$ for some suitable multiplier $k \geq 1$, into a simple continued fraction:

```
Initialize $A_{-2} = 0$, $A_{-1} = 1$, $Q_{-1} = kN$, $r_{-1} = g$, $P_0 = 0$, $Q_0 = 1$, and
    $g = \lfloor\sqrt{kN}\rfloor$.
For each $n$ from 0 to a user-specified limit:
        Compute $q_n$ and $r_n$ using the formula $g + P_n = q_n Q_n + r_n$ where
            $0 \leq r_n \leq Q_n$.
        Compute $A_n \pmod{N}$ using the formula $A_n \equiv q_n A_{n-1} + A_{n-2}$
            $\pmod{N}$.
        Compute $g + P_{n+1}$ using the formula $g + P_{n+1} = 2g - r_n$.
        Compute $Q_{n+1}$ using the formula $Q_{n+1} = Q_{n-1} + q_n(r_n - r_{n-1})$.
```

[6]Michael A. Morrison and John Brillhart, "A Method of Factoring and the Factorization of F_7," *Mathematics of Computation*, volume 29, number 129, January 1975, pages 183–205, available at http://www.ams.org/journals/mcom/1975-29-129/S0025-5718-1975-0371800-5/S0025-5718-1975-0371800-5.pdf

[7]http://programmingpraxis.com/2009/05/19/fermats-method

[8]D. H. Lehmer and R. E. Powers, "On Factoring Large Numbers," *Bulletin of the American Mathematical Society*, volume 37, October 1931, pages 770–776

[9]http://programmingpraxis.com//2009/07/10/the-golden-ratio

Some of the Q_n must be factored. That "some" sounds odd. The idea is to factor those Q_n that have all their prime factors less than some pre-specified limit, said primes having a Jacobi symbol of 0 or 1 indicating that they are quadratic residues of kN and thus potential factors; we saw the Jacobi symbol in the exercise[10] on modular arithmetic. The procedure is to choose a limit, compute a *factor base* of those primes that are potential factors, then use trial division to determine if the Q_n should be added to the out-going list; Morrison and Brillhart call such an A_{n-1} and Q_n an $A - Q$ pair.

Morrison and Brillhart, based on Lehmer and Powers before them, give this example: let $N = 13290059$ and $k = 1$; then $g = 3645$. Then selected results from the expansion of $\sqrt{kN}$ are given below:

n	$g + P_n$	Q_n	q_n	r_n	$A_{n-1} \pmod N$	Q_n factored
-1	...	13290059	...	3645	0	...
0	3645	1	3645	0	1	...
1	7290	4034	1	3256	3645	$2 \cdot 2017$
2	4034	3257	1	777	3646	3257
3	6513	1555	4	293	7291	$5 \cdot 311$
4	6997	1321	5	392	32810	1321
5	6898	2050	3	748	171341	$2 \cdot 5^2 \cdot 41$
10	6318	1333	4	986	6700527	$31 \cdot 43$
22	4779	4633	1	146	5235158	$41 \cdot 113$
23	7144	226	31	138	1914221	$2 \cdot 113$
26	5622	3286	1	2336	11455708	$2 \cdot 31 \cdot 53$
31	6248	5650	1	598	1895246	$2 \cdot 5^2 \cdot 113$
40	6576	4558	1	2018	3213960	$2 \cdot 43 \cdot 53$
52	7273	25	290	23	2467124	5^2

The output from RESIDUE is a list of the following items for each Q_n that could be factored over the factor base: n, A_{n-1}, Q_n, and a list of the odd-power primes dividing Q_n (thus, factors like 5^2 of Q_{31} are omitted); forty years ago, the output was given on punched cards! In the example above, the rows for Q_5, Q_{10}, Q_{22}, Q_{23}, Q_{26}, Q_{31} and Q_{40} should appear in the output, based on a factor base containing 2, 5, 31, 41, 43, 53, and 113; note that 5 is part of the factor base even though it doesn't appear as an odd power in any of the Q_n.

Remeber that the math underlying the continued fraction factorization algorithm is that we are searching for x and y such that $x^2 \equiv y^2 \pmod N$ with $x \not\equiv y \pmod N$. The convergents of the continued fraction expansion of $\sqrt{N}$ have the property that A_{n-1} and Q_n will indicate a factor of N whenever Q_n is a perfect square, as we saw in the example of Q_{52} in the expansion of $\sqrt{13290059}$. But such Q_n are rare, which is why the method of Lehmer and Powers never found favor. Much more common is the case that two or more Q_n multiply to a perfect square, as they share some odd-power prime factors that, when multiplied together, become an even power.

[10]http://programmingpraxis.com//2009/07/07/modular-arithmetic/

Consider the example above. The product of Q_5, Q_{22} and Q_{23} is the perfect square $2050 \cdot 4633 \cdot 226 = 2146468900 = 46330^2 = (2 \cdot 5 \cdot 41 \cdot 113)^2$. The product of the corresponding A_{n-1} is $171341 \cdot 5235158 \cdot 1914221 = 1717050890347212038 \equiv 1469504 \pmod{13290059}$, and we have the congruence $(171341 \cdot 5235158 \cdot 1914221)^2 \equiv (2 \cdot 5 \cdot 41 \cdot 113)^2 \pmod{13290059}$ or $1469504^2 \equiv 46330^2 \pmod{13290059}$. Thus a factor of $N = 13290059$ is $\gcd(1469504 - 46330, 13290059) = 4261$ and $N = 3119 \cdot 4261$.

ANSWER uses the Gaussian elimination algorithm of linear algebra to find a set of Q_n (Morrison and Brillhart call it an S-set) with product a perfect square. The primes in the factorizations of the Q_n are 2, 31, 41, 43, 53 and 113, and we also need -1, as there is an implied factor of -1 attached to each Q_n with odd n. Thus, any Q_n can be represented as a vector of the powers of the exponents of its factors, modulo 2; for instance, Q_5, with odd-power factors 2 and 41, is represented by the vector [1 1 0 1 0 0 0]. Associated with each exponent vector is a history vector that records our work, which initially has 0 everywhere except 1 in the ordinal position corresponding to the row with which it is associated. Here are the exponent matrix, on the left, and history matrix, on the right, corresponding to the seven Q_n of our example problem; the columns of the exponent matrix are factors, left to right from low to high, starting with -1, the columns of the history matrix are Q_n, left to right from low to high, and the rows of both matrices are Q_n, top to bottom from low to high:

$$
\begin{array}{ccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 0
\end{array}
\qquad
\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Once the exponent matrix and history matrix are established, the reduction procedure, which performs the forward step of Gaussian elimination, is done as follows:

```
For each column in the exponent matrix, working from right to left:
    Find the "pivot" vector of smallest subscript (nearest the top)
        whose rightmost 1 is in the current column. If none exists,
        continue working on the next column to the left.
    For each non-pivot vector with rightmost 1 in the current column:
        Replace the exponent vector with the element-wise sum,
            modulo 2, of the pivot vector and the current vector.
        Replace the associated history vector with the element-
            wise sum, modulo 2, of the pivot vector and the
            current vector.
```

The reduction of the initial matrices given above is shown below:

$$\begin{matrix}1&1&0&1&0&0&0\\0&0&1&0&1&0&0\\0&0&0&1&0&0&1\\0&0&0&0&0&0&0\\0&1&1&0&0&1&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{matrix}\qquad\begin{matrix}1&0&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&1&0&0&0&0\\1&0&1&1&0&0&0\\0&0&0&0&1&0&0\\1&0&1&0&0&1&0\\0&1&0&0&1&0&1\end{matrix}$$

Each of the three rows with zero exponent vectors represents an S-set. Some of those S-sets lead to factorizations, but others fail. For instance, the S-set in the seventh row of the reduced matrix gives the congruence $(6700527 \cdot 11455708 \cdot 3213960)^2 \equiv (2 \cdot 31 \cdot 43 \cdot 53)^2 \pmod{13290059}$, or $141298^2 \equiv 141298^2 \pmod{13290059}$, which is useless. The S-set in the sixth row gives the congruence $(171341 \cdot 5235158 \cdot 1895246)^2 \equiv (2 \cdot 5^2 \cdot 41 \cdot 113)^2 \pmod{13290059}$, or $13058409^2 \equiv 231650^2 \pmod{13290059}$, but $\gcd(13058409 - 231650, 13290059) = 1$ and the factorization fails. However, the S-set in the fourth row gives a completed factorization, as shown above.

Your task is to write the RESIDUE and ANSWER programs described above, then compute the factorization of F_7 using a factor base of the first 2700 suitable primes less than 60000 and a multiplier of 257.

We begin with the RESIDUE program that computes the factor base, factors the Q_n, and gathers the $A - Q$ pairs. `Make-factor-base` returns a list of primes included in the factor base. The arguments include both `bound`, the largest possible prime to be included in the list, and `lim`, the number of factors to be included in the list, since different uses may require different arguments:

```
(define (make-factor-base n k bound lim)
  (let loop ((i (- lim 1)) (ps (cdr (primes bound))) (fb '(2)))
    (cond ((or (zero? i) (null? ps)) (reverse fb))
          ((not (negative? (jacobi (* k n) (car ps))))
            (loop (- i 1) (cdr ps) (cons (car ps) fb)))
          (else (loop i (cdr ps) fb)))))
```

`Smooth` determines if Q_n factors over the factor base, using trial division, except that even-power factors are excluded by the `if` in the second `cond` clause. If a factorization is found, the list of factors is returned, otherwise `#f`:

```
(define (smooth n q fb)
  (let loop ((q q) (fb fb) (fs '()) (i 0))
    (cond ((= q 1) (reverse fs))
          ((null? fb) #f)
          ((zero? (modulo q (car fb)))
            (if (and (pair? fs) (= (car fb) (car fs)))
                (loop (/ q (car fb)) fb (cdr fs) (+ i 1))
                (loop (/ q (car fb)) fb (cons (car fb) fs) (+ i 1))))
          (else (loop q (cdr fb) fs (+ i 1))))))
```

In their article, Morrison and Brillhart give a "large-prime" variant of the Q-factoring procedure, but specifically state that they did not use it in their factorization of F_7, so neither will we.

`Residue` implements the RESIDUE program of Morrison and Brillhart. We made two changes for clarity, renaming n as `i` and q as `t`, since Scheme doesn't distinguish case. Otherwise, we follow the algorithm of Morrison and Brillhart exactly; note the first clause of the `cond`, which performs an early return if it finds a Q_n that is a perfect square:

```
(define (residue n k fb lim)
  (let* ((kn (* k n)) (g (isqrt kn)) (a-3 0) (a-2 1)
         (q-2 kn) (r-2 g) (g+p-1 g) (q-1 1)
         (t-1 (quotient g+p-1 q-1)) (r-1 (- g+p-1 (* q-1 t-1)))
         (q0 (+ q-2 (* t-1 (- r-1 r-2)))))
    (let loop ((a-2 1) (a-1 g) (q-1 1) (r-1 0) (g+p (+ g g))
               (q q0) (i 1) (qs '()) (lim lim))
      ; (for-each display `(,i " " ,g+p " " ,q " " ,a-1 #\newline))
      (if (or (= q 1) (zero? lim)) (reverse qs)
        (let* ((t (quotient g+p q))
               (r (- g+p (* q t)))
               (a (modulo (+ (* t a-1) a-2) n))
               (q+1 (+ q-1 (* t (- r r-1))))
               (fs (smooth n q fb)))
          (cond ((null? fs)
                  (let ((d (gcd (- a-1 (isqrt q)) n)))
                    (if (< 1 d n) d
                      (loop a-1 a q r (- (* 2 g) r) q+1 (+ i 1) qs (- lim 1)))))
                (fs (loop a-1 a q r (- (* 2 g) r) q+1 (+ i 1)
                          (cons (cons* i q a-1 fs) qs) (- lim 1)))
                (else (loop a-1 a q r (- (* 2 g) r) q+1 (+ i 1) qs (- lim 1))))))))
```

Here are two examples, the second showing a short-circuit factor when a perfect square Q_n is found:

```
> (residue 13290059 1 '(2 5 31 41 43 53 113) 44)
((5 2050 171341 2 41)
 (10 1333 6700527 31 43)
 (22 4633 5235158 41 113)
 (23 226 1914221 2 113)
 (26 3286 11455708 2 31 53)
 (31 5650 1895246 2 113)
 (40 4558 3213960 2 43 53))
> (residue 13290059 1 '(2 5 31 41 43 53 113) 60)
4261
```

It takes several minutes to make the corresponding calculations for F_7:

```
> (define f7 (+ (expt 2 (expt 2 7) 1))
> (define fb (make-factor-base f7 257 60000 2700))
> (define residues (residue f7 257 fb 1330000))
```

The largest of the 2700 primes in the factor base is 52183. Morrison and Brillhart found 2059 factored Q_n; for some unexplained reason, we found 4034. The maximum number of factors for any of the 4034 Q_n is 12, and a total of 29326 factors, about 7.3 per Q_n, were found.

We turn now to the linear algebra phase of the algorithm. `Make-row` returns the exponent vector for a single Q_n:

```
(define (make-row rs fb len)
  (let ((vec (make-vector (+ len 1) 0)))
    (vector-set! vec 0 (if (odd? (car rs)) 1 0))
    (let loop ((i 1) (rs (cdddr rs)) (fb fb))
      (cond ((or (null? rs) (null? fb)) vec)
            ((= (car fb) (car rs))
              (vector-set! vec i 1)
              (loop (+ i 1) (cdr rs) (cdr fb)))
            (else (loop (+ i 1) rs (cdr fb)))))))
```

`Make-ev` and `make-hv` build the exponent and history matrices, respectively:

```
(define (make-ev fb len residues)
  (let loop ((rss residues) (es '()) (as '()) (qs '()))
    (if (null? rss)
        (values (list->vector (reverse es))
                (list->vector (reverse as))
                (list->vector (reverse qs)))
        (loop (cdr rss)
              (cons (make-row (car rss) fb len) es)
              (cons (caddar rss) as)
              (cons (cadar rss) qs)))))


(define (make-hv ev-len)
  (let ((hv (make-vector ev-len)))
    (do ((i 0 (+ i 1))) ((= i ev-len) hv)
      (let ((h (make-vector ev-len 0)))
        (vector-set! h i 1)
        (vector-set! hv i h)))))
```

The calculation of the rightmost 1 is shown below. Later, in the `answer` function, we will keep track of the rightmost 1 in each row rather than recomputing it each time it is needed, just as Morrison and Brillhart did:

```
(define (right-most-one rv start)
  (let loop ((i start))
    (cond ((negative? i) i)
          ((= (vector-ref rv i) 1) i)
          (else (loop (- i 1))))))
```

Finally, we give the code for ANSWER, which initializes the exponent vector and history vector, creates a pointer vector for the rightmost 1 in each column, performs the reduction procedure, and then examines S-sets until it finds one that gives a factorization:

```
(define (answer n fb residues)
  (let-values (((ev av qv) (make-ev fb (length fb) residues)))
    (let* ((fb-len (length fb)) (ev-len (vector-length ev))
           (hv (make-hv ev-len)) (ptr (make-vector ev-len)))
      (do ((i 0 (+ i 1))) ((= i ev-len))
        (vector-set! ptr i (right-most-one (vector-ref ev i) fb-len)))
      ; reduce ev and hv
      (do ((j fb-len (- j 1))) ((negative? j))
        (let loop ((i 0))
          (when (< i ev-len)
            (if (= (vector-ref ptr i) j)
                (do ((m (+ i 1) (+ m 1))) ((= m ev-len))
                  (when (= (vector-ref ptr m) j)
                    (vector-set! ev m (vector-add (vector-ref ev i) (vector-ref ev m)))
                    (vector-set! hv m (vector-add (vector-ref hv i) (vector-ref hv m)))
                    (vector-set! ptr m (right-most-one (vector-ref ev m) j)))))
                (loop (+ i 1))))))
      ; examine s-sets
      (let loop ((i 0))
        (cond ((= i ev-len) #f)
              ((zero? (vector-sum (vector-ref ev i)))
                (let ((d (gcd (a-minus-q n (vector-ref hv i) ev-len av qv) n)))
                  (if (< 1 d n) d (loop (+ i 1)))))
              (else (loop (+ i 1))))))))
```

`Answer` uses the utility functions shown below:

```
(define (vector-add v1 v2)
  (let* ((len (vector-length v1))
         (v3 (make-vector len)))
    (do ((i 0 (+ i 1))) ((= i len) v3)
      (vector-set! v3 i
        (if (= (vector-ref v1 i)
               (vector-ref v2 i)) 0 1)))))
```

```
(define (vector-sum vec)
  (do ((i 0 (+ i 1)) (sum 0 (+ sum (vector-ref vec i))))
      ((= i (vector-length vec)) sum)))
```

`Answer` also uses `a-minus-q`, which takes an S-set and returns a divisor of N; it is up to `answer` to determine if that factor us useful or trivial. Note that the calculation of the square root of `q-prod` is exact:

```
(define (a-minus-q n vec ev-len av qv)
  (let loop ((i 0) (a-prod 1) (q-prod 1))
    (cond ((= i ev-len) (modulo (- a-prod (isqrt q-prod)) n))
          ((= (vector-ref vec i) 1)
            (loop (+ i 1)
                  (modulo (* a-prod (vector-ref av i)) n)
                  (* q-prod (vector-ref qv i))))
          (else (loop (+ i 1) a-prod q-prod)))))
```

Using the factor base and residues calculated previously, the factorization of F_7 is shown below:

```
> (answer f7 fb residues)
59649589127497217
> (/ f7 59649589127497217)
5704689200685129054721
```

The S-set that completed the factorization contained 466 $A-Q$ pairs. You can see the entire program at http://programmingpraxis.codepad.org/JGMHnA0v.

We used several functions from elsewhere. `Cons*` and `isqrt` are from the Standard Prelude[11]:

```
(define (cons* first . rest)
  (let loop ((curr first) (rest rest))
    (if (null? rest) curr
        (cons curr (loop (car rest) (cdr rest))))))

(define (isqrt n)
  (if (not (and (positive? n) (integer? n)))
      (error 'isqrt "must be positive integer")
      (let loop ((x n))
        (let ((y (quotient (+ x (quotient n x)) 2)))
          (if (< y x) (loop y) x)))))
```

`Primes` comes from the exercise[12] on the Sieve of Eratosthenes:

```
(define (primes n)
  (let* ((max-index (quotient (- n 3) 2))
         (v (make-vector (+ 1 max-index) #t)))
    (let loop ((i 0) (ps '(2)))
      (let ((p (+ i i 3)) (startj (+ (* 2 i i) (* 6 i) 3)))
        (cond ((>= (* p p) n)
               (let loop ((j i) (ps ps))
                  (cond ((> j max-index) (reverse ps))
                        ((vector-ref v j)
                          (loop (+ j 1) (cons (+ j j 3) ps)))
                        (else (loop (+ j 1) ps)))))
              ((vector-ref v i)
                (let loop ((j startj))
                  (if (<= j max-index)
                      (begin (vector-set! v j #f)
                             (loop (+ j p)))))
                      (loop (+ 1 i) (cons p ps)))
              (else (loop (+ 1 i) ps))))))))
```

[11]http://programmingpraxis.com/standard-prelude

[12]http://programmingpraxis.com/2090/02/19/sieve-of-eratosthenes

`Jacobi` comes from the exercise[13] on Modular Arithmetic.

```
(define (jacobi a n)
  (if (not (and (integer? a) (integer? n) (positive? n) (odd? n)))
      (error 'jacobi "modulus must be positive odd integer")
      (let jacobi ((a a) (n n))
        (cond ((= a 0) 0)
              ((= a 1) 1)
              ((= a 2) (case (modulo n 8) ((1 7) 1) ((3 5) -1)))
              ((even? a) (* (jacobi 2 n) (jacobi (quotient a 2) n)))
              ((< n a) (jacobi (modulo a n) n))
              ((and (= (modulo a 4) 3) (= (modulo n 4) 3)) (- (jacobi n a)))
              (else (jacobi n a))))))
```

[13]http://programmingpraxis.com/2009/07/07/modular-arithmetic