

Figure 7.14 Contours of the resonance Hamiltonian H'_V , which has been developed to study the stability of the vertical equilibrium, are shown in the upper plot. A corresponding surface of section for the actual driven pendulum is shown in the lower plot. The parameters are $m = 1\text{ kg}$, $l = 1\text{ m}$, $g = 9.8\text{ m s}^{-2}$, $A = 0.03\text{ m}$, and $\omega = 100\omega_s$, where $\omega_s = \sqrt{g/l}$.

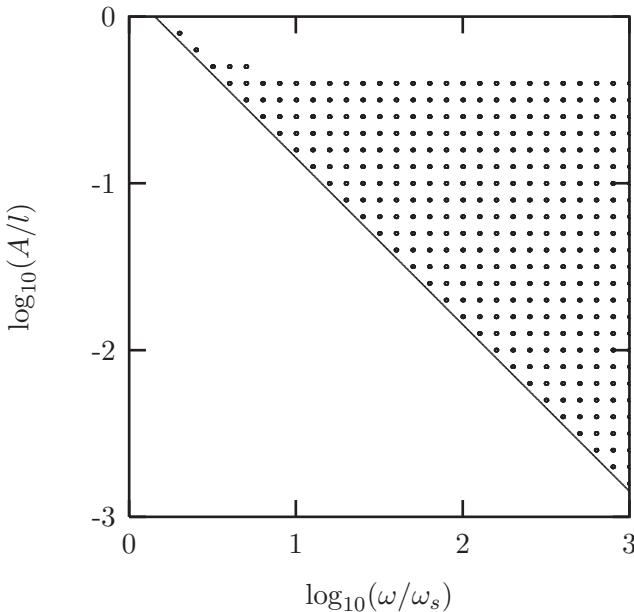


Figure 7.15 Stability of the inverted vertical equilibrium over a range of parameters. The full parameter space displayed was sampled over a regular grid. The dots indicate parameters for which the actual driven pendulum is linearly stable; nothing is plotted in the case of instability. The diagonal line is the locus of points satisfying $(\omega/\omega_s)(A/l) = \sqrt{2}$.

teristic polynomial for a reference orbit at the resonance center. In the figure the stability of the inverted vertical equilibrium was assessed at each point of a grid of assignments of the parameters. A dot is shown for combinations of parameters that are linearly stable. The diagonal line is the analytic boundary of the region of stability of the inverted equilibrium: $(\omega/\omega_s)(A/l) = \sqrt{2}$. We see that the boundary of the region of stability is well approximated by the analytic estimate derived from perturbation theory. Note that for very high drive amplitudes there is another region of instability, which is not captured by this perturbation analysis.

7.5 Summary

The goal of perturbation theory is to relate aspects of the motions of a given system to those of a nearby solvable system. Perturba-

tion theory can be used to predict features such as the size and location of the resonance islands and chaotic zones.

With perturbation analysis we obtain an approximation to the evolution of a system by relating the evolution of the system to that of a different system that, when approximated, can be exactly solved. We can carry this exact solution of the approximate problem back to the original system to obtain an approximate solution of our original problem. The strategy of canonical perturbation theory is to make canonical transformations that eliminate terms in the Hamiltonian that impede solution. Formulation of perturbation theory in terms of Lie series is especially convenient.

We can use first-order perturbation theory to analyze the motion of the undriven pendulum as a free rotor to which gravity is added. In this analysis we find that a small denominator in the series limits the range of applicability of the perturbative solution to regions that are away from the resonant oscillation region.

In higher-order perturbation theory for the pendulum we discover the problem of secular terms, terms that produce error that grow with time. The appearance of secular terms can be avoided by keeping track of how the frequencies change as perturbations are included. In canonical perturbation theory secular terms can be avoided by associating the average part of the perturbation with the solvable part of the Hamiltonian.

In carrying out canonical perturbation theory in higher dimensions we find that the problem of small denominators is more serious. Small denominators arise near every commensurability, and commensurabilities are common. Small denominators can be locally avoided near particular commensurabilities by incorporating the offending terms into the solvable part of the Hamiltonian. If the resonances are isolated, the resulting resonance Hamiltonian is still solvable. In many cases the resonance Hamiltonian is well approximated by a pendulum-like Hamiltonian. A global picture can be constructed by stitching together the solutions for each resonance region constructed separately.

If two resonance regions overlap—that is, if the sum of the half-widths of the resonance regions exceeds their separation—then large-scale chaos ensues. The chaotic regions associated with the separatrices of the overlapping resonances become connected. When the resonances are well approximated by pendulum-like resonances a simple analytic criterion for the appearance of large-scale chaos can be developed.

Higher-order perturbative descriptions can be developed to describe islands that do not correspond to particular terms in the Hamiltonian, secondary resonances, bifurcations, and so on. The theory can be extended to describe as much detail as one wishes.

7.6 Projects

Exercise 7.4: Periodically driven pendulum

- Work out the details of the perturbation theory for the primary driven pendulum resonances, as displayed in figure 7.10.
- Work out the details of the perturbation theory for the stability of the inverted vertical equilibrium. Derive the resonance Hamiltonian and plot its contours. Compare these contours to surfaces of section for a variety of parameters.
- Carry out the linear stability analysis leading to equation (7.88). What is happening in the upper part of figure 7.15? Why is the system unstable when criterion (7.88) predicts stability? Use surfaces of section to investigate this parameter regime.

Exercise 7.5: Spin-orbit coupling

A Hamiltonian for the spin-orbit problem described in section 2.11.2 is

$$\begin{aligned} H(t, \theta, p_\theta) &= \frac{p_\theta^2}{2C} - \frac{n^2 \epsilon^2 C}{4} \frac{a^3}{R^3(t)} \cos 2(\theta - f(t)) \\ &= \frac{p_\theta^2}{2C} - \frac{n^2 \epsilon^2 C}{4} (\cos(2\theta - 2nt) + \frac{7e}{2} \cos(2\theta - 3nt) \\ &\quad - \frac{e}{2} \cos(2\theta - nt) + \dots) \end{aligned} \tag{7.89}$$

where the ignored terms are higher order in eccentricity e . Note that here ϵ is the out-of-roundness parameter.

- Find the widths and centers of the three primary resonances. Compare the predictions for the widths to the island widths seen on surfaces of section. Write the criterion for resonance overlap and compare to numerical experiments for the transition to large-scale chaos.
- The fixed point of the synchronous island is offset from the average rate of rotation. This is indicative of a “forced” oscillation of the rotation of the Moon. Develop a perturbative theory for motion in the synchronous island by using a Lie transform to eliminate the two non-synchronous resonances. Predict the location of the fixed point at the center of the synchronous resonance on the surface of section, and thus predict the amplitude of the forced oscillation of the Moon.

8

Appendix: Scheme

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

IEEE Standard for the Scheme Programming Language [24], p. 3

Here we give an elementary introduction to Scheme.¹ For a more precise explanation of the language see the IEEE standard [24]; for a longer introduction see the textbook [1].

Scheme is a simple programming language based on expressions. An expression names a value. For example, the numeral 3.14 names an approximation to a familiar number. There are primitive expressions, such as numerals, that we directly recognize, and there are compound expressions of several kinds.

Procedure calls

A *procedure call* is a kind of compound expression. A procedure call is a sequence of expressions delimited by parentheses. The first subexpression in a procedure call is taken to name a procedure, and the rest of the subexpressions are taken to name the arguments to that procedure. The value produced by the procedure when applied to the given arguments is the value named by the procedure call. For example,

¹Many of the statements here are valid only assuming that no assignments are used.

```
(+ 1 2.14)
3.14

(+ 1 (* 2 1.07))
3.14
```

are both compound expressions that name the same number as the numeral 3.14.² In these cases the symbols `+` and `*` name procedures that add and multiply, respectively. If we replace any subexpression of any expression with an expression that names the same thing as the original subexpression, the thing named by the overall expression remains unchanged. In general, a procedure call is written

```
( operator operand-1 ... operand-n )
```

where *operator* names a procedure and *operand-i* names the *i*th argument.³

Lambda expressions

Just as we use numerals to name numbers, we use λ -expressions to name procedures.⁴ For example, the procedure that squares its input can be written:

```
(lambda (x) (* x x))
```

This expression can be read: “The procedure of one argument, *x*, that multiplies *x* by *x*.” Of course, we can use this expression in any context where a procedure is needed. For example,

```
((lambda (x) (* x x)) 4)
16
```

²In examples we show the value that would be printed by the Scheme system using slanted characters following the input expression.

³In Scheme every parenthesis is essential: you cannot add extra parentheses or remove any.

⁴The logician Alonzo Church [13] invented λ -notation to allow the specification of an anonymous function of a named parameter: $\lambda x[\text{expression in } x]$. This is read, “That function of one argument that is obtained by substituting the argument for *x* in the indicated expression.”

The general form of a λ -expression is

```
(lambda formal-parameters body)
```

where *formal-parameters* is a list of symbols that will be the names of the arguments to the procedure and *body* is an expression that may refer to the formal parameters. The value of a procedure call is the value of the body of the procedure with the arguments substituted for the formal parameters.

Definitions

We can use the `define` construct to give a name to any object. For example, if we make the definitions⁵

```
(define pi 3.141592653589793)  
(define square (lambda (x) (* x x)))
```

we can then use the symbols `pi` and `square` wherever the numeral or the λ -expression could appear. For example, the area of the surface of a sphere of radius 5 is

```
(* 4 pi (square 5))  
314.1592653589793
```

Procedure definitions may be expressed more conveniently using “syntactic sugar.” The squaring procedure may be defined

```
(define (square x) (* x x))
```

which we may read: “To square *x* multiply *x* by *x*.”

In Scheme, procedures may be passed as arguments and returned as values. For example, it is possible to make a procedure that implements the mathematical notion of the composition of two functions.⁶

⁵The definition of `square` given here is not the definition of `square` in the Scmutils system. In Scmutils, `square` is extended for tuples to mean the sum of the squares of the components of the tuple. However, for arguments that are not tuples the Scmutils `square` does multiply the argument by itself.

⁶The examples are indented to help with readability. Scheme does not care about extra white space, so we may add as much as we please to make things easier to read.

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

((compose square sin) 2)
.826821810431806

(square (sin 2))
.826821810431806
```

Using the syntactic sugar shown above, we can write the definition more conveniently. The following are both equivalent to the definition above:

```
(define (compose f g)
  (lambda (x)
    (f (g x)))

(define ((compose f g) x)
  (f (g x)))
```

Conditionals

Conditional expressions may be used to choose among several expressions to produce a value. For example, a procedure that implements the absolute value function may be written:

```
(define (abs x)
  (cond ((< x 0) (- x))
        ((= x 0) x)
        ((> x 0) x)))
```

The conditional `cond` takes a number of clauses. Each clause has a predicate expression, which may be either true or false, and a consequent expression. The value of the `cond` expression is the value of the consequent expression of the first clause for which the corresponding predicate expression is true. The general form of a conditional expression is

```
(cond ( predicate-1  consequent-1)
      ...
      ( predicate-n  consequent-n))
```

For convenience there is a special predicate expression `else` that can be used as the predicate in the last clause of a `cond`.

The `if` construct provides another way to make a conditional when there is only a binary choice to be made. For example, because we have to do something special only when the argument is negative, we could have defined `abs` as:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

The general form of an `if` expression is

```
(if predicate consequent alternative)
```

If the *predicate* is true the value of the `if` expression is the value of the *consequent*, otherwise it is the value of the *alternative*.

Recursive procedures

Given conditionals and definitions, we can write recursive procedures. For example, to compute the *n*th factorial number we may write:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

(factorial 6)
720

(factorial 40)
815915283247897734345611269596115894272000000000
```

Local names

The `let` expression is used to give names to objects in a local context. For example,

```
(define (f radius)
  (let ((area (* 4 pi (square radius)))
        (volume (* 4/3 pi (cube radius))))
    (/ volume area)))

(f 3)
1
```

The general form of a `let` expression is

```
(let ((variable-1 expression-1)
      ...
      (variable-n expression-n))
  body)
```

The value of the `let` expression is the value of the `body` expression in the context where the variables *variable-i* have the values of the expressions *expression-i*. The expressions *expression-i* may not refer to any of the variables *variable-j* given values in the `let` expression.

A `let*` expression is the same as a `let` expression except that an expression *expression-i* may refer to variables *variable-j* given values earlier in the `let*` expression.

A slight variant of the `let` expression provides a convenient way to express looping constructs. We can write a procedure that implements an alternative algorithm for computing factorials as follows:

```
(define (factorial n)
  (let factlp ((count 1) (answer 1))
    (if (> count n)
        answer
        (factlp (+ count 1) (* count answer)))))

(factorial 6)
720
```

Here, the symbol `factlp` following the `let` is locally defined to be a procedure that has the variables `count` and `answer` as its formal parameters. It is called the first time with the expressions 1 and 1, initializing the loop. Whenever the procedure named `factlp` is called later, these variables get new values that are the values of the operand expressions `(+ count 1)` and `(* count answer)`.

Compound data—lists and vectors

Data can be glued together to form compound data structures. A list is a data structure in which the elements are linked sequentially. A Scheme vector is a data structure in which the elements are packed in a linear array. New elements can be added to lists, but to access the *n*th element of a list takes computing time proportional to *n*. By contrast a Scheme vector is of fixed length, and its elements can be accessed in constant time. All data structures

in this book are implemented as combinations of lists and Scheme vectors. Compound data objects are constructed from components by procedures called constructors and the components are accessed by selectors.

The procedure `list` is the constructor for lists. The selector `list-ref` gets an element of the list. All selectors in Scheme are zero-based. For example,

```
(define a-list (list 6 946 8 356 12 620))

a-list
(6 946 8 356 12 620)

(list-ref a-list 3)
356

(list-ref a-list 0)
6
```

Lists are built from pairs. A pair is made using the constructor `cons`. The selectors for the two components of the pair are `car` and `cdr` (pronounced “could-er”).⁷ A list is a chain of pairs, such that the `car` of each pair is the list element and the `cdr` of each pair is the next pair, except for the last `cdr`, which is a distinguishable value called the empty list and is written `()`. Thus,

```
(car a-list)
6

(cdr a-list)
(946 8 356 12 620)

(car (cdr a-list))
946

(define another-list
  (cons 32 (cdr a-list)))

another-list
(32 946 8 356 12 620)
```

⁷These names are accidents of history. They stand for “Contents of the Address part of Register” and “Contents of the Decrement part of Register” of the IBM 704 computer, which was used for the first implementation of Lisp in the late 1950s. Scheme is a dialect of Lisp.

```
(car (cdr another-list))
946
```

Both `a-list` and `another-list` share the same tail (their `cdr`).

There is a predicate `pair?` that is true of pairs and false on all other types of data.

Vectors are simpler than lists. There is a constructor `vector` that can be used to make vectors and a selector `vector-ref` for accessing the elements of a vector:

```
(define a-vector
  (vector 37 63 49 21 88 56))

a-vector
#(37 63 49 21 88 56)

(vector-ref a-vector 3)
21

(vector-ref a-vector 0)
37
```

Notice that a vector is distinguished from a list on printout by the character `#` appearing before the initial parenthesis.

There is a predicate `vector?` that is true of vectors and false for all other types of data.

The elements of lists and vectors may be any kind of data, including numbers, procedures, lists, and vectors. Numerous other procedures for manipulating list-structured data and vector-structured data can be found in the Scheme online documentation.

Symbols

Symbols are a very important kind of primitive data type that we use to make programs and algebraic expressions. You probably have noticed that Scheme programs look just like lists. In fact, they are lists. Some of the elements of the lists that make up programs are symbols, such as `+` and `vector`.⁸ If we are to make programs that can manipulate programs, we need to be able to write an expression that names such a symbol. This is accomplished by the mechanism of *quotation*. The name of the symbol

⁸Symbols may have any number of characters. A symbol may not contain whitespace or a delimiter character, such as parentheses, brackets, quotation marks, comma, or `#`.

`+` is the expression `'+`, and in general the name of an expression is the expression preceded by a single quote character. Thus the name of the expression `(+ 3 a)` is `'(+ 3 a)`.

We can test if two symbols are identical by using the predicate `eq?`. For example, we can write a program to determine if an expression is a sum:

```
(define (sum? expression)
  (and (pair? expression)
    (eq? (car expression) '+)))

(sum? '(+ 3 a))
#t

(sum? '(* 3 a))
#f
```

Here `#t` and `#f` are the printed representations of the boolean values true and false.

Consider what would happen if we were to leave out the quote in the expression `(sum? '(+ 3 a))`. If the variable `a` had the value 4 we would be asking if 7 is a sum. But what we wanted to know was whether the expression `(+ 3 a)` is a sum. That is why we need the quote.

Effects

Sometimes it is necessary to perform some action, such as plot a point or print a value, in the process of a computation. Such an action is called an *effect*.⁹ For example, to see in more detail how the factorial program computes its answer we can interpolate a `write-line` statement in the body of the `factlp` internal procedure. This will print out a list of the count and the answer for each iteration:

```
(define (factorial n)
  (let factlp ((count 1) (answer 1))
    (write-line (list count answer))
    (if (> count n)
        answer
        (factlp (+ count 1) (* count answer)))))
```

⁹This is computer-science jargon: An effect is a change to something. For example, `write-line` changes the display by printing something to the display.

When we execute the modified `factorial` procedure we can watch the counter incrementing and the answer being built:

```
(factorial 6)
(1 1)
(2 1)
(3 2)
(4 6)
(5 24)
(6 120)
(7 720)
720
```

The body of every procedure or `let`, as well as the consequent of every `cond` clause, allows statements that have effects to be used. The effect statement generally has no useful value. The final expression in the body or clause produces the value that is returned. In this example the `if` expression produces the value of the `factorial`.

Assignments

Effects like printing a value or plotting a point are pretty benign, but there are more powerful (and thus dangerous) effects, called *assignments*. An assignment *changes* the value of a variable or an entry in a data structure. Almost everything we are computing are mathematical functions: for a particular input they always produce the same result. However, with assignment we can make objects that change their behavior as they are used. For example, we can make a device that counts every time we call it:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      count)))
```

Let's make two counters:

```
(define c1 (make-counter))
(define c2 (make-counter))
```

These two counters have independent local state. Calling a counter causes it to increment its local state variable, `count`, and return its value.

```
(c1)
 1
```

```
(c1)
 2
```

```
(c2)
 1
```

```
(c1)
 3
```

```
(c2)
 2
```

Assignment to variables is sometimes useful. For example, it may be useful to accumulate some objects into a list for further analysis. Here is an elegant way to do this:

```
(define (make-collector)
  (let ((lst '()))
    (cons (lambda (new)
            (set! lst (cons new lst))
            new)
          (lambda () lst))))
```

This procedure makes a pair of two procedures. The `car` of the pair is a procedure that adds to a list and the `cdr` of the pair is a procedure that reports the list that has been collected.

Let's make two collectors and play with them:

```
(define c3 (make-collector))
(define c4 (make-collector))

((car c3) 42)
42

((car c4) 'jerry)
jerry

((car c3) 28)
28

((car c3) 14)
14
```

```
((car c4) 'jack)
jack

((cdr c3))
(14 28 42)

((cdr c4))
(jack jerry)
```

It is also possible to assign to the elements of a data structure, such as a list or vector. This is unnecessary in our work so we won't tell you about how to do it! In general, it is good practice to avoid assignments whenever possible, but if you need them they are available.¹⁰

¹⁰The discipline of programming without assignments is called *functional programming*. Functional programs are generally easier to understand, and have fewer bugs than *imperative programs*.

9

Appendix: Our Notation

An adequate notation should be understood by at least two people, one of whom may be the author.
Abdus Salam (1950).

We adopt a *functional mathematical notation* that is close to that used by Spivak in his *Calculus on Manifolds* [40]. The use of functional notation avoids many of the ambiguities of traditional mathematical notation; the ambiguities of traditional notation that can impede clear reasoning in classical mechanics. Functional notation carefully distinguishes the function from the value of the function when applied to particular arguments. In functional notation mathematical expressions are unambiguous and self-contained.

We adopt a *generic arithmetic* in which the basic arithmetic operations, such as addition and multiplication, are extended to a wide variety of mathematical types. Thus, for example, the addition operator $+$ can be applied to numbers, tuples of numbers, matrices, functions, etc. Generic arithmetic formalizes the common informal practice used to manipulate mathematical objects.

We often want to manipulate aggregate quantities, such as the collection of all of the rectangular coordinates of a collection of particles, without explicitly manipulating the component parts. Tensor arithmetic provides a traditional way of manipulating aggregate objects: Indices label the parts; conventions, such as the summation convention, are introduced to manipulate the indices. We introduce a *tuple arithmetic* as an alternative way of manipulating aggregate quantities that usually lets us avoid labeling the parts with indices. Tuple arithmetic is inspired by tensor arithmetic but it is more general: not all of the components of a tuple need to be of the same size or type.

The mathematical notation is in one-to-one correspondence with expressions of the computer language *Scheme* [24]. Scheme is based on the λ -calculus [13] and directly supports the manipulation of functions. We augment Scheme with symbolic, nu-

merical, and generic features to support our applications. For a simple introduction to Scheme, see the Scheme appendix. The correspondence between the mathematical notation and Scheme requires that mathematical expressions be unambiguous and self-contained. Scheme provides immediate feedback in verification of mathematical deductions and facilitates the exploration of the behavior of systems.

Functions

The value of the function f , given the argument x , is written $f(x)$. The expression $f(x)$ denotes the value of the function at the given argument; when we wish to denote the function we write just f . Functions may take several arguments. For example, we may have the function that gives the Euclidean distance between two points in the plane given by their rectangular coordinates:

$$d(x_1, y_1, x_2, y_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}. \quad (9.1)$$

In Scheme we can write this as:

```
(define (d x1 y1 x2 y2)
  (sqrt (+ (square (- x2 x1)) (square (- y2 y1))))))
```

Functions may be composed if the range of one overlaps the domain of the other. The composition of functions is constructed by passing the output of one to the input of the other. We write the composition of two functions using the \circ operation:

$$(f \circ g) : x \mapsto (f \circ g)(x) = f(g(x)). \quad (9.2)$$

A procedure `h` that computes the cube of the sine of its argument may be defined by composing the procedures `cube` and `sin`:

```
(define h (compose cube sin))
(h 2)
.7518269446689928
```

which is the same as

```
(cube (sin 2))
.7518269446689928
```

Arithmetic is extended to the manipulation of functions: the usual mathematical operations may be applied to functions. Examples are addition and multiplication; we may add or multiply two functions if they take the same kinds of arguments and if their values can be added or multiplied:

$$(f + g)(x) = f(x) + g(x), \\ (fg)(x) = f(x)g(x). \quad (9.3)$$

A procedure *g* that multiplies the cube of its argument by the sine of its argument is

```
(define g (* cube sin))

(g 2)
7.274379414605454

(* (cube 2) (sin 2))
7.274379414605454
```

Symbolic values

As in usual mathematical notation, arithmetic is extended to allow the use of symbols that represent unknown or incompletely specified mathematical objects. These symbols are manipulated as if they had values of a known type. By default, a Scheme symbol is assumed to represent a real number. So the expression '*a*' is a literal Scheme symbol that represents an unspecified real number:

```
((compose cube sin) 'a)
(expt (sin a) 3)
```

The default printer simplifies the expression and displays it in a readable form.¹ We can use the simplifier to verify a trigonometric identity:

```
((- (+ (square sin) (square cos)) 1) 'a)
0
```

¹The procedure `print-expression` can be used in a program to print a simplified version of an expression. The default printer in the user interface incorporates the simplifier.

Just as it is useful to be able to manipulate symbolic numbers, it is useful to be able to manipulate symbolic functions. The procedure `literal-function` makes a procedure that acts as a function having no properties other than its name. By default, a literal function is defined to take one real argument and produce one real value. For example, we may want to work with a function $f : \mathbf{R} \rightarrow \mathbf{R}$:

```
((literal-function 'f) 'x)
(f x)

((compose (literal-function 'f) (literal-function 'g)) 'x)
(f (g x))
```

We can also make literal functions of multiple, possibly structured arguments that return structured values. For example, to denote a literal function named g that takes two real arguments and returns a real value ($g : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$) we may write:

```
(define g (literal-function 'g (-> (X Real Real) Real)))

(g 'x 'y)
(g x y)
```

We may use such a literal function anywhere that an explicit function of the same type may be used.

There is a whole language for describing the type of a literal function in terms of the number of arguments, the types of the arguments, and the types of the values. Here we describe a function that maps pairs of real numbers to real numbers with the expression `(-> (X Real Real) Real)`. Later we will introduce structured arguments and values and show extensions of literal functions to handle these.

Tuples

There are two kinds of tuples: *up* tuples and *down* tuples. We write tuples as ordered lists of their components; a tuple is delimited by parentheses if it is an up tuple and by square brackets if it is a down tuple. For example, the up tuple v of velocity components v^0 , v^1 , and v^2 is

$$v = (v^0, v^1, v^2). \quad (9.4)$$

The down tuple p of momentum components p_0 , p_1 , and p_2 is

$$p = [p_0, p_1, p_2]. \quad (9.5)$$

A component of an up tuple is usually identified by a superscript. A component of a down tuple is usually identified by a subscript. We use zero-based indexing when referring to tuple elements. This notation follows the usual convention in tensor arithmetic.

We make tuples with the constructors `up` and `down`:

```
(define v (up 'v^0 'v^1 'v^2))

v
(up v^0 v^1 v^2)

(define p (down 'p_0 'p_1 'p_2))

p
(down p_0 p_1 p_2)
```

Tuple arithmetic is different from the usual tensor arithmetic in that the components of a tuple may also be tuples and different components need not have the same structure. For example, a tuple structure s of phase-space states is

$$s = (t, (x, y), [p_x, p_y]). \quad (9.6)$$

It is an up tuple of the time, the coordinates, and the momenta. The time t has no substructure. The coordinates are an up tuple of the coordinate components x and y . The momentum is a down tuple of the momentum components p_x and p_y . This is written:

```
(define s (up 't (up 'x 'y) (down 'p_x 'p_y)))
```

In order to reference components of tuple structures there are selector functions, for example:

$$\begin{aligned} I(s) &= s \\ I_0(s) &= t \\ I_1(s) &= (x, y) \\ I_2(s) &= [p_x, p_y] \\ I_{1,0}(s) &= x \\ &\dots \\ I_{2,1}(s) &= p_y. \end{aligned} \quad (9.7)$$

The sequence of integer subscripts on the selector describes the access chain to the desired component.

The procedure `component` is the general selector procedure that implements the selector functions. For example, $I_{0,1}$ is implemented by (`component 0 1`):

```
((component 0 1) (up (up 'a 'b) (up 'c 'd)))
b
```

To access a component of a tuple we may also use the selector procedure `ref`, which takes a tuple and an index and returns the indicated element of the tuple:

```
(ref (up 'a 'b 'c) 1)
b
```

We use zero-based indexing everywhere. The procedure `ref` can be used to access any substructure of a tree of tuples:

```
(ref (up (up 'a 'b) (up 'c 'd)) 0 1)
b
```

Two up tuples of the same length may be added or subtracted, elementwise, to produce an up tuple, if the components are compatible for addition. Similarly, two down tuples of the same length may be added or subtracted, elementwise, to produce a down tuple, if the components are compatible for addition.

Any tuple may be multiplied by a number by multiplying each component by the number. Numbers may, of course, be multiplied. Tuples that are compatible for addition form a vector space.

For convenience we define the square of a tuple to be the sum of the squares of the components of the tuple. Tuples can be multiplied, as described below, but the square of a tuple is not the product of the tuple with itself.

The meaning of multiplication of tuples depends on the structure of the tuples. Two tuples are compatible for contraction if they are of opposite types, they are of the same length, and corresponding elements have the following property: either they are both tuples and are compatible for contraction, or one of them is not a tuple. If two tuples are compatible for contraction then generic multiplication is interpreted as contraction: the result is the sum of the products of corresponding components of the tuples. For example, p and v introduced in equations (9.4) and (9.5) above are compatible for contraction; the product is

$$pv = p_0v^0 + p_1v^1 + p_2v^2. \quad (9.8)$$

So the product of tuples that are compatible for contraction is an inner product. Using the tuples p and v defined above gives us

$$\begin{aligned} (*\ p\ v) \\ (+\ (*\ p_0\ v^0)\ (*\ p_1\ v^1)\ (*\ p_2\ v^2)) \end{aligned}$$

Contraction of tuples is commutative: $pv = vp$. Caution: Multiplication of tuples that are compatible for contraction is, in general, not associative. For example, let $u = (5, 2)$, $v = (11, 13)$, and $g = [[3, 5], [7, 9]]$. Then $u(gv) = 964$, but $(ug)v = 878$. The expression ugv is ambiguous. An expression that has this ambiguity does not occur in this book.

The rule for multiplying two structures that are not compatible for contraction is simple. If A and B are not compatible for contraction, the product AB is a tuple of type B whose components are the products of A and the components of B . The same rule is applied recursively in multiplying the components. So if $B = (B^0, B^1, B^2)$, the product of A and B is

$$AB = (AB^0, AB^1, AB^2). \quad (9.9)$$

If A and C are not compatible for contraction and $C = [C_0, C_1, C_2]$, the product is

$$AC = [AC_0, AC_1, AC_2]. \quad (9.10)$$

Tuple structures can be made to represent linear transformations. For example, the rotation commonly represented by the matrix

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (9.11)$$

can be represented as a tuple structure:²

$$\left[\begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix} \begin{pmatrix} -\sin \theta \\ \cos \theta \end{pmatrix} \right]. \quad (9.12)$$

²To emphasize the relationship of simple tuple structures to matrix notation we often format up tuples as vertical arrangements of components and down tuples as horizontal arrangements of components. However, we could just as well have written this tuple as $[(\cos \theta, \sin \theta), (-\sin \theta, \cos \theta)]$.

Such a tuple is compatible for contraction with an up tuple that represents a vector. So, for example:

$$\left[\begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix} \begin{pmatrix} -\sin \theta \\ \cos \theta \end{pmatrix} \right] \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{pmatrix}. \quad (9.13)$$

Two tuples that represent linear transformations, though not compatible for contraction, may also be combined by multiplication. In this case the product represents the composition of the linear transformations. For example, the product of the tuples representing two rotations is

$$\begin{aligned} & \left[\begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix} \begin{pmatrix} -\sin \theta \\ \cos \theta \end{pmatrix} \right] \left[\begin{pmatrix} \cos \varphi \\ \sin \varphi \end{pmatrix} \begin{pmatrix} -\sin \varphi \\ \cos \varphi \end{pmatrix} \right] \\ &= \left[\begin{pmatrix} \cos(\theta + \varphi) \\ \sin(\theta + \varphi) \end{pmatrix} \begin{pmatrix} -\sin(\theta + \varphi) \\ \cos(\theta + \varphi) \end{pmatrix} \right]. \end{aligned} \quad (9.14)$$

Multiplication of tuples that represent linear transformations is associative but generally not commutative, just as the composition of the transformations is associative but not generally commutative.

Derivatives

The derivative of a function f is a function, denoted by Df . Our notational convention is that D is a high-precedence operator. Thus D operates on the adjacent function before any other application occurs: $Df(x)$ is the same as $(Df)(x)$. Higher-order derivatives are described by exponentiating the derivative operator. Thus the n th derivative of a function f is notated as $D^n f$.

The procedure for producing the derivative of a function is named `D`. The derivative of the `sin` procedure is a procedure that computes `cos`:

```
(define derivative-of-sine (D sin))

(derivative-of-sine 'x)
(cos x)
```

The derivative of a function f is the function Df whose value for a particular argument is something that can be multiplied by an increment Δx in the argument to get a linear approximation to the increment in the value of f :

$$f(x + \Delta x) \approx f(x) + Df(x)\Delta x. \quad (9.15)$$

For example, let f be the function that cubes its argument ($f(x) = x^3$); then Df is the function that yields three times the square of its argument ($Df(y) = 3y^2$). So $f(5) = 125$ and $Df(5) = 75$. The value of f with argument $x + \Delta x$ is

$$f(x + \Delta x) = (x + \Delta x)^3 = x^3 + 3x^2\Delta x + 3x\Delta x^2 + \Delta x^3 \quad (9.16)$$

and

$$Df(x)\Delta x = 3x^2\Delta x. \quad (9.17)$$

So $Df(x)$ multiplied by Δx gives us the term in $f(x + \Delta x)$ that is linear in Δx , providing a good approximation to $f(x + \Delta x) - f(x)$ when Δx is small.

Derivatives of compositions obey the chain rule:

$$D(f \circ g) = ((Df) \circ g) \cdot Dg. \quad (9.18)$$

So at x ,

$$(D(f \circ g))(x) = Df(g(x)) \cdot Dg(x). \quad (9.19)$$

D is an example of an *operator*. An operator is like a function except that multiplication of operators is interpreted as composition, whereas multiplication of functions is multiplication of the values (see equation 9.3). If D were an ordinary function, then the rule for multiplication would imply that D^2f would just be the product of Df with itself, which is not what is intended. A product of a number and an operator scales the operator. So, for example

```
(((* 5 D) cos) 'x)
(* -5 (sin x))
```

Arithmetic is extended to allow manipulation of operators. A typical operator is $(D+I)(D-I) = D^2 - I$, where I is the identity operator, which subtracts a function from its second derivative. Such an operator can be constructed and used as follows:

```
(((* (+ D I) (- D I)) (literal-function 'f)) 'x)
(+ (((expt D 2) f) x) (* -1 (f x)))
```

Derivatives of functions of multiple arguments

The derivative generalizes to functions that take multiple arguments. The derivative of a real-valued function of multiple arguments is an object whose contraction with the tuple of increments in the arguments gives a linear approximation to the increment in the function's value.

A function of multiple arguments can be thought of as a function of an up tuple of those arguments. Thus an incremental argument tuple is an up tuple of components, one for each argument position. The derivative of such a function is a down tuple of the partial derivatives of the function with respect to each argument position.

Suppose we have a real-valued function g of two real-valued arguments, and we want to approximate the increment in the value of g from its value at x, y . If the arguments are incremented by the tuple $(\Delta x, \Delta y)$ we compute:

$$\begin{aligned} Dg(x, y) \cdot (\Delta x, \Delta y) &= [\partial_0 g(x, y), \partial_1 g(x, y)] \cdot (\Delta x, \Delta y) \\ &= \partial_0 g(x, y) \Delta x + \partial_1 g(x, y) \Delta y. \end{aligned} \quad (9.20)$$

Using the two-argument literal function `g` defined on page 512, we have:

```
((D g) 'x 'y)
(down (((partial 0) g) x y) (((partial 1) g) x y))
```

In general, partial derivatives are just the components of the derivative of a function that takes multiple arguments (or structured arguments or both; see below). So a partial derivative of a function is a composition of a component selector and the derivative of that function.³ Indeed:

$$\partial_0 g = I_0 \circ Dg \quad (9.21)$$

$$\partial_1 g = I_1 \circ Dg. \quad (9.22)$$

Concretely, if

$$g(x, y) = x^3 y^5 \quad (9.23)$$

³Partial derivative operators such as `(partial 2)` are operators, so `(expt (partial 1) 2)` is a second partial derivative.

then

$$Dg(x, y) = [3x^2y^5, 5x^3y^4] \quad (9.24)$$

and the first-order approximation of the increment for changing the arguments by Δx and Δy is

$$\begin{aligned} g(x + \Delta x, y + \Delta y) - g(x, y) &\approx [3x^2y^5, 5x^3y^4] \cdot (\Delta x, \Delta y) \\ &= 3x^2y^5\Delta x + 5x^3y^4\Delta y. \end{aligned} \quad (9.25)$$

Partial derivatives of compositions also obey a chain rule:

$$\partial_i(f \circ g) = ((Df) \circ g) \cdot \partial_i g. \quad (9.26)$$

So if x is a tuple of arguments, then

$$(\partial_i(f \circ g))(x) = Df(g(x)) \cdot \partial_i g(x). \quad (9.27)$$

Mathematical notation usually does not distinguish functions of multiple arguments and functions of the tuple of arguments. Let $h((x, y)) = g(x, y)$. The function h , which takes a tuple of arguments x and y , is not distinguished from the function g that takes arguments x and y . We use both ways of defining functions of multiple arguments. The derivatives of both kinds of functions are compatible for contraction with a tuple of increments to the arguments. Scheme comes in handy here:

```
(define (h s)
  (g (ref s 0) (ref s 1)))

(h (up 'x 'y))
(g x y)

((D g) 'x 'y)
(down (((partial 0) g) x y) (((partial 1) g) x y))

((D h) (up 'x 'y))
(down (((partial 0) g) x y) (((partial 1) g) x y))
```

A phase-space state function is a function of time, coordinates, and momenta. Let H be such a function. The value of H is $H(t, (x, y), [p_x, p_y])$ for time t , coordinates (x, y) , and momenta $[p_x, p_y]$. Let s be the phase-space state tuple as in (9.6):

$$s = (t, (x, y), [p_x, p_y]). \quad (9.28)$$

The value of H for argument tuple s is $H(s)$. We use both ways of writing the value of H .

We often show a function of multiple arguments that include tuples by indicating the boundaries of the argument tuples with semicolons and separating their components with commas. If H is a function of phase-space states with arguments t , (x, y) , and $[p_x, p_y]$, we may write $H(t; x, y; p_x, p_y)$. This notation loses the up/down distinction, but our semicolon-and-comma notation is convenient and reasonably unambiguous.

The derivative of H is a function that produces an object that can be contracted with an increment in the argument structure to produce an increment in the function's value. The derivative is a down tuple of three partial derivatives. The first partial derivative is the partial derivative with respect to the numerical argument. The second partial derivative is a down tuple of partial derivatives with respect to each component of the up-tuple argument. The third partial derivative is an up tuple of partial derivatives with respect to each component of the down-tuple argument:

$$\begin{aligned} DH(s) &= [\partial_0 H(s), \partial_1 H(s), \partial_2 H(s)] \\ &= [\partial_0 H(s), [\partial_{1,0} H(s), \partial_{1,1} H(s)], (\partial_{2,0} H(s), \partial_{2,1} H(s))], \end{aligned} \quad (9.29)$$

where $\partial_{1,0}$ indicates the partial derivative with respect to the first component (index 0) of the second argument (index 1) of the function, and so on. Indeed, $\partial_z F = I_z \circ DF$ for any function F and access chain z . So, if we let Δs be an incremental phase-space state tuple,

$$\Delta s = (\Delta t, (\Delta x, \Delta y), [\Delta p_x, \Delta p_y]), \quad (9.30)$$

then

$$\begin{aligned} DH(s)\Delta s &= \partial_0 H(s)\Delta t \\ &\quad + \partial_{1,0} H(s)\Delta x + \partial_{1,1} H(s)\Delta y \\ &\quad + \partial_{2,0} H(s)\Delta p_x + \partial_{2,1} H(s)\Delta p_y. \end{aligned} \quad (9.31)$$

Caution: Partial derivative operators with respect to different structured arguments generally do not commute.

In Scheme we must make explicit choices. We usually assume that phase-space state functions are functions of the tuple. For example,

```
(define H
  (literal-function 'H
    (-> (UP Real (UP Real Real) (DOWN Real Real)) Real)))

(H s)
(H (up t (up x y) (down p-x p-y)))

((D H) s)
(down
 (((partial 0) H) (up t (up x y) (down p-x p-y)))
 (down (((partial 1 0) H) (up t (up x y) (down p-x p-y)))
        (((partial 1 1) H) (up t (up x y) (down p-x p-y))))
 (up (((partial 2 0) H) (up t (up x y) (down p-x p-y)))
      (((partial 2 1) H) (up t (up x y) (down p-x p-y)))))
```

Structured results

Some functions produce structured outputs. A function whose output is a tuple is equivalent to a tuple of component functions each of which produces one component of the output tuple.

For example, a function that takes one numerical argument and produces a structure of outputs may be used to describe a curve through space. The following function describes a helical path around the z -axis in three-dimensional space:

$$h(t) = (\cos t, \sin t, t) = (\cos, \sin, I)(t). \quad (9.32)$$

The derivative is just the up tuple of the derivatives of each component of the function:

$$Dh(t) = (-\sin t, \cos t, 1). \quad (9.33)$$

We can write

```
(define (helix t)
  (up (cos t) (sin t) t))
```

or just

```
(define helix (up cos sin identity))
```

Its derivative is just the up tuple of the derivatives of each component of the function:

```
((D helix) 't)
(up (* -1 (sin t)) (cos t) 1)
```

In general, a function that produces structured outputs is just treated as a structure of functions, one for each of the components. The derivative of a function of structured inputs that produces structured outputs is an object that when contracted with an incremental input structure produces a linear approximation to the incremental output. Thus, if we define function g by

$$g(x, y) = ((x + y)^2, (y - x)^3, e^{x+y}), \quad (9.34)$$

then the derivative of g is

$$Dg(x, y) = \left[\begin{pmatrix} 2(x + y) \\ -3(y - x)^2 \\ e^{x+y} \end{pmatrix}, \begin{pmatrix} 2(x + y) \\ 3(y - x)^2 \\ e^{x+y} \end{pmatrix} \right]. \quad (9.35)$$

In Scheme:

```
(define (g x y)
  (up (square (+ x y)) (cube (- y x)) (exp (+ x y)))

  ((D g) 'x 'y)
  (down (up (+ (* 2 x) (* 2 y))
            (+ (* -3 (expt x 2)) (* 6 x y) (* -3 (expt y 2)))
            (* (exp y) (exp x)))
        (up (+ (* 2 x) (* 2 y))
            (+ (* 3 (expt x 2)) (* -6 x y) (* 3 (expt y 2)))
            (* (exp y) (exp x))))
```

Caution must be exercised when taking the derivative of the product of functions that each produce structured results. The problem is that the usual product rule does not hold. Let f and g be functions of x whose results are compatible for contraction to a number. The increment of f for an increment Δx of x is $Df(x)\Delta x$, and similarly for g . The increment of the product fg is $D(fg)(x)\Delta x$, but expanded in terms of the derivative of f and g the increment is $(Df(x)\Delta x)g(x) + f(x)(Dg(x)\Delta x)$. It is not $((Df)(x)g(x) + f(x)(Dg(x)))\Delta x$. The reason is that the shape of the derivative of f is such that $Df(x)$ should be multiplied by Δx rather than $g(x)$.

Exercise 9.1: Chain rule

Let $F(x, y) = x^2y^3$, $G(x, y) = (F(x, y), y)$, and $H(x, y) = F(F(x, y), y)$, so that $H = F \circ G$.

- a. Compute $\partial_0 F(x, y)$ and $\partial_1 F(x, y)$.
- b. Compute $\partial_0 F(F(x, y), y)$ and $\partial_1 F(F(x, y), y)$.
- c. Compute $\partial_0 G(x, y)$ and $\partial_1 G(x, y)$.
- d. Compute $DF(a, b)$, $DG(3, 5)$ and $DH(3a^2, 5b^3)$.

Exercise 9.2: Computing derivatives

We can represent functions of multiple arguments as procedures in several ways, depending upon how we wish to use them. The simplest idea is to identify the procedure arguments with the function's arguments.

For example, we could write implementations of the functions that occur in exercise 9.1 as follows:

```
(define (f x y)
  (* (square x) (cube y)))

(define (g x y)
  (up (f x y) y))

(define (h x y)
  (f (f x y) y))
```

With this choice it is awkward to compose a function that takes multiple arguments, such as f , with a function that produces a tuple of those arguments, such as g . Alternatively, we can represent the function arguments as slots of a tuple data structure, and then composition with a function that produces such a data structure is easy. However, this choice requires the procedures to build and take apart structures.

For example, we may define procedures that implement the functions above as follows:

```
(define (f v)
  (let ((x (ref v 0))
        (y (ref v 1)))
    (* (square x) (cube y)))

(define (g v)
  (let ((x (ref v 0))
        (y (ref v 1)))
    (up (f v) y)))

(define h (compose f g))
```

Repeat exercise 9.1 using the computer. Explore both implementations of multiple-argument functions.

References

- [1] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press and McGraw-Hill, 1996.
- [2] Ralph H. Abraham and Jerrold E. Marsden, *Foundations of Mechanics*, 2nd edition, Addison-Wesley, 1978.
- [3] Ralph H. Abraham, Jerrold E. Marsden, and Tudor Rațiu, *Manifolds, Tensor Analysis, and Applications*, 2nd edition, Springer Verlag, 1993.
- [4] V. I. Arnold, “Small Denominators and Problems of Stability of Motion in Classical and Celestial Mechanics,” *Russian Math. Surveys*, **18**, 6 (1963).
- [5] V. I. Arnold, *Mathematical Methods of Classical Mechanics*, Springer Verlag, 1980.
- [6] V. I. Arnold, V. V. Kozlov, and A. I. Neishtadt, “Mathematical Aspects of Classical and Celestial Mechanics,” *Dynamical Systems III*, Springer Verlag, 1988.
- [7] June Barrow-Green, “Poincaré and the Three Body Problem,” *History of Mathematics*, vol. 11, American Mathematical Society, London Mathematical Society, 1997.
- [8] Max Born, *Vorlesungen über Atommechanik*, Springer, 1925–30.
- [9] Constantin Carathéodory, *Calculus of variations and partial differential equations of the first order*, (translated by Robert B. Dean and Julius J. Brandstatter), Holden-Day, 1965–67.
- [10] Constantin Carathéodory, *Geometrische Optik*, in *Ergebnisse der Mathematik und ihrer Grenzgebiete*, Bd. 4, Springer, 1937.
- [11] Élie Cartan, *Leçons sur les invariants intégraux*, Hermann, 1922; reprinted 1971.
- [12] Boris V. Chirikov, “A Universal Instability of Many-Dimensional Oscillator Systems,” *Physics Reports* **52**, 5, pp. 263–379 (1979).
- [13] Alonzo Church, *The Calculi of Lambda-Conversion*, Princeton University Press, 1941.
- [14] Richard Courant and David Hilbert, *Methods of Mathematical Physics*, 2 vols., Wiley-Interscience, 1957.
- [15] Jean Dieudonné, *Treatise on Analysis*, Academic Press, 1969.
- [16] Albert Einstein, *Relativity, the Special and General Theory*, Crown Publishers, 1961.

- [17] Hans Freudenthal, *Didactical Phenomenology of Mathematical Structures*, Kluwer, 1983.
- [18] Giovanni Gallavotti, *The Elements of Mechanics*, Springer Verlag, 1983.
- [19] F. R. Gantmakher, *Lektsii po analiticheskoi mekhanike* (Lectures on analytical mechanics), Fizmatgiz, 1960; English translation by G. Yankovsky, Mir Publishing, 1970.
- [20] Herbert Goldstein, *Classical Mechanics*, 2nd edition, Addison-Wesley, 1980.
- [21] Michel Hénon, “Numerical Exploration of Hamiltonian Systems,” *Chaotic Behavior of Deterministic Systems*, North-Holland Publishing Company, 1983.
- [22] Michel Hénon and Carl Heiles, “The Applicability of the Third Integral of Motion: Some Numerical Experiments,” *Astronomical Journal*, **69**, pp. 73–79 (1964).
- [23] Robert Hermann, *Differential Geometry and the Calculus of Variations*, Academic Press, 1968.
- [24] IEEE Std 1178-1990, *IEEE Standard for the Scheme Programming Language*, Institute of Electrical and Electronic Engineers, Inc., 1991.
- [25] E. L. Ince, *Ordinary Differential Equations*, Longmans, Green and Co., 1926; Dover Publications, 1956.
- [26] Jorge V. José and Eugene J. Saletan, *Classical Dynamics: A Contemporary Approach*, Cambridge University Press, 1998.
- [27] Res Jost, “Poisson Brackets: An Unpedagogical Lecture,” in *Reviews of Modern Physics*, **36**, p. 572 (1964).
- [28] P. E. B. Jourdain, *The Principle of Least Action*, Open Court Publishing Company, 1913.
- [29] Cornelius Lanczos, *The Variational Principles of Mechanics*, 4th edition, University of Toronto Press, 1970; Dover Publications, 1982.
- [30] L. D. Landau and E. M. Lifshitz, *Mechanics*, 3rd edition, *Course of Theoretical Physics*, vol. 1, Pergamon Press, 1976.
- [31] Edward Lorenz, “Deterministic Nonperiodic Flow,” *Journal of Atmospheric Science* **20**, p. 130 (1963).
- [32] Jerrold E. Marsden and Tudor S. Rāiu, *Introduction to Mechanics and Symmetry*, Springer Verlag, 1994.
- [33] Philip Morse and Hermann Feshbach, *Methods of Theoretical Physics*, 2 vols., McGraw-Hill, 1953.
- [34] Lothar Nordheim, *The Principles of Mechanics*, in *Handbuch der Physik*, vol. 2, Springer, 1927.
- [35] Henri Poincaré, *Les Méthodes nouvelles de la Mécanique céleste*, Paris, 1892; Dover Publications, 1957; English translation by the National Aeronautics and Space Administration, technical report NASA TT F-452.

-
- [36] H. C. Plummer, *An Introductory Treatise on Dynamical Astronomy*, Cambridge University Press, 1918; Dover Publications, 1960.
 - [37] Florian Scheck, *Mechanics, From Newton's Laws to Deterministic Chaos*, 2nd edition, Springer-Verlag, 1994.
 - [38] P. Kenneth Seidelmann, editor, *Explanatory Supplement to the Astronomical Almanac*, University Science Books, 1992.
 - [39] Jean-Marie Souriau, *Structure des Systèmes Dynamiques*, Dunod Université, Paris, 1970; English translation: Birkhäuser Boston, 1998.
 - [40] Michael Spivak, *Calculus on Manifolds*, W. A. Benjamin, 1965.
 - [41] Stanly Steinberg, "Lie Series, Lie Transformations, and Their Applications," in *Lie Methods in Optics*, J. Sánchez Mondragón and K. B. Wolf, editors, Springer Verlag, 1986, pp. 45–103.
 - [42] Shlomo Sternberg, *Celestial Mechanics*, W. A. Benjamin, 1969.
 - [43] E. C. G. Sudarshan and N. Mukunda, *Classical Dynamics: A Modern Perspective*, John Wiley & Sons, 1974.
 - [44] J. B. Taylor, unpublished, 1968.
 - [45] Walter Thirring, *A Course in Mathematical Physics 1: Classical Dynamical Systems*, translated by Evans M. Harell, Springer-Verlag, 1978.
 - [46] E. T. Whittaker, *A Treatise on Analytical Dynamics*, Cambridge University Press, 1937.
 - [47] J. Wisdom, "The Origin of the Kirkwood Gaps: A Mapping for Asteroidal Motion Near the 3/1 Commensurability," *Astron. J.*, **87**, p. 557 (1982).
 - [48] J. Wisdom, "A Mapping for the Hénon-Heiles System," unpublished notes, (1987).
 - [49] J. Wisdom and M. Holman, "Symplectic Maps for the N-Body Problem," *Astron. J.*, **102**, p. 1528 (1991).

List of Exercises

1.1	5	1.10	32	1.19	53	1.28	68	1.37	97
1.2	8	1.11	35	1.20	53	1.29	68	1.38	98
1.3	13	1.12	36	1.21	58	1.30	78	1.39	107
1.4	18	1.13	36	1.22	59	1.31	78	1.40	108
1.5	23	1.14	43	1.23	59	1.32	83	1.41	108
1.6	23	1.15	46	1.24	59	1.33	84	1.42	110
1.7	28	1.16	47	1.25	62	1.34	86	1.43	116
1.8	28	1.17	52	1.26	64	1.35	90	1.44	117
1.9	32	1.18	52	1.27	66	1.36	94		
2.1	129	2.6	134	2.11	143	2.16	164	2.21	193
2.2	129	2.7	135	2.12	154	2.17	169		
2.3	129	2.8	137	2.13	156	2.18	176		
2.4	129	2.9	138	2.14	163	2.19	188		
2.5	134	2.10	143	2.15	164	2.20	193		
3.1	201	3.5	211	3.9	236	3.13	280		
3.2	201	3.6	214	3.10	248	3.14	282		
3.3	205	3.7	214	3.11	272	3.15	282		
3.4	209	3.8	233	3.12	276	3.16	283		
4.1	294	4.3	297	4.5	307	4.7	322	4.9	333
4.2	297	4.4	302	4.6	309	4.8	329	4.10	333
5.1	341	5.6	356	5.11	389	5.16	398	5.21	408
5.2	341	5.7	356	5.12	393	5.17	399		
5.3	346	5.8	357	5.13	393	5.18	399		
5.4	346	5.9	357	5.14	394	5.19	402		
5.5	353	5.10	364	5.15	394	5.20	407		
6.1	413	6.4	425	6.7	445	6.10	452		
6.2	417	6.5	430	6.8	446	6.11	453		
6.3	423	6.6	430	6.9	446	6.12	453		
7.1	468	7.2	483	7.3	483	7.4	496	7.5	496
9.1	523	9.2	523						

Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

Donald E. Knuth, *Fundamental Algorithms*
(Volume 1 of *The Art of Computer Programming*)

Page numbers for Scheme procedure definitions are in italics.
Page numbers followed by *n* indicate footnotes.

- 0, for all practical purposes, 20 *n*.
See also Zero-based indexing
 - o (composition), 7 *n*, 510
 - $\Gamma[q]$
 - for local tuple, 11
 - Lagrangian state path, 203
 - γ (configuration-path function), 7
 - δ function, 454 (ex. 6.12)
 - δ_η (variation operator), 26
 - λ -calculus, 509
 - λ -expression, 498–499
 - λ -notation, 498 *n*
 - $\Pi_L[q]$ (Hamiltonian state path), 203
 - χ (coordinate function), 7
 - σ (phase-space path), 218
 - ω matrix, 124
 - $\vec{\omega}$ (angular velocity), 124
 - ω (symplectic 2-form), 359
 - C (local-tuple transformation), 44
 - C_H (canonical phase-space transformation), 337 *n*
 - D . *See* Derivative
 - D_t (total time derivative), 64
 - ∂ . *See* Partial derivative
 - E (Euler–Lagrange operator), 98
 - \mathcal{E} (energy state function), 82
 - F_1 – F_4 . *See also* Generating functions
 - $F_1(t, q, q')$, 373
 - $F_2(t, q, p')$, 373
 - $F_3(t, p, q')$, 374
 - $F_4(t, p, p')$, 374
 - H (Hamiltonian), 199
 - I (identity operator), 517
 - I with subscript (selector), 64 *n*, 513
 - \tilde{J} (shuffle function), 350
 - \mathbf{J}, \mathbf{J}_n (symplectic unit), 301, 355
 - L (Lagrangian), 11
 - L (Lie derivative), 447
 - P (momentum selector), 199, 220
 - \mathcal{P} (momentum state function), 79
 - Q (coordinate selector), 220
 - Q (velocity selector), 64
 - q (coordinate path), 7
 - S (action), 10
 - Lagrangian, 12
 - ' (quote in Scheme), 505
 - , in tuple, 520
 - : names starting with, 21 *n*
 - ; in tuple, 31 *n*, 520
 - # in Scheme, 504
 - { } for Poisson brackets, 218
 - [] for down tuples, 512
 - [] for functional arguments, 10 *n*
 - () for up tuples, 512
 - in Scheme, 497, 498 *n*, 503
- Action, 9–13
computing, 14–23
coordinate-independence of, 17
free particle, 14–20

- Action (*continued*)
 generating functions and,
 421–425
 Hamilton–Jacobi equation and,
 421–425
 Lagrangian, 12
 minimizing, 18–23
 parametric, 21
 principles (*see* Principle of stationary action)
S, 10
 time evolution and, 423–425,
 435–437
 variation of, 28
- Action-angle coordinates, 311
 Hamiltonian in, 311
 Hamilton–Jacobi equation and,
 413
 Hamilton’s equations in, 311
 harmonic oscillator in, 346 (eq.
 5.31)
 perturbation of Hamiltonian,
 316, 458
 surfaces of section in, 313
- Action principle. *See* Principle of stationary action
- Alphabet, insufficient size of, 15 *n*
- Alternative in conditional, 501
- angle-axis->rotation-matrix**,
 184
- Angles, Euler. *See* Euler angles
- Angular momentum. *See also* Vector angular momentum
 conservation of, 43, 80, 86,
 142–143
 equilibrium points for, 149
 Euler’s equations and, 151–153
 in terms of principal moments
 and angular velocity, 136
 kinetic energy in terms of, 148
 Lie commutation relations for,
 452 (ex. 6.10)
 as Lie generator of rotations,
 440
 of free rigid body, 146–150,
 151–153
 of rigid body, 135–137
 sphere of, 148
 z component of, 85
- Angular velocity vector ($\vec{\omega}$), 124,
 139
 Euler’s equations for, 151–153
 kinetic energy in terms of, 131,
 134
 representation of, 123–126
- Anomaly, true, 171 *n*
- antisymmetric->column-matrix**,
 126
- Antisymmetry of Poisson bracket, 220
- Area preservation by maps, 278
- Liouville’s theorem and, 272
- Poincaré–Cartan integral invariant and, 434–435
 of surfaces of section, 272,
 434–435
- Arguments. *See also* Function(s); Functional arguments
 active vs. passive in Legendre transformation, 208
 in Scheme, 497
- Arithmetic
 generic, 16 *n*, 509
 on functions, 18 *n*, 511
 on operators, 34 *n*, 517
 on procedures, 19 *n*
 on symbolic values, 511
 on tuples, 509, 513–516
- Arnold, V. I., xiii, xv *n*, 113. *See also* Kolmogorov–Arnold–Moser theorem
- Assignment in Scheme, 506–508
- Associativity and
 non-associativity of tuple multiplication, 515, 516
- Asteroids, rotational alignment of, 151
- Astronomy. *See* Celestial objects
- Asymptotic trajectories, 223, 287, 302
- Atomic scale, 8 *n*
- Attractor, 274
- Autonomous systems, 82. *See also* Extended phase space
 surfaces of section for, 248–263
- Awake top, 231

- Axes, principal, 133
of this dense book, 135 (ex. 2.7), 150
- Axisymmetric potential of galaxy, 250
- Axisymmetric top
awake, 231
behavior of, 161–165, 231–232
conserved quantities for, 160
degrees of freedom of, 5 (ex. 1.1)
Euler angles for, 159
Hamiltonian treatment of, 228–233
kinetic energy of, 159
Lagrangian treatment of, 157–165
nutation of, 162 (fig. 2.5), 164 (ex. 2.15)
potential energy of, 160
precession of, 119, 162 (fig. 2.6), 164 (ex. 2.16)
rotation of, 119
sleeping, 231
symmetries of, 228
- Baker, Henry. *See* Baker–Campbell–Hausdorff formula
- Baker–Campbell–Hausdorff formula, 453 (ex. 6.11)
- Banana. *See* Book
- Barrow-Green, June, 457
- Basin of attraction, 274
- Bicycle wheel, 156 (ex. 2.13)
- Birkhoff, George David. *See* Poincaré–Birkhoff theorem
- bisect (bisection search), 321, 326
- Body components of vector, 134
- Boltzmann, Ludwig, 12 *n*, 203 *n*, 274 *n*
- Book
banana-like behavior of, 128
rotation of, 119, 150
- Brackets. *See also* Poisson brackets
for down tuples, 512
for functional arguments, 10 *n*
- bulirsch-stoer, 145
Bulirsch–Stoer integration method, 74 *n*
- Butterfly effect, 241 *n*
- C* (local-tuple transformation), 44
- C_H* (canonical phase-space transformation), 337 *n*
- Campbell, John. *See* Baker–Campbell–Hausdorff formula
- canonical?, 344
- Canonical-H?, 348
- Canonical-K?, 348
- canonical-transform?, 351
- Canonical condition, 342–352
Poisson brackets and, 352–353
- Canonical equations. *See* Hamilton's equations
- Canonical heliocentric coordinates, 409 (ex. 5.21)
- Canonical perturbation theory.
See Perturbation theory
- Canonical plane, 362 *n*
- Canonical transformations, 335–336. *See also* Generating functions; Symplectic transformations
composition of, 346 (ex. 5.4), 381, 393 (ex. 5.12)
conditions for, 342–357
for driven pendulum, 392
general, 342–357
group properties of, 346 (ex. 5.4)
for harmonic oscillator, 344
invariance of antisymmetric bilinear form under, 359–362
invariance of phase volume under, 358–359
invariance of Poisson brackets under, 358
- invariants of, 357–364 (*see also* Integral invariants)
- as Lie series, 448
- Lie transforms (*see* Lie transforms)

- Canonical transformations
(continued)
- point transformations (*see*
 Point transformations)
- polar-canonical (*see*
 Polar-canonical
 transformation)
- to rotating coordinates,
 348–349, 377–378
- time evolution as, 426–437
- total time derivative and,
 390–393
- Cantor, cantori, 244 *n*, 330
- car, 503
- Cartan, Élie. *See also*
 Poincaré–Cartan integral
 invariant
- Cauchy, Augustin Louis, 39 *n*
- cdr, 503
- Celestial objects. *See also*
 Asteroids; Comets; Earth;
 Galaxy; Hyperion; Jupiter;
 Mercury; Moon; Phobos;
 Planets
- rotation of, 151, 165, 170–171
- Center of mass, 121
- in two-body problem, 381
- Jacobi coordinates and, 409 (ex.
 5.21)
- kinetic energy and, 121
- vector angular momentum and,
 135
- Central force
- collapsing orbits, 389 (ex. 5.11)
- epicyclic motion, 381–389
- gravitational, 31
- in 2 dimensions, 40, 227–228,
 381–389
- in 3 dimensions, 47 (ex. 1.16),
 84
- Lie series for motion in, 450
- orbits, 78 (ex. 1.30)
- reduced phase space for motion
 in, 405–407
- Central potential. *See* Central
 force
- Centrifugal force, 47, 49
- Chain rule
- for derivatives, 517, 523 (ex.
 9.1)
- for partial derivatives, 519, 523
 (ex. 9.1)
- for total time derivatives, 64
 (ex. 1.26)
- in traditional notation, xiv *n*
- for variations, 27 (eq. 1.26)
- Chaotic motion, 241. *See also*
 Exponential divergence
 homoclinic tangle and, 307
 in Hénon–Heiles problem, 259
 in restricted three-body
 problem, 283 (ex. 3.16)
- in spin-orbit coupling, 282 (ex.
 3.15), 496 (ex. 7.5)
- near separatrices, 290, 484, 486
- of Hyperion, 151, 170–176
- of non-axisymmetric top, 263
- of periodically driven
 pendulum, 76, 243
- overlapping resonances and, 488
- Characteristic exponent, 293
- Characteristic multiplier, 296
- Chirikov, Boris V., 278 *n*
- Chirikov–Taylor map, 278 *n*
- Church, Alonzo, 498 *n*
- Colon, names starting with, 21 *n*
- Comets, rotation of, 151
- Comma in tuple, 520
- Commensurability, 312. *See also*
 Resonance
- islands and, 309
- of pendulum period with drive,
 289, 290
- periodic orbits and, 309, 316
- rational rotation number and,
 316
- small denominators and, 475
- Commutativity. *See also*
 Non-commutativity
- of some tuple multiplication,
 515
- of variation (δ) with
 differentiation and integration,
 27

- Commutator, 451
of angular-momentum Lie operators, 452 (ex. 6.10)
Jacobi identity for, 451
of Lie derivative, 452 (ex. 6.10)
Poisson brackets and, 452 (ex. 6.10)
- compatible-shape**, 351 *n*
Compatible shape, 351 *n*
component, 15 *n*, 514
compose, 500
Composition
 of canonical transformations, 346 (ex. 5.4), 381, 393 (ex. 5.12)
 of functions, 7 *n*, 510, 523 (ex. 9.2)
 of Lie transforms, 451
 of linear transformations, 516
 of operators, 517
 of rotations, 123, 187
Compound data in Scheme, 502–504
cond, 500
Conditionals in Scheme, 500–501
Configuration, 4
Configuration manifold, 7 *n*
Configuration path. *See* Path
Configuration space, 4–5
Conjugate momentum, 79
 non-uniqueness of, 239
cons, 503
Consequent in conditional, 500
Conserved quantities, 78, 195.
 See also Hénon–Heiles problem, integrals of motion
 angular momentum, 43, 80, 86, 142–143
 coordinate choice and, 79–81
 cyclic coordinates and, 80
 energy, 81–83, 142, 211
 Jacobi constant, 89 *n*, 383, 400
 Lyapunov exponents and, 267
 momentum, 79–81
 Noether’s theorem, 90–91
 phase space reduction and, 224–226
- phase volume (*see* Phase-volume conservation)
Poisson brackets of, 221
symmetry and, 79, 90
for top, 160
Constant of motion (integral of motion), 78. *See also* Conserved quantities; Hénon–Heiles problem
Constraint(s)
 augmented Lagrangian and, 102, 109
 configuration space and, 4
 as coordinate transformations, 59–63
 explicit, 99–103
 in extended bodies, 4
 holonomic, 4 *n*, 109
 integrable, 4 *n*, 109
 linear in velocities, 112
 nonholonomic (non-integrable), 112
 on coordinates, 101
 rigid, 49–63
 as subsystem couplers, 105
 total time derivative and, 108
 velocity-dependent, 108
 velocity-independent, 101
Constraint force, 104
Constructors in Scheme, 503
Contact transformation. *See* Canonical transformations
Continuation procedure, 247
Continued-fraction
 approximation of irrational number, 325
Contraction of tuples, 514
coordinate, 15 *n*
Coordinate(s). *See also*
 Generalized coordinates
 action-angle (*see* Action-angle coordinates)
 conserved quantities and choice of, 79–81
 constraints on, 101
 cyclic, 80, 224 *n*
 heliocentric, 409 (ex. 5.21)

- Coordinate(s) (*continued*)
 ignorable (cyclic), 80
 Jacobi, 409 (ex. 5.21)
 polar (*see* Polar coordinates)
 redundant, and initial
 conditions, 69 *n*
 rotating (*see* Rotating
 coordinates)
 spherical, 84
- Coordinate function (χ), 7
- Coordinate-independence
 of action, 17
 of Lagrange equations, 30, 43
 (ex. 1.14)
 of variational formulation, 3, 39
- Coordinate path (q), 7. *See also*
 Local tuple
- Coordinate selector (Q), 220
- Coordinate singularity, 144
- Coordinate transformations,
 44–47
 constraints as, 59–63
- Coriolis force, 47, 49
- Correction fluid, 150
- Cotangent space, bundle, 203 *n*
- Coupling, spin-orbit. *See*
 Spin-orbit coupling
- Coupling systems, 105–106
- Curves, invariant. *See* Invariant
 curves
- Cyclic coordinate, 80, 224 *n*
- D.* *See* Derivative
- D** (Scheme procedure for
 derivative), 16 *n*, 516
- D-as-matrix**, 355 *n*
- D-phase-space**, 347
- ∂ . *See* Partial derivative
- D_t (total time derivative), 64
- d'Alembert–Lagrange principle
 (Jean leRond d'Alembert), 113
- Damped harmonic oscillator, 274
- define**, 499
- definite-integral**, 17
- Definite integral, 10 *n*
- Definitions in Scheme, 499–500
- Degrees of freedom, 4–5
- Delta function, 454 (ex. 6.12)
- Derivative, 8 *n*, 516–521. *See also*
 Total time derivative
 as operator, 517
 as Poisson bracket, 446
 chain rule, 517, 523 (ex. 9.1)
 in Scheme programs: **D**, 16 *n*,
 516
 notation: **D**, 8 *n*, 516
 of function of multiple
 arguments, 29 *n*, 518–521
 of function with structured
 arguments, 24 *n*
 of function with structured
 inputs and outputs, 522
 of state, 71
 partial (*see* Partial derivative)
 precedence of, 8 *n*, 516
 with respect to a tuple, 29 *n*
- determinant**, 144
- Differentiable manifold, 7 *n*
- Dimension of configuration space,
 4–5
- Dirac, Paul Adrien Maurice, 12 *n*
- Dissipation of energy
 in free-body rotation, 150
 tidal friction, 170
- Dissipative system, phase-volume
 conservation, 274
- Dissolution of invariant curves,
 329–330, 486
- Distribution functions, 276
- Divided phase space, 244, 258,
 286–290
- Dot notation, 32 *n*
- Double pendulum. *See*
 Pendulum, double
- down**, 15 *n*, 513
- Down tuples, 512
- Driven harmonic oscillator, 430
 (ex. 6.6)
- Driven pendulum. *See* Pendulum
 (driven)
- Driven rotor, 317, 321
- Dt** (total time derivative), 97
- Dynamical state. *See* State
- E** (Euler–Lagrange operator), 98
- \mathcal{E} (energy state function), 82

- Earth
 precession of, 176 (ex. 2.18)
 rotational alignment of, 151
- Effective Hamiltonian, 230
- Effects in Scheme, 505–508
- Eigenvalues and eigenvectors
 for equilibria, 293
 for fixed points, 296
 for Hamiltonian systems, 298
 of inertia tensor, 132
 for unstable fixed point, 303
- Einstein, Albert, 1
- Einstein summation convention,
 367 *n*
- else**, 500
- Empty list, 503
- Energy, 81
 as sum of kinetic and potential
 energies, 82
 conservation of, 81–83, 142, 211
 dissipation of (*see* Dissipation
 of energy)
- Energy state function (\mathcal{E}), 82
 Hamiltonian and, 200
- Epicyclic motion, 381–389
- eq?**, 505
- Equilibria, 222–223, 291–295. *See also* Fixed points
 for angular momentum, 149
 inverted, for pendulum, 246,
 282 (ex. 3.14), 491–494, 496
 (ex. 7.4)
 linear stability of, 291–295
 relative, 149
 stable and unstable, 287
- Equinox, precession of, 176 (ex.
 2.18)
- Ergodic motion, 312 *n*
- Ergodic theorem, 251
- Euler, Leonhard, 13 *n*
- Euler->M**, 139
- Euler-state->omega-body**, 140
- Euler angles, 137–141
 for axisymmetric top, 159
 kinetic energy in terms of, 141
 singularities and, 143, 154
- Euler–Lagrange equations. *See*
 Lagrange equations
- Euler-Lagrange-operator** (\mathcal{E}),
 98
- Euler–Lagrange operator (\mathcal{E}), 98
- Euler’s equations, 151–157
 singularities in, 154
- Euler’s theorem on homogeneous
 functions, 83 *n*
- Euler’s theorem on rotations, 123
 Euler angles and, 182
- Evolution. *See* Time evolution of
 state
- evolve**, 75, 145, 238
- explore-map**, 248
- Exponential(s)
 of differential operator, 443
 of Lie derivative, 447 (eq. 6.147)
 of noncommuting operators,
 451–453
- Exponential divergence, 241, 243,
 263–267. *See also* Chaotic
 motion; Lyapunov exponent
 homoclinic tangle and, 307
- Expressions in Scheme, 497
- Extended phase space, 394–402
 generating functions in, 407
- F₁–F₄**. *See also* Generating
 functions
 $F_1(t, q, q')$, 373
 $F_2(t, q, p')$, 373
 $F_3(t, p, q')$, 374
 $F_4(t, p, p')$, 374
- F->C**, 46, 96
- F->CH**, 339
- F->K**, 340
- Fermat, Pierre, 13 (ex. 1.3)
 Fermat’s principle (optics), 13
 (ex. 1.3), 13 *n*
- Fermi, Enrico, 251
- Feynman, Richard P., 12 *n*
- find-path**, 21
- First amendment. *See* Degrees of
 freedom
- First integral, 78
- Fixed points, 295. *See also*
 Equilibria
- elliptic, 299, 320

- Fixed points (*continued*)
 - equilibria or periodic motion and, 290, 295
 - for Hamiltonian systems, 298
 - hyperbolic, 299, 320
 - linear stability of, 295–297
 - manifolds for, 303
 - parabolic, 299
 - Poincaré–Birkhoff fixed points, 320
 - Poincaré–Birkhoff theorem, 316–321
 - rational rotation number and, 316
- Floating-point numbers in Scheme, 18 *n*
- Floquet multiplier, 296 *n*
- Flow, defined by vector field, 447 *n*
- Force
 - central (*see* Central force)
 - exerted by constraint, 104
- Forced libration of the Moon, 175
- Forced rigid body. *See* Rigid body, forced
- Formal parameters
 - of a function, 14 *n*
 - of a procedure, 499
- Foucault pendulum, 62 (ex. 1.25), 78 (ex. 1.31)
- frame**, 76 *n*
- Free libration of the Moon, 175
- Free particle
 - action, 14–20
 - Lagrange equations for, 33
 - Lagrangian for, 14–15
- Free rigid body. *See* Rigid body (free)
- Freudenthal, Hans, xiv *n*
- Friction
 - internal, 150
 - tidal, 170
- Function(s), 510–511
 - arithmetic operations on, 18 *n*, 511
 - composition of, 7 *n*, 510, 523 (ex. 9.2)
 - homogeneous, 83 *n*
- operator vs., 448 *n*, 517
- orthogonal, tuple-valued, 101 *n*
- parallel, tuple-valued, 101 *n*
- selector (*see* Selector function)
- tuple of, 7 *n*, 521
- vs. value when applied, 509, 510
- with multiple arguments, 518, 519, 523 (ex. 9.2)
- with structured arguments, 24 *n*, 519, 523 (ex. 9.2)
- with structured output, 521, 523 (ex. 9.2)
- Functional arguments, 10 *n*
- Functional mathematical notation, xiv, 509
- Function definition, 14 *n*
- Fundamental Poisson brackets, 352
- $\Gamma[q]$
 - for local tuple, 11
 - Lagrangian state path, 203
- Galaxy, 248–252
 - axisymmetric potential of, 250
- Galilean invariance, 68 (ex. 1.29), 341 (ex. 5.1)
- Gamma** (Scheme procedure for Γ), 16
 - optional argument, 36 (ex. 1.13)
- Gamma-bar**, 95
- Gas in corner of room, 273
- Generalized coordinates, 6–8, 39.
 See also Coordinate(s)
 - Euler angles as, 138 (*see also* Euler angles)
- Generalized momentum, 79
 - transformation of, 337 (eq. 5.5)
- Generalized velocity, 8
 - transformation of, 45
- Generating functions, 364–394
 - in extended phase space, 407
 - F_1 – F_4 , 373–374
 - F_1 , 364–368
 - F_2 , 371–373
 - F_2 and point transformations, 375–376
 - F_2 for polar coordinate transformation, 376–377

- F_2 for rotating coordinates, 377–378
 integral invariants and, 368–373
 Lagrangian action and, 421–425
 Legendre transformation
 between F_1 and F_2 , 373
 mixed-variable, 374
 Generic arithmetic, 16 *n*, 509
 Gibbs, Josiah Willard, 12 *n*, 203 *n*
 Golden number, 325
 Golden ratio, a most irrational number, 325
 Golden rotation number, 328
 Goldstein, Herbert, 119
 Goldstein’s hoop, 110
 Golf ball, tiny, 108 (ex. 1.41)
 Grand Old Duke of York. *See*
 neither up nor down
 Graphing, 23 (ex. 1.5), 75, 248
 Gravitational potential
 central, 31
 of galaxy, 250
 multipole expansion of, 165–169
 rigid-body, 166
 Group properties
 of canonical transformations, 346 (ex. 5.4)
 of rotations, 187 (*see also*
 Euler’s theorem on rotations)
- H* (Hamiltonian), 199
H-central, 339
H-harmonic, 448
H-pend-sysder, 237
 Hamilton, Sir William Rowan, 39 *n*, 183
 Hamiltonian, 199
 in action-angle coordinates, 311
 computing (*see H-...*)
 cyclic in coordinate, 224 *n*
 energy state function and, 200
 for axisymmetric potential, 250
 for central potential, 227, 339,
 381, 382
 for damped harmonic oscillator, 275
 for driven pendulum, 392
 for driven rotor, 317
 for harmonic oscillator, 344
 for harmonic oscillator, in
 action-angle coordinates, 346
 (eq. 5.31)
 for Kepler problem, 418
 for pendulum, 460
 for periodically driven
 pendulum, 236, 476
 for restricted three-body
 problem, 399, 400
 for spin-orbit coupling, 496 (ex.
 7.5)
 for top, 230
 for two-body problem, 378
 Hénon–Heiles, 252, 455 (ex.
 6.12)
 Lagrangian and, 200 (eq. 3.19), 210
 perturbation of action-angle, 316, 458
 time-dependent, and
 dissipation, 276
Hamiltonian->Lagrangian, 213
Hamiltonian->state-derivative, 204
 Hamiltonian flow, 447 *n*
 Hamiltonian formulation, 195
 Lagrangian formulation and, 217
 Hamiltonian state, 202–203
 Hamiltonian state derivative, 202, 204
 Hamiltonian state path $\Pi_L[q]$, 203
 Hamilton–Jacobi equation,
 411–413
 action-angle coordinates and, 413
 action at endpoints and, 425
 for harmonic oscillator, 413–417
 for Kepler problem, 417–421
 separation in spherical
 coordinates, 418–421
 time-independent, 413
Hamilton-equations, 203

- Hamilton's equations, 197–200
 in action-angle coordinates, 311
 computation of, 203–205
 dynamical, 217
 for central potential, 227
 for damped harmonic oscillator, 275
 for harmonic oscillator, 344
 from action principle, 215–217
 from Legendre transformation, 210–211
 numerical integration of, 236
 Poisson bracket form, 220
- Hamilton's principle, 38
 for systems with rigid constraints, 49–50
- Harmonic oscillator
 coupled, 105
 damped, 274
 decoupling via Lie transform, 442
 driven, 430 (ex. 6.6)
 first-order equations for, 72
 Hamiltonian for, 344
 Hamiltonian in action-angle coordinates, 346 (eq. 5.31)
 Hamilton's equations for, 344
 Lagrange equations for, 30, 72
 Lagrangian for, 21
 Lie series for, 448
 solution of, 34, 344
 solution via canonical transformation, 344
 solution via Hamilton–Jacobi, 413–417
- Hausdorff, Felix. *See* Baker–Campbell–Hausdorff formula
- Heiles, Carl, 241, 248. *See also* Hénon
- Heisenberg, Werner, 12 *n*, 203 *n*
- Heliocentric coordinates, 409 (ex. 5.21)
- Hénon, Michel, 195, 241, 248
- Hénon–Heiles problem, 248–263
 computing surfaces of section, 261–263
- Hamiltonian for, 252
 history of, 248–252
 integrals of motion, 251, 254, 256–260
 interpretation of model, 256–260
 model of, 252–254
 potential energy, 253
 surface of section, 254–263
- Hénon's quadratic map, 280 (ex. 3.13)
- Heteroclinic intersection, 305
- Higher-order perturbation theory, 468–473, 489–494
- History
 Hénon–Heiles problem, 248–252
 variational principles, 10 *n*, 13 *n*, 39 *n*
- Holonomic system, 4 *n*, 109
- Homoclinic intersection, 304
- Homoclinic tangle, 302–309
 chaotic regions and, 307
 computing, 307–309
 exponential divergence and, 307
- Homogeneous function, Euler's theorem, 83 *n*
- Huygens, Christiaan, 10 *n*
- Hyperion, chaotic tumbling of, 151, 170–176
- I* (identity operator), 517
- I* with subscript (selector), 64 *n*, 513
- if**, 501
- Ignorable coordinate. *See* Cyclic coordinate
- Indexing, zero-based. *See* Zero-based indexing
- Inertia, moments of. *See* Moment(s) of inertia
- Inertia matrix, 128. *See also* Inertia tensor
- Inertia tensor, 127
 diagonalization of, 132–133
 kinetic energy in terms of, 131
 principal axes of, 133
 transformation of, 130–132
- Initial conditions. *See* Sensitivity to initial conditions; State

- Inner product of tuples, 515
Instability. *See also* Equilibria;
 Linear stability
 free-body rotation, 149–151
Integers in Scheme, 18 *n*
Integrable constraints, 4 *n*, 109
Integrable systems, 285, 309–316
 periodic orbits of
 near-integrable systems, 316
 perturbation of, 316, 322, 457
 reduction to quadrature and,
 311 (*see also* Quadrature)
 surfaces of section for, 313–316
Integral, definite, 10 *n*
Integral invariant
 generating functions and,
 368–373
 Poincaré, 362–364
 Poincaré–Cartan, 402, 431–434
Integral of motion, 78. *See also*
 Conserved quantities;
 Hénon–Heiles problem
Integration. *See* Numerical
 integration
Invariant curves, 243, 322–330
 dissolution of, 329–330, 486
 finding (computing), 326–329
 finding (strategy), 322–325
 irrational rotation number and,
 322
 Kolmogorov–Arnold–Moser
 theorem, 322
Invariants of canonical
 transformations, 357–364. *See*
 also Integral invariants
Irrational number,
 continued-fraction
 approximation, 325
Islands in surfaces of section. *See*
 also Resonance
for Hénon–Heiles problem, 259
for periodically driven
 pendulum, 244–246, 289–290,
 483–486
for standard map, 279
perturbative vs. actual, 483–486
in Poincaré–Birkhoff
 construction, 321
Poisson series and, 488
secondary, 260, 290
size of, 322, 488
small denominators and, 322,
 488
iterated-map, 308 *n*
Iteration in Scheme, 502
 \tilde{J} (shuffle function), 350
J, \mathbf{J}_n (symplectic unit), 301, 355
J-func, 351
J-matrix, 353
Jac (Jacobian of map), 270
Jacobi, Carl Gustav Jacob, 39 *n*.
 See also Hamilton–Jacobi
 equation
Jacobian, 270
Jacobi constant, 89 *n*, 383, 400
Jacobi coordinates, 409 (ex. 5.21)
Jacobi identity
 for commutators, 451
 for Poisson brackets, 221
Jeans, Sir James, “theorem” of,
 251
Jupiter, 129 (ex. 2.4)
KAM theorem. *See*
 Kolmogorov–Arnold–Moser
 theorem
Kepler, Johannes. *See* Kepler...
Kepler problem, 31, 35 (ex. 1.11)
 in reduced phase space, 406
 reduction to, 378–381
 solution via Hamilton–Jacobi
 equation, 417–421
Kepler’s third law, 35 (ex. 1.11),
 173
Kinematics of rotation, 122–126
Kinetic energy
 ellipsoid of, 148
 in Lagrangian, 38–39
 as Lagrangian for free body,
 122, 141
 as Lagrangian for free particle,
 14
of axisymmetric top, 159
of free rigid body, 148–150
of rigid body, 120–122 (*see also*
 Rigid body, kinetic energy...)

- Fixed points (*continued*)
 rotational and translational, 122
 in spherical coordinates, 84
- Knuth, Donald E., 531
- Kolmogorov, A. N.. *See*
 Kolmogorov–Arnold–Moser theorem
- Kolmogorov–Arnold–Moser theorem, 302, 322
- L* (Lagrangian), 11
- L* (Lie derivative), 447
- L-axisymmetric-top**, 229
- L-body**, 137
- L-body-Euler**, 141
- L-central-polar**, 43, 47
- L-central-rectangular**, 41
- L-free-particle**, 14
- L-harmonic**, 22
- L-pend**, 52
- L-periodically-driven-pendulum**, 74
- L-rectangular**, 213
- L-space**, 137
- L-space-Euler**, 141
- L-uniform-acceleration**, 40, 61
- Lagrange, Joseph Louis, 13 *n*, 39 *n*
- Lagrange-equations**, 33
- Lagrange equations, 23–25
 at a moment, 97
 computing, 33–36
 coordinate-independence of, 30, 43 (ex. 1.14)
 derivation of, 25–30
 as first-order system, 72
 for central potential (polar), 43
 for central potential (rectangular), 41
 for damped harmonic oscillator, 275
 for driven pendulum, 52
 for free particle, 33
 for free rigid body, 141
 for gravitational potential, 32
 for harmonic oscillator, 30, 72
 for periodically driven pendulum, 74
- for spin-orbit coupling, 173
 from Newton's equations, 36–38, 54–58
 vs. Newton's equations, 39
 numerical integration of, 73
 off the beaten path, 97
 singularities in, 143
 traditional notation for, xiv, 24
 uniqueness of solution, 69
- Lagrange-interpolation-function**, 20 *n*
- Lagrange interpolation polynomial, 20
- Lagrange multiplier. *See*
 Lagrangian, augmented
- Lagrangian, 12
 adding total time derivatives to, 65
 augmented, 102, 109
 computing, 14–15 (*see also L-...*)
 coordinate transformations of, 44
 cyclic in coordinate, 80
 energy and, 12
 for axisymmetric top, 159
 for central potential (polar), 42–43, 227
 for central potential (rectangular), 41
 for central potential (spherical), 84
 for constant acceleration, 40
 for damped harmonic oscillator, 275
 for driven pendulum, 51, 66
 for free particle, 14–15
 for free rigid body, 122, 141
 for gravitational potential, 31
 for harmonic oscillator, 21
 for spin-orbit coupling, 173
 for systems with rigid constraints, 49
 generating functions and, 421–423
 Hamiltonian and, 200 (eq. 3.19), 210
 kinetic energy as, 14, 122, 141

- kinetic minus potential energy
as, 38–39 (*see also* Hamilton’s principle)
non-uniqueness of, 63–66
parameter names in, 14 *n*
rotational and translational, 141
symmetry of, 90
- Lagrangian-action**, 17
- Lagrangian->energy**, 82
- Lagrangian->Hamiltonian**, 213
- Lagrangian->state-derivative**,
71
- Lagrangian action, 12
- Lagrangian formulation, 195
Hamiltonian formulation and,
217
- Lagrangian reduction, 233–236
- Lagrangian state. *See* State tuple
- Lagrangian state derivative, 71
- Lagrangian state path $\Gamma[q]$, 203
- lambda**, 498
- Lambda calculus, 509
- Lambda expression, 498–499
- Lanczos, Cornelius, 335
- Least action, principle of. *See*
Principle of stationary action
- Legendre, Adrien Marie. *See*
Legendre...
- Legendre polynomials, 167
- Legendre-transform**, 212
- Legendre transformation,
205–212
active arguments in, 208
passive arguments in, 208–209
of quadratic functions, 211
- Leibniz, Gottfried, 10 *n*
- let**, 501
- let***, 502
- Libration of the Moon, 174, 175
- Lie, Sophus. *See* Lie...
- Lie-derivative**, 448, 448 *n*
- Lie derivative, 447 *n*
commutator for, 452 (ex. 6.10)
Lie transform and, 447 (eq.
6.147)
operator L_H , 447
- Lie series, 443–451
computing, 448–451
for central field, 450
- for harmonic oscillator, 448
in perturbation theory, 458–460
- Lie-transform**, 448
- Lie transforms, 437–443
advantage of, 441
composition of, 451
computing, 448
exponential identities, 451–453
for finding normal modes, 442
Lie derivative and, 447 (eq.
6.147)
in perturbation theory, 458
- Lindstedt, A., 471
- linear-interpolants**, 20 *n*
- Linear momentum, 80
- Linear separation of regular
trajectories, 263
- Linear stability, 290
equilibria and fixed points,
297–302
nonlinear stability and, 302
of equilibria, 291–295
of fixed points, 295–297
of inverted equilibrium of
pendulum, 492, 496 (ex. 7.4)
- Linear transformations
as tuples, 515
composition of, 516
- Liouville, Joseph. *See* Liouville...
- Liouville equation, 276
- Liouville’s theorem, 268–272
from canonical transformation,
428
- Lipschitz condition (Rudolf
Lipschitz), 69 *n*
- Lisp, 503 *n*
- list**, 503
- list-ref**, 503
- Lists in Scheme, 502–504
- literal-function**, 15, 512, 521
- Literal symbol in Scheme,
504–505
- Local names in Scheme, 501–502
- Local state tuple, 71
- Local tuple, 11
component names, 14 *n*
functions of, 14 *n*
in Scheme programs, 15 *n*
transformation of (C), 44

- Log, falling off, 84 (ex. 1.33)
- Loops in Scheme, 502
- Lorentz, Hendrik Antoon. *See* Lorentz transformations
- Lorentz transformations as point transformations, 399 (ex. 5.18)
- Lorenz, Edward, 241 *n*
- Lyapunov, Alexey M.. *See* Lyapunov exponent
- Lyapunov exponent, 267. *See also* Chaotic motion
- conserved quantities and, 267
- exponential divergence and, 267
- Hamiltonian constraints, 302
- linear stability and, 297
- M-of-q->omega-body-of-t**, 126
- M-of-q->omega-of-t**, 126
- M->omega**, 126
- M->omega-body**, 126, 185
- MacCullagh's formula, 168 *n*
- make-path**, 20, 20 *n*
- Manifold
- differentiable, 7 *n*
 - stable and unstable, 303–309
- Map
- area-preserving, 278
 - Chirikov–Taylor, 278 *n*
 - fixed points of, 295–297 (*see also* Fixed points)
 - Hénon's quadratic, 280 (ex. 3.13)
 - Poincaré, 242
 - representation in programs, 247
 - standard, 277–280
 - symplectic, 301
 - twist, 315
- Mars. *See* Phobos
- Mass point. *See* Point mass
- Mathematical notation. *See* Notation
- Mather, John N. (discoverer of sets named *cantori* by Ian Percival), 244 *n*
- Matrix
- inertia, 128 (*see also* Inertia tensor)
 - orthogonal, 124, 130 *n*
 - symplectic, 301, 355, 356 (ex. 5.6)
 - as tuple, 515
- Maupertuis, Pierre-Louis Moreau de, 13 *n*
- Mean motion, 175 *n*
- Mechanics, 1–496
- Newtonian vs. variational formulation, 3, 39
- Mercury, resonant rotation of, 171, 193 (ex. 2.21)
- Minimization
- of action, 18–23
 - in Scmutils, 19 *n*, 21 *n*
- minimize**, 19 *n*
- Mixed-variable generating functions, 374
- Moment(s) of inertia, 126–130
- about a line, 128
 - about a pivot point, 159
 - principal, 132–135
 - of top, 159
- Momentum. *See also* Angular momentum
- conjugate to coordinate (*see* Conjugate momentum)
 - conservation of, 79–81
 - generalized (*see* Generalized momentum)
 - variation of, 216 *n*
- momentum**, 204
- Momentum path, 80
- Momentum selector (*P*), 199, 220
- Momentum state function (\mathcal{P}), 79
- Moon
- head-shaking, 174
 - history of, 9 *n*
 - libration of, 174, 175
 - rotation of, 119, 151, 170–176, 496 (ex. 7.5)
- Moser, Jürgen. *See* Kolmogorov–Arnold–Moser theorem
- Motion
- atomic-scale, 8 *n*

- chaotic (*see* Chaotic motion)
constrained, 99–103 (*see also* Constraint(s))
dense, on torus, 312 *n*
deterministic, 9
epicyclic, 381–389
ergodic, 312 *n*
periodic (*see* Periodic motion)
quasiperiodic, 243, 312
realizable vs. conceivable, 2
regular vs. chaotic, 241 (*see also* Regular motion)
smoothness of, 8
tumbling (*see* Chaotic motion, of Hyperion; Rotation(s), (in)stability of)
- multidimensional-minimize**, 21, 21 *n*
- Multiplication of operators as composition, 517
- Multiplication of tuples, 514–516 as composition, 516 as contraction, 514
- Multiply periodic functions, Poisson series for, 474
- Multipole expansion of potential energy, 165–169
- n*-body problem, 408 (ex. 5.21). *See also* Three-body problem, restricted; Two-body problem
- Nelder–Mead minimization method, 21 *n*
- Newton, Sir Isaac, 3
- Newtonian formulation of mechanics, 3, 39
- Newton's equations as Lagrange equations, 36–38, 54–58
vs. Lagrange equations, 39
- Noether, Emmy, 81 *n*
- Noether's integral, 91
- Noether's theorem, 90–91 angular momentum and, 143
- Non-associativity and associativity of tuple multiplication, 515, 516
- Non-axisymmetric top, 263
- Non-commutativity. *See also* Commutativity exponential(s) of noncommuting operators, 451–453 of some partial derivatives, 427 *n*, 520 of some tuple multiplication, 516
- Nonholonomic system, 112
- Nonsingular structure, 368 *n*
- Notation, 509–523. *See also* Subscripts; Superscripts; Tuples { } for Poisson brackets, 218 () for up tuples, 512 [] for down tuples, 512 [] for functional arguments, 10 *n* ambiguous, xiv–xv composition of functions, 7 *n* definite integral, 10 *n* derivative, partial: ∂ , xv, 24, 520 derivative: D , 8 *n*, 516 functional, xiv, 509 functional arguments, 10 *n* function of local tuple, 14 *n* selector function: I with subscript, 64 *n*, 513 total time derivative: D_t , 64 traditional, xiv–xv, 24, 200 *n*, 218 *n*, 509
- Numbers in Scheme, 18 *n*
- Numerical integration of Hamilton's equations, 236 of Lagrange equations, 73 in Scmutils, 17 *n*, 74 *n*, 145 symplectic, 453 (ex. 6.12)
- Numerical minimization in Scmutils, 19 *n*, 21 *n*
- Nutation of top, 162 (fig. 2.5), 164 (ex. 2.15)
- Oblateness, 170
- omega** (symplectic 2-form), 361
- omega-cross**, 126

- Operator, 517
 arithmetic operations on, 34 *n*, 517
 composition of, 517
 exponential identities, 451–453
 function vs., 448 *n*, 517
 generic, 16 *n*
- Operators
 derivative (*D*) (*see* Derivative)
 Euler–Lagrange (*E*), 98
 Lie derivative (L_H), 447
 Lie transform ($E'_{\epsilon,W}$), 439
 partial derivative (∂) (*see*
 Partial derivative)
 variation (δ_η), 26
- Optical libration of the Moon, 174
- Optics
 Fermat, 13 (ex. 1.3)
 Snell’s law, 13 *n*
- Orbit. *See* Orbital motion;
 Phase-space trajectory
- Orbital elements, 421
- Orbital motion. *See also*
 Epicyclic motion; Kepler
 problem
 in a central potential, 78 (ex.
 1.30)
 Lagrange equations for, 31–32
 retrodiction of, 9 *n*
- Orientation. *See also* Rotation(s)
 Euler’s equations and, 153–154
 nonsingular coordinates for,
 181–191
 specified by Euler angles, 138
 specified by rotations, 123
- Orientation vector, 182
- Orthogonal matrix, 124, 130 *n*
- Orthogonal transformation. *See*
 Orthogonal matrix
- Orthogonal tuple-valued
 functions, 101 *n*
- Oscillator. *See* Harmonic
 oscillator
- osculating-path**, 96
- Osculation of paths, 94
- Ostrogradsky, M. V., 39 *n*
- Out-of-roundness parameter, 173
- P* (momentum selector), 199, 220
- P* (momentum state function), 79
- p->r* (polar-to-rectangular), 46
- pair?**, 504
- Pairs in Scheme, 503
- Parallel tuple-valued functions,
 101 *n*
- Parameters, formal. *See* Formal
 parameters
- Parametric path, 20
- parametric-path-action**, 21
 with graph, 23 (ex. 1.5)
- Parentheses
 in Scheme, 497, 498 *n*
 for up tuples, 512
- partial**, 33 *n*
- Partial derivative, 24, 518–519,
 520
 chain rule, 519, 523 (ex. 9.1)
 notation: ∂ , xv, 24, 520
- Particle, free. *See* Free particle
- Path, 2
 coordinate path (*q*), 7 (*see also*
 Local tuple)
 finding, 20–23
 momentum path, 80
 osculation of, 94
 parametric, 20
 realizable (*see* Realizable path)
 variation of, 12, 18, 26
- Path-distinguishing function, 2,
 8. *See also* Action
- Path functions, abstraction of, 94
- Peak, 222
- Pendulum. *See also* Pendulum
 (driven); Periodically driven
 pendulum
 behavior of, 223, 286–287
 constraints and, 103
 degrees of freedom of, 5 (ex.
 1.1)
 double (planar), 6, 117 (ex.
 1.44)
 double (spherical), 5 (ex. 1.1)
 equilibria, stable and unstable,
 287
- Foucault, 62 (ex. 1.25), 78 (ex.
 1.31)

- Hamiltonian for, 460
Lagrangian for, 32 (ex. 1.9)
periodically driven pendulum
vs., 244
as perturbed rotor, 460–473
phase plane of, 223, 286
phase-volume conservation for,
268
spherical, 5 (ex. 1.1), 86 (ex.
1.34)
width of oscillation region, 466
- Pendulum (driven), 50–52. *See*
also Pendulum; Periodically
driven pendulum
drive as modification of gravity,
66
Hamiltonian for, 392
Lagrange equations for, 52
Lagrangian for, 51, 66
- Pericenter, 171 *n*
- Period doubling, 245
- Periodically driven pendulum.
See also Pendulum (driven);
Pendulum
behavior of, 196, 244–246
chaotic behavior of, 76, 243
emergence of divided phase
space, 286–290
Hamiltonian for, 236, 476
inverted equilibrium, 246, 282
(ex. 3.14), 491–494, 496 (ex.
7.4)
islands in sections for, 244–246,
289–290, 483–486
Lagrange equations for, 74
linear stability analysis, 492,
496 (ex. 7.4)
as perturbed rotor, 476–478
phase-space descriptions for,
239
phase space evolution of, 236
resonances for, 481–491
spin-orbit coupling and, 173
surface of section for, 242–248,
282 (ex. 3.14), 287–290,
483–494
undriven pendulum vs., 244
- with zero-amplitude drive,
286–289
- Periodically driven systems,
surfaces of section, 241–248
- Periodic motion, 312
fixed points and, 295
integrable systems and, 309, 316
- Periodic points, 295
Poincaré–Birkhoff theorem,
316–321
- rational rotation number and,
316
- resonance islands and, 290
- Perturbation of action-angle
Hamiltonian, 316, 458
- Perturbation theory, 457
for many degrees of freedom,
473–478
- for pendulum, 466–468
- for periodically driven
pendulum, 491–494
- for spin-orbit coupling, 496 (ex.
7.5)
- higher-order, 468–473, 489–494
- Lie series in, 458–460
- nonlinear resonance, 478–494
- secular-term elimination,
471–473
- secular terms in, 470
- small denominators in, 475, 476
- Phase portrait, 231, 248 (ex.
3.10)
- Phase space, 203. *See also*
Surface of section
chaotic regions, 241
divided, 244, 258, 286–290
evolution in, 236–238 (*see also*
Time evolution of state)
extended, 394–402
non-uniqueness, 238–239
of pendulum, 223, 286
qualitative features, 242–246,
258–260, 285–286
reduced, 402–407
regular regions, 241
two-dimensional, 222
volume (*see* Phase-volume
conservation)

- Phase space reduction, 224–226
 conserved quantities and,
 224–226
 Lagrangian, 233–236
- Phase-space state function, 519
 in Scheme, 521
- Phase-space trajectory (orbit)
 asymptotic, 223, 287, 302
 chaotic, 243, 259
 periodic, 309, 312, 316
 quasiperiodic, 243, 312
 regular, 243, 258
 regular vs. chaotic, 241
- Phase-volume conservation, 268,
 428
 for damped harmonic oscillator,
 274
 for pendulum, 268
 under canonical
 transformations, 358–359
- Phobos, rotation of, 171
- Pit, 222
- Planets. *See also* Earth; Jupiter;
 Mercury
 moment of inertia of, 129 (ex.
 2.4)
 rotational alignment of, 151
 rotation of, 165
- plot-parametric-fill**, 308
- plot-point**, 76 *n*
- Plotting, 23 (ex. 1.5), 75, 248
- Poe, Edgar Allan. *See* Pit;
 Pendulum
- Poincaré, Henri, 239 *n*, 251, 285,
 302, 471
- Poincaré–Birkhoff theorem,
 316–321
 computing fixed points, 321–322
 recursive nature of, 321
- Poincaré–Cartan integral
 invariant, 402
 time evolution and, 431–434
- Poincaré integral invariant,
 362–364
 generating functions and,
 368–373
- Poincaré map, 242
- Poincaré recurrence, 272
- Poincaré section. *See* Surface of
 section
- Point mass, 4 *n*. *See also* Golf
 ball, tiny
- Point transformations, 336–341.
See also Canonical
 transformations
 computing, 339–341
 general canonical
 transformations vs., 357
 generating functions for,
 375–376
 polar-rectangular conversion,
 339, 376–377
 to rotating coordinates,
 348–349, 377–378
 time-independent, 338
- Poisson, Siméon Denis, 33 (ex.
 1.10)
- Poisson brackets, 218–222
 canonical condition and,
 352–353
 commutator and, 452 (ex. 6.10)
 of conserved quantities, 221
 as derivations, 446
 fundamental, 352
 Hamilton’s equations in terms
 of, 220
 in terms of \tilde{J} , 352
 in terms of symplectic 2-form,
 ω , 360
 invariance under canonical
 transformations, 358
 Jacobi identity for, 221
 Lie derivative and, 447
- Poisson series
 for multiply periodic function,
 474
 resonance islands and, 488
- polar-canonical**, 345
- Polar-canonical transformation,
 344
 generating function for, 365
 harmonic oscillator and, 346
- Polar coordinates
 Lagrangian in, 42–43
 point transformation to
 rectangular, 339, 376–377

- transformation to rectangular, 46
- Potential. *See* Central force; Gravitational potential
- Potential energy
- of axisymmetric top, 160
 - Hénon–Heiles, 253
 - in Lagrangian, 38–39
 - multipole expansion of, 165–169
- Precession
- of equinox, 176 (ex. 2.18)
 - of top, 119, 162 (fig. 2.6), 164 (ex. 2.16)
- Predicate in conditional, 500
- Predicting the past, 9 *n*
- principal-value**, 76 *n*
- Principal axes, 133
- of this dense book, 135 (ex. 2.7), 150
- Principal moments of inertia, 132–135
- kinetic energy in terms of, 134, 141, 148
- Principle of
- d'Alembert–Lagrange, 113
- Principle of least action. *See* Principle of stationary action
- Principle of stationary action (action principle), 8–13
- Hamilton's equations and, 215–217
- principle of least action, 10 *n*, 13 *n*, 39 *n*
- statement of, 12
 - used to find paths, 20
- print-expression**, 444, 511 *n*
- Probability density in phase space, 276
- Procedure calls, 497–498
- Procedures
- arithmetic operations on, 19 *n*
 - generic, 16 *n*
- Products of inertia, 128
- Q*** (coordinate selector), 220
- Q̇*** (velocity selector), 64
- q*** (coordinate path), 7
- qcrk4** (quality-controlled Runge–Kutta), 145
- Quadratic functions, Legendre transformation of, 211
- Quadrature, 161 *n*, 222. *See also* Integrable systems
- integrable systems and, 311
 - reduction to, 224 *n*
- Quartet, 300 (fig. 4.5)
- Quasiperiodic motion, 243, 312
- quaternion->angle-axis**, 184
- quaternion->RM**, 184
- quaternion->rotation-matrix**, 185
- quaternion-state->omega-body**, 186
- Quaternions, 181–191
- Hamilton's discovery of, 39 *n*
 - quaternion units, 188
- Quotation in Scheme, 504–505
- qw-state->L-space**, 190
- qw-sysder**, 189
- Radial momentum, 80
- Reaction force. *See* Constraint force
- Realizable path, 9
- conserved quantities and, 78
 - as solution of Hamilton's equations, 202
 - as solution of Lagrange equations, 23
- stationary action and, 9–13
- uniqueness, 12
- Recurrence theorem of Poincaré, 272
- Recursive procedures, 501
- Reduced mass, 35 (ex. 1.11), 380
- Reduced phase space, 402–407
- Reduction
- Lagrangian, 233–236
 - of phase space (*see* Phase space reduction)
 - to quadrature, 224 *n*
- ref**, 15 *n*, 514
- Regular motion, 241, 243, 258
- linear separation of trajectories, 263

- Renormalization, 267 *n*
- Resonance. *See also*
Commensurability
- center, 480
- islands (*see* Islands in surfaces of section)
- nonlinear, 478–494
- of Mercury’s rotation, 171, 193 (ex. 2.21)
- overlap criterion, 488–489, 496 (ex. 7.5)
- for periodically driven pendulum, 481–491
- spin-orbit, 177–181
- width, 483 (ex. 7.2), 488
- Restricted three-body problem.
See Three-body problem, restricted
- Rigid body, 120
- forced, 154–157 (*see also* Spin-orbit coupling; Top)
- free (*see* Rigid body (free))
- kinetic energy, 120–122
- kinetic energy in terms of inertia tensor and angular velocity, 126–129, 131
- kinetic energy in terms of principal moments and angular momentum, 148
- kinetic energy in terms of principal moments and angular velocity, 134
- kinetic energy in terms of principal moments and Euler angles, 141
- vector angular momentum, 135–137
- Rigid body (free), 141
- angular momentum, 151–153
- angular momentum and kinetic energy, 146–150
- computing motion of, 143–145
- Euler’s equations and, 151–154
- (in)stability, 149–151
- orientation, 153–154
- Rigid constraints, 49–63
- as coordinate transformations, 59–63
- Rotating coordinates
- in extended phase space, 400–402
- generating function for, 377–378
- point transformation for, 348–349, 377–378
- Rotation(s). *See also* Orientation
- active, 130
- composition of, 123, 187
- computing, 93
- group property of, 187
- (in)stability of, 149–151
- kinematics of, 122–126
- kinetic energy of (*see* Rigid body, kinetic energy...)
- Lie generator for, 440
- matrices for, 138
- of celestial objects, 151, 165, 170–171
- of Hyperion, 170–176
- of Mercury, 171, 193 (ex. 2.21)
- of Moon, 119, 151, 170–176, 496 (ex. 7.5)
- of Phobos, 171
- of top, book, and Moon, 119
- orientation as, 123
- orientation vector and, 182
- passive, 130
- as tuples, 515
- Rotation number, 315
- golden, 328
- irrational, and invariant curves, 322
- rational, and commensurability, 316
- rational, and fixed and periodic points, 316
- Rotor
- driven, 317, 321
- pendulum as perturbation of, 460–473
- periodically driven pendulum as perturbation of, 476–478
- Routh, Edward John
- Routhian, 234
- Routhian equations, 236 (ex. 3.9)

- Runge–Kutta integration
method, 74 *n*
`qcrk4`, 145
- `Rx`, 63 (ex. 1.25), 93 *n*
- `Rx-matrix`, 139
- `Ry`, 63 (ex. 1.25), 93 *n*
- `Rz`, 63 (ex. 1.25), 93 *n*
- `Rz-matrix`, 139
- S* (action), 10
Lagrangian, 12
- `s->m` (structure to matrix), 353
- `s->r` (spherical-to-rectangular), 85
- Saddle point, 222
- Salam, Abdus, 509
- Saturn. *See* Hyperion
- Scheme, xvi, 497–508, 509. *See also* Scmutils
for Gnu/Linux, where to get it, xvi
- Schrödinger, Erwin, 12 *n*, 203 *n*
- Scmutils, xvi, 509–523. *See also* Scheme
generic arithmetic, 16 *n*, 509
minimization, 19 *n*, 21 *n*
numerical integration, 17 *n*, 74 *n*, 145
operations on operators, 34 *n*
simplification of expressions, 511
where to get it, xvi
- Second law of thermodynamics, 274
- Section, surface of. *See* Surface of section
- Secular terms in perturbation theory, 470
elimination of, 471–473
- Selector function, 64 *n*, 513
coordinate selector (Q), 220
momentum selector (P), 199, 220
velocity selector (\dot{Q}), 64
- Selectors in Scheme, 503
- Semicolon in tuple, 31 *n*, 520
- Sensitivity to initial conditions, 241 *n*, 243, 263. *See also* Chaotic motion
- Separatrix, 147, 222. *See also* Asymptotic trajectories
chaos near, 290, 484, 486
motion near, 302
- series, 462
- series:for-each, 444
- series:sum, 463
- set-ode-integration-method!, 145
- show-expression, 16, 46 *n*
- Shuffle function \tilde{J} , 350
- Simplification of expressions, 511
- Singularities, 202 *n*
Euler angles and, 143, 154
in Euler’s equations, 154
quaternions, 181–191
- Sleeping top, 231
- Small denominators
for periodically driven pendulum, 477
in perturbation theory, 475, 476
resonance islands and, 322, 488
- Small divisors. *See* Small denominators
- Snell’s law, 13 *n*
- Solvable systems. *See* Integrable systems
- `solve-linear-left`, 71 *n*
- `solve-linear-right`, 339 *n*
- Spherical coordinates
kinetic energy in, 84
Lagrangian in, 84
- Spin-orbit coupling, 165–181
chaotic motion, 282 (ex. 3.15), 496 (ex. 7.5)
Hamiltonian for, 496 (ex. 7.5)
Lagrange equations for, 173
Lagrangian for, 173
periodically driven pendulum and, 173
perturbation theory for, 496 (ex. 7.5)
resonances, 177–181
surface of section for, 282 (ex. 3.15)
- Spivak, Michael, xiv *n*, 509
- Spring–mass system. *See* Harmonic oscillator

- square**, 21 *n*, 499
 for tuples, 40 *n*, 499 *n*
- Stability.** *See* Equilibria; Instability; Linear stability
- Stable manifold**, 303–309
 computing, 307–309
- standard-map**, 278
- Standard map**, 277–280
- Stars.** *See* Galaxy
- State**, 68–71
 evolution of (*see* Time evolution of state)
 Hamiltonian vs. Lagrangian, 202–203
 in terms of coordinates and momenta (Hamiltonian), 196
- in terms of coordinates and velocities (Lagrangian), 69
- state-advancer**, 74
- State derivative**
 Hamiltonian, 204
 Hamiltonian vs. Lagrangian, 202
 Lagrangian, 71
- State path**
 Hamiltonian, 203
 Lagrangian, 203
- State tuple**, 71
- Stationarity condition**, 28
- Stationary action.** *See* Principle of stationary action
- Stationary point**, 2 *n*
- Steiner's theorem**, 129 (ex. 2.2)
- String theory**, 119 *n*, 150. *See also* Quartet
- Stroboscopic surface of section**, 241–248. *See also* Surface of section
 computing, 246
- Subscripts**
 down and, 15 *n*
 for down-tuple components, 513
 for momentum components, 79 *n*, 338 *n*
 for selectors, 513
- Summation convention**, 367 *n*
- Superscripts**
 for coordinate components, 7 *n*, 15 *n*, 79 *n*
- for up-tuple components, 513
- for velocity components, 15 *n*, 338 *n*
 up and, 15 *n*
- Surface of section**, 239–248
 in action-angle coordinates, 313
 area preservation of, 272, 434–435
 computing (Hénon–Heiles), 261–263
 computing (stroboscopic), 246
 fixed points (*see* Fixed points)
 for autonomous systems, 248–263
 for Hénon–Heiles problem, 254–263
 for integrable system, 313–316
 for non-axisymmetric top, 263
 for periodically driven pendulum, 242–248, 282 (ex. 3.14), 287–290, 483–494
 for restricted three-body problem, 283 (ex. 3.16)
 for spin-orbit coupling, 282 (ex. 3.15)
 for standard map, 277–280
 invariant curves (*see* Invariant curves)
 islands (*see* Islands in surfaces of section)
 stroboscopic, 241–248
- Symbolic values**, 511–512
- Symbols in Scheme**, 504–505
- Symmetry**
 conserved quantities and, 79, 90
 continuous, 195
 of Lagrangian, 90
 of top, 228
- symplectic-matrix?**, 355
- symplectic-transform?**, 355
- symplectic-unit**, 355
- Symplectic bilinear form** (2-form), 359–362
- invariance under canonical transformations, 359
- Symplectic condition.** *See* Symplectic transformations
- Symplectic integration**, 453 (ex. 6.12)

- Symplectic map, 301
Symplectic matrix, 301, 356 (ex. 5.6), 353–357
Symplectic transformations, 355.
See also Canonical transformations
antisymmetric bilinear form and, 359–362
Symplectic unit \mathbf{J} , \mathbf{J}_n , 301, 355
Syntactic sugar, 499
System derivative. *See* State derivative
- T-body**, 134
T-body-Euler, 141
T-func, 347
Taylor, J. B., 278 *n*
Tensor. *See* Inertia tensor
Tensor arithmetic
 notation and, 79 *n*, 338 *n*
 summation convention, 367 *n*
 tuple arithmetic vs., 509, 513
Theology and principle of least action, 13 *n*
Thermodynamics, second law, 274
Three-body problem, restricted, 86–90, 283 (ex. 3.16), 399–402
chaotic motion, 283 (ex. 3.16)
surface of section for, 283 (ex. 3.16)
Tidal friction, 170
time, 15 *n*
Time-dependent transformations, 347–349
Time evolution of state, 68–78
 action and, 423–425, 435–437
 as canonical transformation, 426–437
 in phase space, 236–238
Poincaré–Cartan integral
 invariant and, 431–434
Time-independence. *See also* Extended phase space
 energy conservation and, 81
Top
 axisymmetric (*see* Axisymmetric top)
 non-axisymmetric, 263
Top banana. *See* Non-axisymmetric top
Torque, 165 (ex. 2.16)
 in Euler's equations, 154
 in spin-orbit coupling, 173
Total time derivative, 63–65
 adding to Lagrangians, 65
 affecting conjugate momentum, 239
canonical transformation and, 390–393
commutativity of, 91 *n*
computing, 97
constraints and, 108
identifying, 68 (ex. 1.28)
notation: D_t , 64
properties, 67
Trajectory. *See* Path; Phase-space trajectory
Transformation
 canonical (*see* Canonical transformations)
 coordinate (*see* Coordinate transformations)
 Legendre (*see* Legendre transformation)
 Lie (*see* Lie transforms)
 orthogonal (*see* Orthogonal matrix)
 point (*see* Point transformations)
 symplectic (*see* Symplectic transformations)
 time-dependent, 347–349
Transpose, 351 *n*
transpose, 126, 351 *n*
True anomaly, 171 *n*
Tumbling. *See* Chaotic motion, of Hyperion; Rotation(s), (in)stability of
Tuples, 512–516
 arithmetic on, 509, 513–516
 commas and semicolons in, 520
component selector: I with subscript, 64 *n*, 513
composition and, 514
contraction, 514
of coordinates, 7
down and up, 512

- Tuples (*continued*)
 - of functions, 7 *n*, 521
 - inner product, 515
 - linear transformations as, 515
 - local (*see* Local tuple)
 - matrices as, 515
 - multiplication of, 514–516
 - rotations as, 515
 - semicolons and commas in, 520
 - squaring, 499 *n*, 514
 - state tuple, 71
 - up and down, 512
- Twist map, 315
- Two-body problem, 378–381
- Two-trajectory method, 265
- Undriven pendulum. *See* Pendulum
- Uniform circle map, 326
- Uniqueness
 - of Lagrangian—not!, 63–66
 - of phase-space description—not!, 238–239
 - of realizable path, 12
 - of solution to Lagrange equations, 69
- unstable-manifold**, 308
- Unstable manifold, 303–309
 - computing, 307–309
- up, 15 *n*, 513
- Up tuples, 512
- Vakonomic mechanics, 114 *n*
- Variation
 - chain rule, 27 (eq. 1.26)
 - of action, 28
 - of a function, 26
 - of a path, 12, 18, 26
 - operator: δ_η , 26
- Variational equations, 266
- Variational formulation of mechanics, 2–3, 39
- Variational principle. *See* Principle of stationary action
- Vector
 - body components of, 134
 - in Scheme, 502–504
 - square of, 18 *n*, 21 *n*
 - vector**, 504
 - vector?**, 504
 - vector-ref**, 504
 - Vector angular momentum, 135–137. *See also* Angular momentum
 - center-of-mass decomposition, 135
 - in terms of angular velocity and inertia tensor, 136
 - in terms of principal moments and Euler angles, 141
- Vector space of tuples, 514
- Vector torque. *See* Torque
- Velocity. *See* Angular velocity vector; Generalized velocity
- velocity**, 15 *n*
- Velocity dispersion in galaxy, 248
- Velocity selector (\dot{Q}), 64
- Web site for this book, xvi
- Wheel, 156 (ex. 2.13)
- Whittaker transform (Sir Edmund Whittaker), 357 (ex. 5.9)
- Width of oscillation region, 466 *n*
- write-line**, 505 *n*
- Zero-amplitude drive for pendulum, 286–289
- Zero-based indexing, 7 *n*, 15 *n*, 503, 513, 514, 531

THE ART OF COMPUTER PROGRAMMING

FASCICLE 1

MMIX

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY



Internet page <http://www-cs-faculty.stanford.edu/~knuth/taocp.html> contains current information about this book and related books.

See also <http://www-cs-faculty.stanford.edu/~knuth/mmix.html> for downloadable software, and <http://mmixmasters.sourceforge.net> for general news about MMIX.

Copyright © 1999 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher, except that the official electronic file may be used to print single copies for personal (not commercial) use.

Zeroth printing (revision 15), 15 February 2004

PREFACE

fas·ci·cle \'fasəkəl\ n . . . 1: a small bundle . . . an inflorescence consisting of a compacted cyme less capitate than a glomerule . . . 2: one of the divisions of a book published in parts
— P. B. GOVE, *Webster's Third New International Dictionary* (1961)

THIS IS THE FIRST of a series of updates that I plan to make available at regular intervals as I continue working toward the ultimate editions of *The Art of Computer Programming*.

I was inspired to prepare fascicles like this by the example of Charles Dickens, who issued his novels in serial form; he published a dozen installments of *Oliver Twist* before having any idea what would become of Bill Sikes! I was thinking also of James Murray, who began to publish 350-page portions of the Oxford English Dictionary in 1884, finishing the letter B in 1888 and the letter C in 1895. (Murray died in 1915 while working on the letter T; my task is, fortunately, much simpler than his.)

Unlike Dickens and Murray, I have computers to help me edit the material, so that I can easily make changes before putting everything together in its final form. Although I'm trying my best to write comprehensive accounts that need no further revision, I know that every page brings me hundreds of opportunities to make mistakes and to miss important ideas. My files are bursting with notes about beautiful algorithms that have been discovered, but computer science has grown to the point where I cannot hope to be an authority on all the material I wish to cover. Therefore I need extensive feedback from readers before I can finalize the official volumes.

In other words, I think these fascicles will contain a lot of Good Stuff, and I'm excited about the opportunity to present everything I write to whoever wants to read it, but I also expect that beta-testers like you can help me make it Way Better. As usual, I will gratefully pay a reward of \$2.56 to the first person who reports anything that is technically, historically, typographically, or politically incorrect.

Charles Dickens usually published his work once a month, sometimes once a week; James Murray tended to finish a 350-page installment about once every 18 months. My goal, God willing, is to produce two 128-page fascicles per year. Most of the fascicles will represent new material destined for Volumes 4 and higher; but sometimes I will be presenting amendments to one or more of the earlier volumes. For example, Volume 4 will need to refer to topics that belong in Volume 3, but weren't invented when Volume 3 first came out. With luck, the entire work will make sense eventually.

Fascicle Number One is about MMIX, the long-promised replacement for MIX. Thirty years have passed since the MIX computer was designed, and computer architecture has been converging during those years towards a rather different style of machine. Therefore I decided in 1990 to replace MIX with a new computer that would contain even less saturated fat than its predecessor.

Exercise 1.3.1–25 in the first three editions of Volume 1 spoke of an extended MIX called MixMaster, which was upward compatible with the old version. But MixMaster itself has long been hopelessly obsolete. It allowed for several gigabytes of memory, but one couldn't even use it with ASCII code to print lowercase letters. And ouch, its standard subroutine calling convention was irrevocably based on self-modifying instructions! Decimal arithmetic and self-modifying code were popular in 1962, but they sure have disappeared quickly as machines have gotten bigger and faster. Fortunately the new RISC machines have a very appealing structure, so I've had a chance to design a new computer that is not only up to date but also fun.

Many readers are no doubt thinking, “Why does Knuth replace MIX by another machine instead of just sticking to a high-level programming language? Hardly anybody uses assemblers these days.” Such people are entitled to their opinions, and they need not bother reading the machine-language parts of my books. But the reasons for machine language that I gave in the preface to Volume 1, written in the early 1960s, remain valid today:

- One of the principal goals of my books is to show how high-level constructions are actually implemented in machines, not simply to show how they are applied. I explain coroutine linkage, tree structures, random number generation, high-precision arithmetic, radix conversion, packing of data, combinatorial searching, recursion, etc., from the ground up.
- The programs needed in my books are generally so short that their main points can be grasped easily.
- People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird.
- Machine language is necessary in any case, as output of some of the software that I describe.
- Expressing basic methods like algorithms for sorting and searching in machine language makes it possible to carry out meaningful studies of the effects of cache and RAM size and other hardware characteristics (memory speed, pipelining, multiple issue, lookaside buffers, the size of cache blocks, etc.) when comparing different schemes.

Moreover, if I did use a high-level language, what language should it be? In the 1960s I would probably have chosen Algol W; in the 1970s, I would then have had to rewrite my books using Pascal; in the 1980s, I would surely have changed everything to C; in the 1990s, I would have had to switch to C++ and then probably to Java. In the 2000s, yet another language will no doubt be *de*

rigueur. I cannot afford the time to rewrite my books as languages go in and out of fashion; languages aren't the point of my books, the point is rather what you can do in your favorite language. My books focus on timeless truths.

Therefore I will continue to use English as the high-level language in *The Art of Computer Programming*, and I will continue to use a low-level language to indicate how machines actually compute. Readers who only want to see algorithms that are already packaged in a plug-in way, using a trendy language, should buy other people's books.

The good news is that programming for MMIX is pleasant and simple. This fascicle presents

- 1) a programmer's introduction to the machine (replacing Section 1.3.1 of Volume 1);
- 2) the MMIX assembly language (replacing Section 1.3.2);
- 3) new material on subroutines, coroutines, and interpretive routines (replacing Sections 1.4.1, 1.4.2, and 1.4.3).

Of course, MIX appears in many places throughout Volumes 1–3, and dozens of programs need to be rewritten for MMIX. Readers who would like to help with this conversion process are encouraged to join the MMIXmasters, a happy group of volunteers based at mmixmasters.sourceforge.net.

I am extremely grateful to all the people who helped me with the design of MMIX. In particular, John Hennessy and Richard L. Sites deserve special thanks for their active participation and substantial contributions. Thanks also to Vladimir Ivanović for volunteering to be the MMIX grandmaster/webmaster.

*Stanford, California
May 1999*

D. E. K.

*You can, if you want, rewrite forever.
— NEIL SIMON, *Rewrites: A Memoir* (1996)*

CONTENTS

Chapter 1 — Basic Concepts	1
1.3'. MMIX	2
1.3.1'. Description of MMIX	2
1.3.2'. The MMIX Assembly Language	28
1.4'. Some Fundamental Programming Techniques	52
1.4.1'. Subroutines	52
1.4.2'. Coroutines	66
1.4.3'. Interpretive Routines	73
Answers to Exercises	94
Index and Glossary	127

1.3'. MMIX

IN MANY PLACES throughout this book we will have occasion to refer to a computer's internal machine language. The machine we use is a mythical computer called "MMIX." MMIX—pronounced *EM-micks*—is very much like nearly every general-purpose computer designed since 1985, except that it is, perhaps, nicer. The language of MMIX is powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned.

The reader is urged to study this section carefully, since MMIX language appears in so many parts of this book. There should be no hesitation about learning a machine language; indeed, the author once found it not uncommon to be writing programs in a half dozen different machine languages during the same week! Everyone with more than a casual interest in computers will probably get to know at least one machine language sooner or later. Machine language helps programmers understand what really goes on inside their computers. And once one machine language has been learned, the characteristics of another are easy to assimilate. Computer science is largely concerned with an understanding of how low-level details make it possible to achieve high-level goals.

Software for running MMIX programs on almost any real computer can be downloaded from the website for this book (see page ii). The complete source code for the author's MMIX routines appears in the book *MMIXware* [*Lecture Notes in Computer Science 1750* (1999)]; that book will be called "the *MMIXware* document" in the following pages.

1.3.1'. Description of MMIX

MMIX is a polyunsaturated, 100% natural computer. Like most machines, it has an identifying number—the 2009. This number was found by taking 14 actual computers very similar to MMIX and on which MMIX could easily be simulated, then averaging their numbers with equal weight:

$$\begin{aligned}
 & (\text{Cray I} + \text{IBM 801} + \text{RISC II} + \text{Clipper C300} + \text{AMD 29K} + \text{Motorola 88K} \\
 & \quad + \text{IBM 601} + \text{Intel i960} + \text{Alpha 21164} + \text{POWER 2} + \text{MIPS R4000} \\
 & \quad + \text{Hitachi SuperH4} + \text{StrongARM 110} + \text{Sparc 64}) / 14 \\
 & = 28126 / 14 = 2009.
 \end{aligned} \tag{1}$$

The same number may also be obtained in a simpler way by taking Roman numerals.

Bits and bytes. MMIX works with patterns of 0s and 1s, commonly called binary digits or *bits*, and it usually deals with 64 bits at a time. For example, the 64-bit quantity

$$100111100011011101111001101110010111111010010100111110000010110 \tag{2}$$

is a typical pattern that the machine might encounter. Long patterns like this can be expressed more conveniently if we group the bits four at a time and use

hexadecimal digits to represent each group. The sixteen hexadecimal digits are

$$\begin{array}{llll} 0 = 0000, & 4 = 0100, & 8 = 1000, & c = 1100, \\ 1 = 0001, & 5 = 0101, & 9 = 1001, & d = 1101, \\ 2 = 0010, & 6 = 0110, & a = 1010, & e = 1110, \\ 3 = 0011, & 7 = 0111, & b = 1011, & f = 1111. \end{array} \quad (3)$$

We shall always use a distinctive typeface for hexadecimal digits, as shown here, so that they won't be confused with the decimal digits 0–9; and we will usually also put the symbol # just before a hexadecimal number, to make the distinction even clearer. For example, (2) becomes

$$\#9e3779b97f4a7c16 \quad (4)$$

in hexadecimalese. Uppercase digits ABCDEF are often used instead of abcdef, because #9E3779B97F4A7C16 looks better than #9e3779b97f4a7c16 in some contexts; there is no difference in meaning.

A sequence of eight bits, or two hexadecimal digits, is commonly called a *byte*. Most computers now consider bytes to be their basic, individually addressable units of information; we will see that an MMIX program can refer to as many as 2^{64} bytes, each with its own address from #0000000000000000 to #ffffffffffffffffff. Letters, digits, and punctuation marks of languages like English are often represented with one byte per character, using the American Standard Code for Information Interchange (ASCII). For example, the ASCII equivalent of MMIX is #4d4d4958. ASCII is actually a 7-bit code with control characters #00–#1f, printing characters #20–#7e, and a “delete” character #7f [see CACM 8 (1965), 207–214; 11 (1968), 849–852; 12 (1969), 166–178]. It was extended during the 1980s to an international standard 8-bit code known as Latin-1 or ISO 8859-1, thereby encoding accented letters: *pâté* is #70e274e9.

“Of the 256th squadron?”
 “Of the fighting 256th Squadron,” Yossarian replied.
 . . . “That’s two to the fighting eighth power.”
 — JOSEPH HELLER, *Catch-22* (1961)

A 16-bit code that supports nearly *every* modern language became an international standard during the 1990s. This code, known as Unicode or ISO/IEC 10646 UCS-2, includes not only Greek letters like Σ and σ (#03a3 and #03c3), Cyrillic letters like ІІ and ѿ (#0429 and #0449), Armenian letters like Ը and ՚ (#0547 and #0577), Hebrew letters like ו (#05e9), Arabic letters like ڻ (#0634), and Indian letters like ް (#0936) or ޱ (#09b6) or ޲ (#0b36) or ޴ (#0bb7), etc., but also tens of thousands of East Asian ideographs such as the Chinese character for mathematics and computing, 算 (#7b97). It even has special codes for Roman numerals: MMIX = #216f 216f 2160 2169. Ordinary ASCII or Latin-1 characters are represented by simply giving them a leading byte of zero: *pâté* is #0070 00e2 0074 00e9, à l’Unicode.

We will use the convenient term *wyde* to describe a 16-bit quantity like the wide characters of Unicode, because two-byte quantities are quite important in practice. We also need convenient names for four-byte and eight-byte quantities, which we shall call *tetrabytes* (or “*tetras*”) and *octabytes* (or “*octas*”). Thus

$$\begin{aligned} 2 \text{ bytes} &= 1 \text{ wyde}; \\ 2 \text{ wydes} &= 1 \text{ tetra}; \\ 2 \text{ tetras} &= 1 \text{ octa}. \end{aligned}$$

One octabyte equals four wydes equals eight bytes equals sixty-four bits.

Bytes and multibyte quantities can, of course, represent numbers as well as alphabetic characters. Using the binary number system,

- an unsigned byte can express the numbers 0 .. 255;
- an unsigned wyde can express the numbers 0 .. 65,535;
- an unsigned tetra can express the numbers 0 .. 4,294,967,295;
- an unsigned octa can express the numbers 0 .. 18,446,744,073,709,551,615.

Integers are also commonly represented by using *two’s complement notation*, in which the leftmost bit indicates the sign: If the leading bit is 1, we subtract 2^n to get the integer corresponding to an n -bit number in this notation. For example, -1 is the signed byte `#ff`; it is also the signed wyde `#ffff`, the signed tetrabyte `#ffffffffffff`, and the signed octabyte `#ffffffffffff`. In this way

- a signed byte can express the numbers $-128 \dots 127$;
- a signed wyde can express the numbers $-32,768 \dots 32,767$;
- a signed tetra can express the numbers $-2,147,483,648 \dots 2,147,483,647$;
- a signed octa can express the numbers $-9,223,372,036,854,775,808 \dots 9,223,372,036,854,775,807$.

Memory and registers. From a programmer’s standpoint, an MMIX computer has 2^{64} cells of *memory* and 2^8 general-purpose *registers*, together with 2^5 special registers (see Fig. 13). Data is transferred from the memory to the registers, transformed in the registers, and transferred from the registers to the memory. The cells of memory are called $M[0], M[1], \dots, M[2^{64}-1]$; thus if x is any octabyte, $M[x]$ is a byte of memory. The general-purpose registers are called $\$0, \$1, \dots, \$255$; thus if x is any byte, $\$x$ is an octabyte.

The 2^{64} bytes of memory are grouped into 2^{63} wydes, $M_2[0] = M_2[1] = M[0]M[1], M_2[2] = M_2[3] = M[2]M[3], \dots$; each wyde consists of two consecutive bytes $M[2k]M[2k+1] = M[2k] \times 2^8 + M[2k+1]$, and is denoted either by $M_2[2k]$ or by $M_2[2k+1]$. Similarly there are 2^{62} tetrabytes

$$M_4[4k] = M_4[4k+1] = \dots = M_4[4k+3] = M[4k]M[4k+1] \dots M[4k+3],$$

and 2^{61} octabytes

$$M_8[8k] = M_8[8k+1] = \dots = M_8[8k+7] = M[8k]M[8k+1] \dots M[8k+7].$$

In general if x is any octabyte, the notations $M_2[x]$, $M_4[x]$, and $M_8[x]$ denote the wyde, the tetra, and the octa that contain byte $M[x]$; we ignore the least

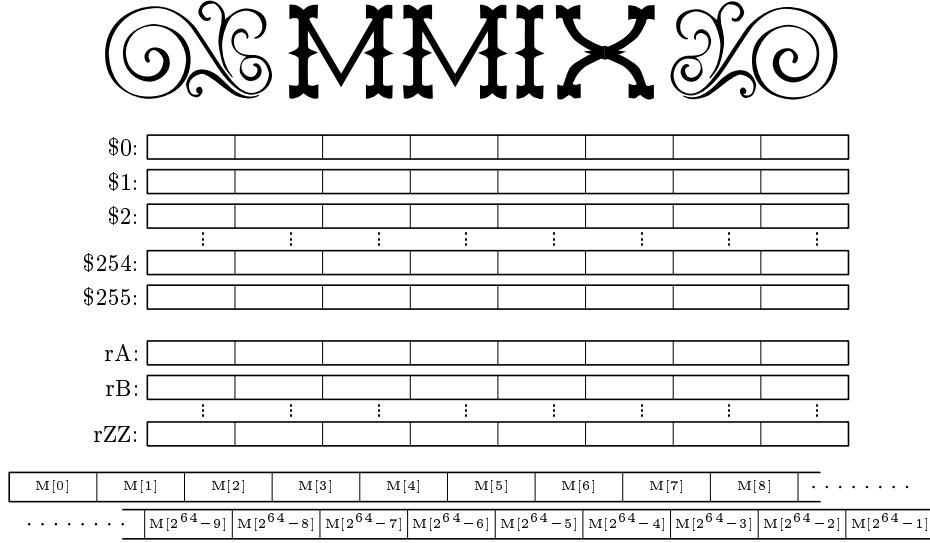


Fig. 13. The MMIX computer, as seen by a programmer, has 256 general-purpose registers and 32 special-purpose registers, together with 2^{64} bytes of virtual memory. Each register holds 64 bits of data.

significant $\lg t$ bits of x when referring to $M_t[x]$. For completeness, we also write $M_1[x] = M[x]$, and we define $M[x] = M[x \bmod 2^{64}]$ when $x < 0$ or $x \geq 2^{64}$.

The 32 special registers of MMIX are called rA, rB, ..., rZ, rBB, rTT, rWW, rXX, rYY, and rZZ. Like their general-purpose cousins, they each hold an octabyte. Their uses will be explained later; for example, we will see that rA controls arithmetic interrupts while rR holds the remainder after division.

Instructions. MMIX's memory contains instructions as well as data. An *instruction* or “command” is a tetrabyte whose four bytes are conventionally called OP, X, Y, and Z. OP is the *operation code* (or “opcode,” for short); X, Y, and Z specify the *operands*. For example, #20010203 is an instruction with OP = #20, X = #01, Y = #02, and Z = #03, and it means “Set \$1 to the sum of \$2 and \$3.” The operand bytes are always regarded as unsigned integers.

Each of the 256 possible opcodes has a symbolic form that is easy to remember. For example, opcode #20 is ADD. We will deal almost exclusively with symbolic opcodes; the numeric equivalents can be found, if needed, in Table 1 below, and also in the endpapers of this book.

The X, Y, and Z bytes also have symbolic representations, consistent with the assembly language that we will discuss in Section 1.3.2'. For example, the instruction #20010203 is conventionally written ‘ADD \$1,\$2,\$3’, and the addition instruction in general is written ‘ADD \$X,\$Y,\$Z’. Most instructions have three operands, but some of them have only two, and a few have only one. When there are two operands, the first is X and the second is the two-byte quantity YZ; the symbolic notation then has only one comma. For example, the instruction

'INCL \$X,YZ' increases register \$X by the amount YZ. When there is only one operand, it is the unsigned three-byte number XYZ, and the symbolic notation has no comma at all. For example, we will see that 'JMP @+4*XYZ' tells MMIX to find its next instruction by skipping ahead XYZ tetrabytes; the instruction 'JMP @+1000000' has the hexadecimal form #f003d090, because JMP = #f0 and 250000 = #03d090.

We will describe each MMIX instruction both informally and formally. For example, the informal meaning of 'ADD \$X,\$Y,\$Z' is "Set \$X to the sum of \$Y and \$Z"; the formal definition is $s(\$X) \leftarrow s(\$Y) + s(\$Z)$. Here $s(x)$ denotes the *signed integer* corresponding to the bit pattern x , according to the conventions of two's complement notation. An assignment like $s(x) \leftarrow N$ means that x is to be set to the bit pattern for which $s(x) = N$. (Such an assignment causes *integer overflow* if N is too large or too small to fit in x . For example, an ADD will overflow if $s(\$Y) + s(\$Z)$ is less than -2^{63} or greater than $2^{63} - 1$. When we're discussing an instruction informally, we will often gloss over the possibility of overflow; the formal definition, however, will make everything precise. In general the assignment $s(x) \leftarrow N$ sets x to the binary representation of $N \bmod 2^n$, where n is the number of bits in x , and it signals overflow if $N < -2^{n-1}$ or $N \geq 2^{n-1}$; see exercise 5.)

Loading and storing. Although MMIX has 256 different opcodes, we will see that they fall into a few easily learned categories. Let's start with the instructions that transfer information between the registers and the memory.

Each of the following instructions has a *memory address* A obtained by adding \$Y to \$Z. Formally,

$$A = (u(\$Y) + u(\$Z)) \bmod 2^{64} \quad (5)$$

is the sum of the *unsigned integers* represented by \$Y and \$Z, reduced to a 64-bit number by ignoring any carry that occurs at the left when those two integers are added. In this formula the notation $u(x)$ is analogous to $s(x)$, but it considers x to be an unsigned binary number.

- LDB \$X,\$Y,\$Z (load byte): $s(\$X) \leftarrow s(M_1[A])$.
- LDW \$X,\$Y,\$Z (load wyde): $s(\$X) \leftarrow s(M_2[A])$.
- LDT \$X,\$Y,\$Z (load tetra): $s(\$X) \leftarrow s(M_4[A])$.
- LDO \$X,\$Y,\$Z (load octa): $s(\$X) \leftarrow s(M_8[A])$.

These instructions bring data from memory into register \$X, changing the data if necessary from a signed byte, wyde, or tetrabyte to a signed octabyte of the same value. For example, suppose the octabyte $M_8[1002] = M_8[1000]$ is

$$M[1000]M[1001]\dots M[1007] = \#01\ 23\ 45\ 67\ 89\ ab\ cd\ ef. \quad (6)$$

Then if \$2 = 1000 and \$3 = 2, we have A = 1002, and

```
LDB $1,$2,$3 sets $1 ← #0000000000000045;
LDW $1,$2,$3 sets $1 ← #0000000000004567;
LDT $1,$2,$3 sets $1 ← #0000000001234567;
LDO $1,$2,$3 sets $1 ← #0123456789abcd.
```

But if $\$3 = 5$, so that $A = 1005$,

```
LDB $1,$2,$3 sets $1 ← #ffffffffff89ab;
LDW $1,$2,$3 sets $1 ← #ffffffffff89ab;
LDT $1,$2,$3 sets $1 ← #ffffffffff89ab cdef;
LDO $1,$2,$3 sets $1 ← #0123456789ab cdef.
```

When a signed byte or wyde or tetra is converted to a signed octa, its sign bit is “extended” into all positions to the left.

- LDBU \$X,\$Y,\$Z (load byte unsigned): $u(\$X) \leftarrow u(M_1[A])$.
- LDWU \$X,\$Y,\$Z (load wyde unsigned): $u(\$X) \leftarrow u(M_2[A])$.
- LDTU \$X,\$Y,\$Z (load tetra unsigned): $u(\$X) \leftarrow u(M_4[A])$.
- LDOU \$X,\$Y,\$Z (load octa unsigned): $u(\$X) \leftarrow u(M_8[A])$.

These instructions are analogous to LDB, LDW, LDT, and LDO, but they treat the memory data as *unsigned*; bit positions at the left of the register are set to zero when a short quantity is being lengthened. Thus, in the example above, LDBU \$1,\$2,\$3 with $\$2 + \$3 = 1005$ would set $\$1 \leftarrow #00000000000000ab$.

The instructions LDO and LDOU actually have exactly the same behavior, because no sign extension or padding with zeros is necessary when an octabyte is loaded into a register. But a good programmer will use LDO when the sign is relevant and LDOU when it is not; then readers of the program can better understand the significance of what is being loaded.

- LDHT \$X,\$Y,\$Z (load high tetra): $u(\$X) \leftarrow u(M_4[A]) \times 2^{32}$.

Here the tetrabyte $M_4[A]$ is loaded into the *left* half of $\$X$, and the right half is set to zero. For example, LDHT \$1,\$2,\$3 sets $\$1 \leftarrow #89ab cdef 0000 0000$, assuming (6) with $\$2 + \$3 = 1005$.

- LDA \$X,\$Y,\$Z (load address): $u(\$X) \leftarrow A$.

This instruction, which puts a memory address into a register, is essentially the same as the ADDU instruction described below. Sometimes the words “load address” describe its purpose better than the words “add unsigned.”

- STB \$X,\$Y,\$Z (store byte): $s(M_1[A]) \leftarrow s(\$X)$.
- STW \$X,\$Y,\$Z (store wyde): $s(M_2[A]) \leftarrow s(\$X)$.
- STT \$X,\$Y,\$Z (store tetra): $s(M_4[A]) \leftarrow s(\$X)$.
- STO \$X,\$Y,\$Z (store octa): $s(M_8[A]) \leftarrow s(\$X)$.

These instructions go the other way, placing register data into the memory. Overflow is possible if the (signed) number in the register lies outside the range of the memory field. For example, suppose register $\$1$ contains the number $-65536 = #ffffffffff0000$. Then if $\$2 = 1000$, $\$3 = 2$, and (6) holds,

```
STB $1,$2,$3 sets M8[1000] ← #0123006789ab cdef (with overflow);
STW $1,$2,$3 sets M8[1000] ← #0123000089ab cdef (with overflow);
STT $1,$2,$3 sets M8[1000] ← #ffff000089ab cdef;
STO $1,$2,$3 sets M8[1000] ← #ffffffffff0000.
```

- **STBU \$X,\$Y,\$Z** (store byte unsigned):
 $u(M_1[A]) \leftarrow u(\$X) \bmod 2^8$.
- **STWU \$X,\$Y,\$Z** (store wyde unsigned):
 $u(M_2[A]) \leftarrow u(\$X) \bmod 2^{16}$.
- **STTU \$X,\$Y,\$Z** (store tetra unsigned):
 $u(M_4[A]) \leftarrow u(\$X) \bmod 2^{32}$.
- **STOU \$X,\$Y,\$Z** (store octa unsigned): $u(M_8[A]) \leftarrow u(\$X)$.

These instructions have exactly the same effect on memory as their signed counterparts **STB**, **STW**, **STT**, and **STO**, but overflow never occurs.

- **STHT \$X,\$Y,\$Z** (store high tetra): $u(M_4[A]) \leftarrow \lfloor u(\$X)/2^{32} \rfloor$.

The left half of register **\$X** is stored in memory tetrabyte $M_4[A]$.

- **STCO X,\$Y,\$Z** (store constant octabyte): $u(M_8[A]) \leftarrow X$.

A constant between 0 and 255 is stored in memory octabyte $M_8[A]$.

Arithmetic operators. Most of MMIX's operations take place strictly between registers. We might as well begin our study of the register-to-register operations by considering addition, subtraction, multiplication, and division, because computers are supposed to be able to compute.

- **ADD \$X,\$Y,\$Z** (add): $s(\$X) \leftarrow s(\$Y) + s(\$Z)$.
- **SUB \$X,\$Y,\$Z** (subtract): $s(\$X) \leftarrow s(\$Y) - s(\$Z)$.
- **MUL \$X,\$Y,\$Z** (multiply): $s(\$X) \leftarrow s(\$Y) \times s(\$Z)$.
- **DIV \$X,\$Y,\$Z** (divide): $s(\$X) \leftarrow \lfloor s(\$Y)/s(\$Z) \rfloor$ [$\$Z \neq 0$], and
 $s(rR) \leftarrow s(\$Y) \bmod s(\$Z)$.

Sums, differences, and products need no further discussion. The **DIV** command forms the quotient and remainder as defined in Section 1.2.4; the remainder goes into the special *remainder register* **rR**, where it can be examined by using the instruction **GET \$X,rR** described below. If the divisor **\$Z** is zero, **DIV** sets **\$X** $\leftarrow 0$ and **rR** $\leftarrow \$Y$ (see Eq. 1.2.4-(1)); an “integer divide check” also occurs.

- **ADDU \$X,\$Y,\$Z** (add unsigned): $u(\$X) \leftarrow (u(\$Y) + u(\$Z)) \bmod 2^{64}$.
- **SUBU \$X,\$Y,\$Z** (subtract unsigned): $u(\$X) \leftarrow (u(\$Y) - u(\$Z)) \bmod 2^{64}$.
- **MULU \$X,\$Y,\$Z** (multiply unsigned): $u(rH\$X) \leftarrow u(\$Y) \times u(\$Z)$.
- **DIVU \$X,\$Y,\$Z** (divide unsigned): $u(\$X) \leftarrow \lfloor u(rD\$Y)/u(\$Z) \rfloor$, $u(rR) \leftarrow u(rD\$Y) \bmod u(\$Z)$, if $u(\$Z) > u(rD)$; otherwise $\$X \leftarrow rD$, **rR** $\leftarrow \$Y$.

Arithmetic on unsigned numbers never causes overflow. A full 16-byte product is formed by the **MULU** command, and the upper half goes into the special *himult register* **rH**. For example, when the unsigned number **#9e3779b97f4a7c16** in (2) and (4) above is multiplied by itself we get

$$rH \leftarrow \#61c8864680b583ea, \quad \$X \leftarrow \#1bb32095ccdd51e4. \quad (7)$$

In this case the value of **rH** has turned out to be exactly 2^{64} minus the original number **#9e3779b97f4a7c16**; this is not a coincidence! The reason is that (2) actually gives the first 64 bits of the binary representation of the golden ratio $\phi^{-1} = \phi - 1$, if we place a binary radix point at the *left*. (See Table 2 in Appendix A.) Squaring gives us an approximation to the binary representation of $\phi^{-2} = 1 - \phi^{-1}$, with the radix point now at the left of **rH**.

Division with DIVU yields the 8-byte quotient and remainder of a 16-byte dividend with respect to an 8-byte divisor. The upper half of the dividend appears in the special *dividend register* rD, which is zero at the beginning of a program; this register can be set to any desired value with the command PUT rD,\$Z described below. If rD is greater than or equal to the divisor, DIVU \$X,\$Y,\$Z simply sets \$X \leftarrow rD and rR \leftarrow \$Y. (This case always arises when \$Z is zero.) But DIVU never causes an integer divide check.

The ADDU instruction computes a memory address A, according to definition (5); therefore, as discussed earlier, we sometimes give ADDU the alternative name LDA. The following related commands also help with address calculation.

- 2ADDU \$X,\$Y,\$Z (times 2 and add unsigned):
 $u(\$X) \leftarrow (u(\$Y) \times 2 + u(\$Z)) \bmod 2^{64}$.
- 4ADDU \$X,\$Y,\$Z (times 4 and add unsigned):
 $u(\$X) \leftarrow (u(\$Y) \times 4 + u(\$Z)) \bmod 2^{64}$.
- 8ADDU \$X,\$Y,\$Z (times 8 and add unsigned):
 $u(\$X) \leftarrow (u(\$Y) \times 8 + u(\$Z)) \bmod 2^{64}$.
- 16ADDU \$X,\$Y,\$Z (times 16 and add unsigned):
 $u(\$X) \leftarrow (u(\$Y) \times 16 + u(\$Z)) \bmod 2^{64}$.

It is faster to execute the command 2ADDU \$X,\$Y,\$Y than to multiply by 3, if overflow is not an issue.

- NEG \$X,Y,\$Z (negate): $s(\$X) \leftarrow Y - s(\$Z)$.
- NEGU \$X,Y,\$Z (negate unsigned): $u(\$X) \leftarrow (Y - u(\$Z)) \bmod 2^{64}$.

In these commands Y is simply an unsigned constant, not a register number (just as X was an unsigned constant in the STCO instruction). Usually Y is zero, in which case we can write simply NEG \$X,\$Z or NEGU \$X,\$Z.

- SL \$X,\$Y,\$Z (shift left): $s(\$X) \leftarrow s(\$Y) \times 2^{u(\$Z)}$.
- SLU \$X,\$Y,\$Z (shift left unsigned): $u(\$X) \leftarrow (u(\$Y) \times 2^{u(\$Z)}) \bmod 2^{64}$.
- SR \$X,\$Y,\$Z (shift right): $s(\$X) \leftarrow \lfloor s(\$Y)/2^{u(\$Z)} \rfloor$.
- SRU \$X,\$Y,\$Z (shift right unsigned): $u(\$X) \leftarrow \lfloor u(\$Y)/2^{u(\$Z)} \rfloor$.

SL and SLU both produce the same result in \$X, but SL might overflow while SLU never does. SR extends the sign when shifting right, but SRU shifts zeros in from the left. Therefore SR and SRU produce the same result in \$X if and only if \$Y is nonnegative or \$Z is zero. The SL and SR instructions are much faster than MUL and DIV by powers of 2. An SLU instruction is much faster than MULU by a power of 2, although it does not affect rH as MULU does. An SRU instruction is much faster than DIVU by a power of 2, although it is not affected by rD. The notation $y \ll z$ is often used to denote the result of shifting a binary value y to the left by z bits; similarly, $y \gg z$ denotes shifting to the right.

- CMP \$X,\$Y,\$Z (compare):
 $s(\$X) \leftarrow [s(\$Y) > s(\$Z)] - [s(\$Y) < s(\$Z)]$.
- CMPU \$X,\$Y,\$Z (compare unsigned):
 $s(\$X) \leftarrow [u(\$Y) > u(\$Z)] - [u(\$Y) < u(\$Z)]$.

These instructions each set \$X to either -1, 0, or 1, depending on whether register \$Y is less than, equal to, or greater than register \$Z.

Conditional instructions. Several instructions base their actions on whether a register is positive, or negative, or zero, etc.

- CSN $\$X, \$Y, \$Z$ (conditional set if negative): if $s(\$Y) < 0$, set $\$X \leftarrow \Z .
- CSZ $\$X, \$Y, \$Z$ (conditional set if zero): if $\$Y = 0$, set $\$X \leftarrow \Z .
- CSP $\$X, \$Y, \$Z$ (conditional set if positive): if $s(\$Y) > 0$, set $\$X \leftarrow \Z .
- CSOD $\$X, \$Y, \$Z$ (conditional set if odd): if $s(\$Y) \bmod 2 = 1$, set $\$X \leftarrow \Z .
- CSNN $\$X, \$Y, \$Z$ (conditional set if nonnegative): if $s(\$Y) \geq 0$, set $\$X \leftarrow \Z .
- CSNZ $\$X, \$Y, \$Z$ (conditional set if nonzero): if $\$Y \neq 0$, set $\$X \leftarrow \Z .
- CSNP $\$X, \$Y, \$Z$ (conditional set if nonpositive): if $s(\$Y) \leq 0$, set $\$X \leftarrow \Z .
- CSEV $\$X, \$Y, \$Z$ (conditional set if even): if $s(\$Y) \bmod 2 = 0$, set $\$X \leftarrow \Z .

If register $\$Y$ satisfies the stated condition, register $\$Z$ is copied to register $\$X$; otherwise nothing happens. A register is negative if and only if its leading (leftmost) bit is 1. A register is odd if and only if its trailing (rightmost) bit is 1.

- ZSN $\$X, \$Y, \$Z$ (zero or set if negative): $\$X \leftarrow \$Z [s(\$Y) < 0]$.
- ZSZ $\$X, \$Y, \$Z$ (zero or set if zero): $\$X \leftarrow \$Z [\$Y = 0]$.
- ZSP $\$X, \$Y, \$Z$ (zero or set if positive): $\$X \leftarrow \$Z [s(\$Y) > 0]$.
- ZSOD $\$X, \$Y, \$Z$ (zero or set if odd): $\$X \leftarrow \$Z [s(\$Y) \bmod 2 = 1]$.
- ZSNN $\$X, \$Y, \$Z$ (zero or set if nonnegative): $\$X \leftarrow \$Z [s(\$Y) \geq 0]$.
- ZSNZ $\$X, \$Y, \$Z$ (zero or set if nonzero): $\$X \leftarrow \$Z [\$Y \neq 0]$.
- ZSNP $\$X, \$Y, \$Z$ (zero or set if nonpositive): $\$X \leftarrow \$Z [s(\$Y) \leq 0]$.
- ZSEV $\$X, \$Y, \$Z$ (zero or set if even): $\$X \leftarrow \$Z [s(\$Y) \bmod 2 = 0]$.

If register $\$Y$ satisfies the stated condition, register $\$Z$ is copied to register $\$X$; otherwise register $\$X$ is set to zero.

Bitwise operations. We often find it useful to think of an octabyte x as a *vector* $v(x)$ of 64 individual bits, and to perform operations simultaneously on each component of two such vectors.

- AND $\$X, \$Y, \$Z$ (bitwise and): $v(\$X) \leftarrow v(\$Y) \wedge v(\$Z)$.
- OR $\$X, \$Y, \$Z$ (bitwise or): $v(\$X) \leftarrow v(\$Y) \vee v(\$Z)$.
- XOR $\$X, \$Y, \$Z$ (bitwise exclusive-or): $v(\$X) \leftarrow v(\$Y) \oplus v(\$Z)$.
- ANDN $\$X, \$Y, \$Z$ (bitwise and-not): $v(\$X) \leftarrow v(\$Y) \wedge \bar{v}(\$Z)$.
- ORN $\$X, \$Y, \$Z$ (bitwise or-not): $v(\$X) \leftarrow v(\$Y) \vee \bar{v}(\$Z)$.
- NAND $\$X, \$Y, \$Z$ (bitwise not-and): $\bar{v}(\$X) \leftarrow v(\$Y) \wedge v(\$Z)$.
- NOR $\$X, \$Y, \$Z$ (bitwise not-or): $\bar{v}(\$X) \leftarrow v(\$Y) \vee v(\$Z)$.
- NXOR $\$X, \$Y, \$Z$ (bitwise not-exclusive-or): $\bar{v}(\$X) \leftarrow v(\$Y) \oplus v(\$Z)$.

Here \bar{v} denotes the *complement* of vector v , obtained by changing 0 to 1 and 1 to 0. The binary operations \wedge , \vee , and \oplus , defined by the rules

$$\begin{array}{lll} 0 \wedge 0 = 0, & 0 \vee 0 = 0, & 0 \oplus 0 = 0, \\ 0 \wedge 1 = 0, & 0 \vee 1 = 1, & 0 \oplus 1 = 1, \\ 1 \wedge 0 = 0, & 1 \vee 0 = 1, & 1 \oplus 0 = 1, \\ 1 \wedge 1 = 1, & 1 \vee 1 = 1, & 1 \oplus 1 = 0, \end{array} \tag{8}$$

are applied independently to each bit. Anding is the same as multiplying or taking the minimum; oring is the same as taking the maximum. Exclusive-oring is the same as adding mod 2.

- **MUX \$X,\$Y,\$Z** (bitwise multiplex): $v(\$X) \leftarrow (v(\$Y) \wedge v(rM)) \vee (v(\$Z) \wedge \bar{v}(rM))$. The MUX operation combines two bit vectors by looking at the special *multiplex mask register* rM, choosing bits of \$Y where rM is 1 and bits of \$Z where rM is 0.
- **SADD \$X,\$Y,\$Z** (sideways add): $s(\$X) \leftarrow s(\sum(v(\$Y) \wedge \bar{v}(\$Z)))$. The SADD operation counts the number of bit positions in which register \$Y has a 1 while register \$Z has a 0.

Bytewise operations. Similarly, we can regard an octabyte x as a vector $b(x)$ of eight individual bytes, each of which is an integer between 0 and 255; or we can think of it as a vector $w(x)$ of four individual wydes, or a vector $t(x)$ of two unsigned tetras. The following operations deal with all components at once.

- **BDIF \$X,\$Y,\$Z** (byte difference): $b(\$X) \leftarrow b(\$Y) \dot{-} b(\$Z)$.
- **WDIF \$X,\$Y,\$Z** (wyde difference): $w(\$X) \leftarrow w(\$Y) \dot{-} w(\$Z)$.
- **TDIF \$X,\$Y,\$Z** (tetra difference): $t(\$X) \leftarrow t(\$Y) \dot{-} t(\$Z)$.
- **ODIF \$X,\$Y,\$Z** (octa difference): $u(\$X) \leftarrow u(\$Y) \dot{-} u(\$Z)$.

Here $\dot{-}$ denotes the operation of *saturating subtraction*,

$$y \dot{-} z = \max(0, y - z). \quad (9)$$

These operations have important applications to text processing, as well as to computer graphics (when the bytes or wydes represent pixel values). Exercises 27–30 discuss some of their basic properties.

We can also regard an octabyte as an 8×8 Boolean matrix, that is, as an 8×8 array of 0s and 1s. Let $m(x)$ be the matrix whose rows from top to bottom are the bytes of x from left to right; and let $m^T(x)$ be the transposed matrix, whose *columns* are the bytes of x . For example, if $x = \#9e3779b97f4a7c16$ is the octabyte (2), we have

$$m(x) = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}, \quad m^T(x) = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}. \quad (10)$$

This interpretation of octabytes suggests two operations that are quite familiar to mathematicians, but we will pause a moment to define them from scratch.

If A is an $m \times n$ matrix and B is an $n \times s$ matrix, and if \circ and \bullet are binary operations, the *generalized matrix product* $A \bullet B$ is the $m \times s$ matrix C defined by

$$C_{ij} = (A_{i1} \bullet B_{1j}) \circ (A_{i2} \bullet B_{2j}) \circ \cdots \circ (A_{in} \bullet B_{nj}) \quad (11)$$

for $1 \leq i \leq m$ and $1 \leq j \leq s$. [See K. E. Iverson, *A Programming Language* (Wiley, 1962), 23–24; we assume that \circ is associative.] An ordinary matrix product is obtained when \circ is $+$ and \bullet is \times , but we obtain important operations

on Boolean matrices if we let \circ be \vee or \oplus :

$$(A \underset{\times}{\vee} B)_{ij} = A_{i1}B_{1j} \vee A_{i2}B_{2j} \vee \cdots \vee A_{in}B_{nj}; \quad (12)$$

$$(A \underset{\times}{\oplus} B)_{ij} = A_{i1}B_{1j} \oplus A_{i2}B_{2j} \oplus \cdots \oplus A_{in}B_{nj}. \quad (13)$$

Notice that if the rows of A each contain at most one 1, at most one term in (12) or (13) is nonzero. The same is true if the columns of B each contain at most one 1. Therefore $A \underset{\times}{\vee} B$ and $A \underset{\times}{\oplus} B$ both turn out to be the same as the ordinary matrix product $A \underset{\times}{+} B = AB$ in such cases.

- **MOR \$X,\$Y,\$Z** (multiple or): $m^T(\$X) \leftarrow m^T(\$Y) \underset{\times}{\vee} m^T(\$Z)$;
equivalently, $m(\$X) \leftarrow m(\$Z) \underset{\times}{\vee} m(\$Y)$. (See exercise 32.)
- **MXOR \$X,\$Y,\$Z** (multiple exclusive-or): $m^T(\$X) \leftarrow m^T(\$Y) \underset{\times}{\oplus} m^T(\$Z)$;
equivalently, $m(\$X) \leftarrow m(\$Z) \underset{\times}{\oplus} m(\$Y)$.

These operations essentially set each byte of $\$X$ by looking at the corresponding byte of $\$Z$ and using its bits to select bytes of $\$Y$; the selected bytes are then ored or xored together. If, for example, we have

$$\$Z = \#0102040810204080, \quad (14)$$

then both **MOR** and **MXOR** will set register $\$X$ to the *byte reversal* of register $\$Y$: The k th byte from the left of $\$X$ will be set to the k th byte from the right of $\$Y$, for $1 \leq k \leq 8$. On the other hand if $\$Z = \#00000000000000ff$, **MOR** and **MXOR** will set all bytes of $\$X$ to zero except for the rightmost byte, which will become either the **OR** or the **XOR** of all eight bytes of $\$Y$. Exercises 33–37 illustrate some of the many practical applications of these versatile commands.

Floating point operators. **MMIX** includes a full implementation of the famous IEEE/ANSI Standard 754 for floating point arithmetic. Complete details of the floating point operations appear in Section 4.2 and in the *MMIXware* document; a rough summary will suffice for our purposes here.

Every octabyte x represents a floating binary number $f(x)$ determined as follows: The leftmost bit of x is the sign ($0 = '+'$, $1 = '-'$); the next 11 bits are the *exponent* E ; the remaining 52 bits are the *fraction* F . The value represented is then

$$\begin{aligned} &\pm 0.0, \text{ if } E = F = 0 \text{ (zero);} \\ &\pm 2^{-1074}F, \text{ if } E = 0 \text{ and } F \neq 0 \text{ (denormal);} \\ &\pm 2^{E-1023}(1 + F/2^{52}), \text{ if } 0 < E < 2047 \text{ (normal);} \\ &\pm \infty, \text{ if } E = 2047 \text{ and } F = 0 \text{ (infinite);} \\ &\pm \text{NaN}(F/2^{52}), \text{ if } E = 2047 \text{ and } F \neq 0 \text{ (Not-a-Number).} \end{aligned}$$

The “short” floating point number $f(t)$ represented by a tetrabyte t is similar, but its exponent part has only 8 bits and its fraction has only 23; the normal case $0 < E < 255$ of a short float represents $\pm 2^{E-127}(1 + F/2^{23})$.

- **FADD \$X,\$Y,\$Z** (floating add): $f(\$X) \leftarrow f(\$Y) + f(\$Z)$.
- **FSUB \$X,\$Y,\$Z** (floating subtract): $f(\$X) \leftarrow f(\$Y) - f(\$Z)$.
- **FMUL \$X,\$Y,\$Z** (floating multiply): $f(\$X) \leftarrow f(\$Y) \times f(\$Z)$.
- **FDIV \$X,\$Y,\$Z** (floating divide): $f(\$X) \leftarrow f(\$Y)/f(\$Z)$.

- **FREM \$X,\$Y,\$Z** (floating remainder): $f(\$X) \leftarrow f(\$Y) \text{ rem } f(\$Z)$.
- **FSQRT \$X,\$Z or FSQRT \$X,Y,\$Z** (floating square root): $f(\$X) \leftarrow f(\$Z)^{1/2}$.
- **FINT \$X,\$Z or FINT \$X,Y,\$Z** (floating integer): $f(\$X) \leftarrow \text{int } f(\$Z)$.
- **FCMP \$X,\$Y,\$Z** (floating compare): $s(\$X) \leftarrow [f(\$Y) > f(\$Z)] - [f(\$Y) < f(\$Z)]$.
- **FEQL \$X,\$Y,\$Z** (floating equal to): $s(\$X) \leftarrow [f(\$Y) = f(\$Z)]$.
- **FUN \$X,\$Y,\$Z** (floating unordered): $s(\$X) \leftarrow [f(\$Y) \parallel f(\$Z)]$.
- **FCMPE \$X,\$Y,\$Z** (floating compare with respect to epsilon):

$$s(\$X) \leftarrow [f(\$Y) \succ f(\$Z) (f(rE))] - [f(\$Y) \prec f(\$Z) (f(rE))], \text{ see 4.2.2-(21)}$$
- **FEQLE \$X,\$Y,\$Z** (floating equivalent with respect to epsilon):

$$s(\$X) \leftarrow [f(\$Y) \approx f(\$Z) (f(rE))], \text{ see 4.2.2-(24)}$$
- **FUNE \$X,\$Y,\$Z** (floating unordered with respect to epsilon):

$$s(\$X) \leftarrow [f(\$Y) \parallel f(\$Z) (f(rE))]$$
- **FIX \$X,\$Z or FIX \$X,Y,\$Z** (convert floating to fixed): $s(\$X) \leftarrow \text{int } f(\$Z)$.
- **FIXU \$X,\$Z or FIXU \$X,Y,\$Z** (convert floating to fixed unsigned):

$$u(\$X) \leftarrow (\text{int } f(\$Z)) \bmod 2^{64}$$
- **FLOT \$X,\$Z or FLOT \$X,Y,\$Z** (convert fixed to floating): $f(\$X) \leftarrow s(\$Z)$.
- **FLOTU \$X,\$Z or FLOTU \$X,Y,\$Z** (convert fixed to floating unsigned):

$$f(\$X) \leftarrow u(\$Z)$$
- **SFLOT \$X,\$Z or SFLOT \$X,Y,\$Z** (convert fixed to short float):

$$f(\$X) \leftarrow f(T) \leftarrow s(\$Z)$$
- **SFLOTU \$X,\$Z or SFLOTU \$X,Y,\$Z** (convert fixed to short float unsigned):

$$f(\$X) \leftarrow f(T) \leftarrow u(\$Z)$$
- **LDSF \$X,\$Y,\$Z or LDSF \$X,A** (load short float): $f(\$X) \leftarrow f(M_4[A])$.
- **STSF \$X,\$Y,\$Z or STSF \$X,A** (store short float): $f(M_4[A]) \leftarrow f(\$X)$.

Assignment to a floating point quantity uses the current rounding mode to determine the appropriate value when an exact value cannot be assigned. Four rounding modes are supported: 1 (**ROUND_OFF**), 2 (**ROUND_UP**), 3 (**ROUND_DOWN**), and 4 (**ROUND_NEAR**). The Y field of **FSQRT**, **FINT**, **FIX**, **FIXU**, **FLOT**, **FLOTU**, **SFLOT**, and **SFLOTU** can be used to specify a rounding mode other than the current one, if desired. For example, **FIX \$X,ROUND_UP,\$Z** sets $s(\$X) \leftarrow [f(\$Z)]$. Operations **SFLOT** and **SFLOTU** first round as if storing into an anonymous tetrabyte T, then they convert that number to octabyte form.

The ‘int’ operation rounds to an integer. The operation $y \text{ rem } z$ is defined to be $y - nz$, where n is the nearest integer to y/z , or the nearest *even* integer in case of a tie. Special rules apply when the operands are infinite or NaN, and special conventions govern the sign of a zero result. The values +0.0 and -0.0 have different floating point representations, but **FEQL** calls them equal. All such technicalities are explained in the **MMIXware** document, and Section 4.2 explains why the technicalities are important.

Immediate constants. Programs often need to deal with small constant numbers. For example, we might want to add or subtract 1 from a register, or we might want to shift by 32, etc. In such cases it’s a nuisance to load the small constant from memory into another register. So **MMIX** provides a general mechanism by which such constants can be obtained “immediately” from an

instruction itself: *Every instruction we have discussed so far has a variant in which \$Z is replaced by the number Z*, unless the instruction treats \$Z as a floating point number.

For example, ‘ADD \$X,\$Y,\$Z’ has a counterpart ‘ADD \$X,\$Y,Z’, meaning $s(\$X) \leftarrow s(\$Y) + Z$; ‘SRU \$X,\$Y,\$Z’ has a counterpart ‘SRU \$X,\$Y,Z’, meaning $u(\$X) \leftarrow \lfloor u(\$Y)/2^Z \rfloor$; ‘FLOT \$X,\$Z’ has a counterpart ‘FLOT \$X,Z’, meaning $f(\$X) \leftarrow Z$. But ‘FADD \$X,\$Y,\$Z’ has no immediate counterpart.

The opcode for ‘ADD \$X,\$Y,\$Z’ is #20 and the opcode for ‘ADD \$X,\$Y,Z’ is #21; we use the same symbol ADD in both cases for simplicity. In general the opcode for the immediate variant of an operation is one greater than the opcode for the register variant.

Several instructions also feature *wyde immediate* constants, which range from #0000 = 0 to #ffff = 65535. These constants, which appear in the YZ bytes, can be shifted into the high, medium high, medium low, or low wyde positions of an octabyte.

- SETH \$X,YZ (set high wyde): $u(\$X) \leftarrow YZ \times 2^{48}$.
- SETMH \$X,YZ (set medium high wyde): $u(\$X) \leftarrow YZ \times 2^{32}$.
- SETML \$X,YZ (set medium low wyde): $u(\$X) \leftarrow YZ \times 2^{16}$.
- SETL \$X,YZ (set low wyde): $u(\$X) \leftarrow YZ$.
- INCH \$X,YZ (increase by high wyde): $u(\$X) \leftarrow (u(\$X) + YZ \times 2^{48}) \bmod 2^{64}$.
- INCMH \$X,YZ (increase by medium high wyde):
 $u(\$X) \leftarrow (u(\$X) + YZ \times 2^{32}) \bmod 2^{64}$.
- INCML \$X,YZ (increase by medium low wyde):
 $u(\$X) \leftarrow (u(\$X) + YZ \times 2^{16}) \bmod 2^{64}$.
- INCL \$X,YZ (increase by low wyde): $u(\$X) \leftarrow (u(\$X) + YZ) \bmod 2^{64}$.
- ORH \$X,YZ (bitwise or with high wyde): $v(\$X) \leftarrow v(\$X) \vee v(YZ \ll 48)$.
- ORMH \$X,YZ (bitwise or with medium high wyde):
 $v(\$X) \leftarrow v(\$X) \vee v(YZ \ll 32)$.
- ORML \$X,YZ (bitwise or with medium low wyde):
 $v(\$X) \leftarrow v(\$X) \vee v(YZ \ll 16)$.
- ORL \$X,YZ (bitwise or with low wyde): $v(\$X) \leftarrow v(\$X) \vee v(YZ)$.
- ANDNH \$X,YZ (bitwise and-not high wyde): $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ \ll 48)$.
- ANDNMH \$X,YZ (bitwise and-not medium high wyde):
 $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ \ll 32)$.
- ANDNML \$X,YZ (bitwise and-not medium low wyde):
 $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ \ll 16)$.
- ANDNL \$X,YZ (bitwise and-not low wyde): $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ)$.

Using at most four of these instructions, we can get any desired octabyte into a register without loading anything from the memory. For example, the commands

```
SETH $0,#0123; INCMH $0,#4567; INCML $0,#89ab; INCL $0,#cdef
put #0123456789ab cdef into register $0.
```

The MMIX assembly language allows us to write SET as an abbreviation for SETL, and SET \$X,\$Y as an abbreviation for the common operation OR \$X,\$Y,0.

Jumps and branches. Instructions are normally executed in their natural sequence. In other words, the command that is performed after MMIX has obeyed the tetrabyte in memory location @ is normally the tetrabyte found in memory location $\text{@} + 4$. (The symbol @ denotes the place where we're "at.") But jump and branch instructions allow this sequence to be interrupted.

- **JMP RA** (jump): $\text{@} \leftarrow \text{RA}$.

Here RA denotes a three-byte *relative address*, which could be written more explicitly as $\text{@} + 4 * \text{XYZ}$, namely XYZ tetrabytes following the current location @ . For example, 'JMP $\text{@} + 4 * 2$ ' is a symbolic form for the tetrabyte $\#f0000002$; if this instruction appears in location $\#1000$, the next instruction to be executed will be the one in location $\#1008$. We might in fact write 'JMP $\#1008$ '; but then the value of XYZ would depend on the location jumped from.

Relative offsets can also be negative, in which case the opcode increases by 1 and XYZ is the offset plus 2^{24} . For example, 'JMP $\text{@} - 4 * 2$ ' is the tetrabyte $\#f1fffffe$. Opcode $\#f0$ tells the computer to "jump forward" and opcode $\#f1$ tells it to "jump backward," but we write both as JMP. In fact, we usually write simply 'JMP Addr ' when we want to jump to location Addr , and the MMIX assembly program figures out the appropriate opcode and the appropriate value of XYZ . Such a jump will be possible unless we try to stray more than about 67 million bytes from our present location.

- **GO $\$X, \$Y, \$Z$** (go): $u(\$X) \leftarrow \text{@} + 4$, then $\text{@} \leftarrow A$.

The GO instruction allows us to jump to an *absolute address*, anywhere in memory; this address A is calculated by formula (5), exactly as in the load and store commands. Before going to the specified address, the location of the instruction that would ordinarily have come next is placed into register $\$X$. Therefore we could return to that location later by saying, for example, 'GO $\$X, \$X, 0$ ', with $Z = 0$ as an immediate constant.

- **BN $\$X, \text{RA}$** (branch if negative): if $s(\$X) < 0$, set $\text{@} \leftarrow \text{RA}$.
- **BZ $\$X, \text{RA}$** (branch if zero): if $\$X = 0$, set $\text{@} \leftarrow \text{RA}$.
- **BP $\$X, \text{RA}$** (branch if positive): if $s(\$X) > 0$, set $\text{@} \leftarrow \text{RA}$.
- **BOD $\$X, \text{RA}$** (branch if odd): if $s(\$X) \bmod 2 = 1$, set $\text{@} \leftarrow \text{RA}$.
- **BNN $\$X, \text{RA}$** (branch if nonnegative): if $s(\$X) \geq 0$, set $\text{@} \leftarrow \text{RA}$.
- **BNZ $\$X, \text{RA}$** (branch if nonzero): if $\$X \neq 0$, set $\text{@} \leftarrow \text{RA}$.
- **BNP $\$X, \text{RA}$** (branch if nonpositive): if $s(\$X) \leq 0$, set $\text{@} \leftarrow \text{RA}$.
- **BEV $\$X, \text{RA}$** (branch if even): if $s(\$X) \bmod 2 = 0$, set $\text{@} \leftarrow \text{RA}$.

A *branch* instruction is a conditional jump that depends on the contents of register $\$X$. The range of destination addresses RA is more limited than it was with JMP, because only two bytes are available to express the relative offset; but still we can branch to any tetrabyte between $\text{@} - 2^{18}$ and $\text{@} + 2^{18} - 4$.

- **PBN $\$X, \text{RA}$** (probable branch if negative): if $s(\$X) < 0$, set $\text{@} \leftarrow \text{RA}$.
- **PBZ $\$X, \text{RA}$** (probable branch if zero): if $\$X = 0$, set $\text{@} \leftarrow \text{RA}$.
- **PBP $\$X, \text{RA}$** (probable branch if positive): if $s(\$X) > 0$, set $\text{@} \leftarrow \text{RA}$.
- **PBOD $\$X, \text{RA}$** (probable branch if odd): if $s(\$X) \bmod 2 = 1$, set $\text{@} \leftarrow \text{RA}$.
- **PBNN $\$X, \text{RA}$** (probable branch if nonnegative): if $s(\$X) \geq 0$, set $\text{@} \leftarrow \text{RA}$.

- PBNZ \$X,RA (probable branch if nonzero): if $\$X \neq 0$, set $@ \leftarrow RA$.
- PBNP \$X,RA (probable branch if nonpositive): if $s(\$X) \leq 0$, set $@ \leftarrow RA$.
- PBEV \$X,RA (probable branch if even): if $s(\$X) \bmod 2 = 0$, set $@ \leftarrow RA$.

High-speed computers usually work fastest if they can anticipate when a branch will be taken, because foreknowledge helps them look ahead and get ready for future instructions. Therefore MMIX encourages programmers to give hints about whether branching is likely or not. Whenever a branch is expected to be taken more than half of the time, a wise programmer will say PB instead of B.

***Subroutine calls.** MMIX also has several instructions that facilitate efficient communication between subprograms, via a *register stack*. The details are somewhat technical and we will defer them until Section 1.4'; an informal description will suffice here. Short programs do not need to use these features.

- PUSHJ \$X,RA (push registers and jump): push(X) and set $rJ \leftarrow @ + 4$, then set $@ \leftarrow RA$.
- PUSHGO \$X,\$Y,\$Z (push registers and go): push(X) and set $rJ \leftarrow @ + 4$, then set $@ \leftarrow A$.

The special *return-jump register* rJ is set to the address of the tetrabyte following the PUSH command. The action “push(X)” means, roughly speaking, that local registers $\$0$ through $\$X$ are saved and made temporarily inaccessible. What used to be $\$(X+1)$ is now $\$0$, what used to be $\$(X+2)$ is now $\$1$, etc. But all registers $\$k$ for $k \geq rG$ remain unchanged; rG is the special *global threshold register*, whose value always lies between 32 and 255, inclusive.

Register $\$k$ is called *global* if $k \geq rG$. It is called *local* if $k < rL$; here rL is the special *local threshold register*, which tells how many local registers are currently active. Otherwise, namely if $rL \leq k < rG$, register $\$k$ is called *marginal*, and $\$k$ is equal to zero whenever it is used as a source operand in a command. If a marginal register $\$k$ is used as a destination operand in a command, rL is automatically increased to $k + 1$ before the command is performed, thereby making $\$k$ local.

- POP X,YZ (pop registers and return): pop(X), then $@ \leftarrow rJ + 4 * YZ$.

Here “pop(X)” means, roughly speaking, that all but X of the current local registers become marginal, and then the local registers hidden by the most recent “push” that has not yet been “popped” are restored to their former values. Full details appear in Section 1.4', together with numerous examples.

- SAVE \$X,0 (save process state): $u(\$X) \leftarrow \text{context}$.
- UNSAVE \$Z (restore process state): $\text{context} \leftarrow u(\$Z)$.

The SAVE instruction stores all current registers in memory at the top of the register stack, and puts the address of the topmost stored octabyte into $u(\$X)$. Register $\$X$ must be global; that is, X must be $\geq rG$. All of the currently local and global registers are saved, together with special registers like rA , rD , rE , rG , rH , rJ , rM , rR , and several others that we have not yet discussed. The UNSAVE instruction takes the address of such a topmost octabyte and restores the associated context, essentially undoing a previous SAVE. The value of rL is set to zero by SAVE, but restored by UNSAVE. MMIX has special registers called

the *register stack offset* (rO) and *register stack pointer* (rS), which control the PUSH, POP, SAVE, and UNSAVE operations. (Again, full details can be found in Section 1.4.)

***System considerations.** Several opcodes, intended primarily for ultrafast and/or parallel versions of the MMIX architecture, are of interest only to advanced users, but we should at least mention them here. Some of the associated operations are similar to the “probable branch” commands, in the sense that they give hints to the machine about how to plan ahead for maximum efficiency. Most programmers do not need to use these instructions, except perhaps SYNCID.

- LDUNC $\$X, \$Y, \$Z$ (load octa uncached): $s(\$X) \leftarrow s(M_8[A])$.
- STUNC $\$X, \$Y, \$Z$ (store octa uncached): $s(M_8[A]) \leftarrow s(\$X)$.

These commands perform the same operations as LD0 and ST0, but they also inform the machine that the loaded or stored octabyte and its near neighbors will probably not be read or written in the near future.

- PRELD $X, \$Y, \Z (preload data).

Says that many of the bytes $M[A]$ through $M[A + X]$ will probably be loaded or stored in the near future.

- PREST $X, \$Y, \Z (prestore data).

Says that all of the bytes $M[A]$ through $M[A + X]$ will definitely be written (stored) before they are next read (loaded).

- PREGO $X, \$Y, \Z (prefetch to go).

Says that many of the bytes $M[A]$ through $M[A + X]$ will probably be used as instructions in the near future.

- SYNCID $X, \$Y, \Z (synchronize instructions and data).

Says that all of the bytes $M[A]$ through $M[A + X]$ must be fetched again before being interpreted as instructions. *MMIX* is allowed to assume that a program’s instructions do not change after the program has begun, unless the instructions have been prepared by SYNCID. (See exercise 57.)

- SYNCD $X, \$Y, \Z (synchronize data).

Says that all of bytes $M[A]$ through $M[A + X]$ must be brought up to date in the physical memory, so that other computers and input/output devices can read them.

- SYNC XYZ (synchronize).

Restricts parallel activities so that different processors can cooperate reliably; see *MMIXware* for details. XYZ must be 0, 1, 2, or 3.

- CSWAP $\$X, \$Y, \$Z$ (compare and swap octabytes).

If $u(M_8[A]) = u(rP)$, where rP is the special *prediction register*, set $u(M_8[A]) \leftarrow u(\$X)$ and $u(\$X) \leftarrow 1$. Otherwise set $u(rP) \leftarrow u(M_8[A])$ and $u(\$X) \leftarrow 0$. This is an atomic (indivisible) operation, useful when independent computers share a common memory.

- LDVTS $\$X, \$Y, \$Z$ (load virtual translation status).

This instruction, described in *MMIXware*, is for the operating system only.

***Interrupts.** The normal flow of instructions from one tetrabyte to the next can be changed not only by jumps and branches but also by less predictable events like overflow or external signals. Real-world machines must also cope with such things as security violations and hardware failures. MMIX distinguishes two kinds of program interruptions: “trips” and “traps.” A trip sends control to a *trip handler*, which is part of the user’s program; a trap sends control to a *trap handler*, which is part of the operating system.

Eight kinds of exceptional conditions can arise when MMIX is doing arithmetic, namely integer divide check (D), integer overflow (V), float-to-fix overflow (W), invalid floating operation (I), floating overflow (O), floating underflow (U), floating division by zero (Z), and floating inexact (X). The special *arithmetic status register* rA holds current information about all these exceptions. The eight bits of its rightmost byte are called its *event bits*, and they are named D_BIT (#80), V_BIT (#40), . . . , X_BIT (#01), in order DVWIOUZX.

The eight bits just to the left of the event bits in rA are called the *enable bits*; they appear in the same order DVWIOUZX. When an exceptional condition occurs during some arithmetic operation, MMIX looks at the corresponding enable bit before proceeding to the next instruction. If the enable bit is 0, the corresponding event bit is set to 1; otherwise the machine invokes a trip handler by “tripping” to location #10 for exception D, #20 for exception V, . . . , #80 for exception X. Thus the event bits of rA record the exceptions that have not caused trips. (If more than one enabled exception occurs, the leftmost one takes precedence. For example, simultaneous O and X is handled by O.)

The two bits of rA just to the left of the enable bits hold the current rounding mode, mod 4. The other 46 bits of rA should be zero. A program can change the setting of rA at any time, using the PUT command discussed below.

- TRIP X, Y, Z or TRIP X, YZ or TRIP XYZ (trip).

This command forces a trip to the handler at location #00.

Whenever a trip occurs, MMIX uses five special registers to record the current state: the *bootstrap register* rB, the *where-interrupted register* rW, the *execution register* rX, the *Y operand register* rY, and the *Z operand register* rZ. First rB is set to \$255, then \$255 is set to rJ, and rW is set to @ + 4. The left half of rX is set to #8000 0000, and the right half is set to the instruction that tripped. If the interrupted instruction was not a store command, rY is set to \$Y and rZ is set to \$Z (or to Z in case of an immediate constant); otherwise rY is set to A (the memory address of the store command) and rZ is set to \$X (the quantity to be stored). Finally control passes to the handler by setting @ to the handler address (#00 or #10 or . . . or #80).

- TRAP X, Y, Z or TRAP X, YZ or TRAP XYZ (trap).

This command is analogous to TRIP, but it forces a trap to the operating system. Special registers rBB, rWW, rXX, rYY, and rZZ take the place of rB, rW, rX, rY, and rZ; the special *trap address register* rT supplies the address of the trap handler, which is placed in @. Section 1.3.2’ describes several TRAP commands that provide simple input/output operations. The normal way to conclude a

program is to say ‘TRAP 0’; this instruction is the tetrabyte #00000000, so you might run into it by mistake.

The *MMIXware* document gives further details about external interrupts, which are governed by the special *interrupt mask register* rK and *interrupt request register* rQ. Dynamic traps, which arise when $rK \wedge rQ \neq 0$, are handled at address rTT instead of rT.

- **RESUME 0** (resume after interrupt).

If $s(rX)$ is negative, MMIX simply sets $@ \leftarrow rW$ and takes its next instruction from there. Otherwise, if the leading byte of rX is zero, MMIX sets $@ \leftarrow rW - 4$ and executes the instruction in the lower half of rX as if it had appeared in that location. (This feature can be used even if no interrupt has occurred. The inserted instruction must not itself be RESUME.) Otherwise MMIX performs special actions described in the *MMIXware* document and of interest primarily to the operating system; see exercise 1.4.3–14.

The complete instruction set. Table 1 shows the symbolic names of all 256 opcodes, arranged by their numeric values in hexadecimal notation. For example, ADD appears in the upper half of the row labeled #2x and in the column labeled #0 at the top, so ADD is opcode #20; ORL appears in the lower half of the row labeled #Ex and in the column labeled #B at the bottom, so ORL is opcode #EB.

Table 1 actually says ‘ADD [I]’, not ‘ADD’, because the symbol ADD really stands for two opcodes. Opcode #20 arises from ADD \$X,\$Y,\$Z using register \$Z, while opcode #21 arises from ADD \$X,\$Y,Z using the immediate constant Z. When a distinction is necessary, we say that opcode #20 is ADD and opcode #21 is ADDI (“add immediate”); similarly, #F0 is JMP and #F1 is JMPB (“jump backward”). This gives every opcode a unique name. However, the extra I and B are generally dropped for convenience when we write MMIX programs.

We have discussed nearly all of MMIX’s opcodes. Two of the stragglers are

- GET \$X,Z (get from special register): $u(\$X) \leftarrow u(g[Z])$, where $0 \leq Z < 32$.
- PUT X,\$Z (put into special register): $u(g[X]) \leftarrow u(\$Z)$, where $0 \leq X < 32$.

Each special register has a code number between 0 and 31. We speak of registers rA, rB, . . . , as aids to human understanding; but register rA is really g[21] from the machine’s point of view, and register rB is really g[0], etc. The code numbers appear in Table 2 on page 21.

GET commands are unrestricted, but certain things cannot be PUT: No value can be put into rG that is greater than 255, less than 32, or less than the current setting of rL. No value can be put into rA that is greater than #3ffff. If a program tries to increase rL with the PUT command, rL will stay unchanged. Moreover, a program cannot PUT anything into rC, rN, rO, rS, rI, rT, rTT, rK, rQ, rU, or rV; these “extraspecial” registers have code numbers in the range 8–18.

Most of the special registers have already been mentioned in connection with specific instructions, but MMIX also has a “clock register” or *cycle counter*, rC, which keeps advancing; an *interval counter*, rI, which keeps decreasing, and which requests an interrupt when it reaches zero; a *serial number register*, rN, which gives each MMIX machine a unique number; a *usage counter*, rU, which

Table 1
THE OPCODES OF MMIX

	# 0	# 1	# 2	# 3	# 4	# 5	# 6	# 7	
# 0x	TRAP $5v$	FCMP v	FUN v	FEQL v	FADD $4v$	FIX $4v$	FSUB $4v$	FIXU $4v$	# 0x
	FLOT[I] $4v$		FLOTU[I] $4v$		SFLOT[I] $4v$		SFLOTU[I] $4v$		
# 1x	FMUL $4v$	FCMPE $4v$	FUNE v	FEQLE $4v$	FDIV $40v$	FSQRT $40v$	FREM $4v$	FINT $4v$	# 1x
	MUL[I] $10v$		MULU[I] $10v$		DIV[I] $60v$		DIVU[I] $60v$		
# 2x	ADD[I] v		ADDU[I] v		SUB[I] v		SUBU[I] v		# 2x
	2ADDU[I] v		4ADDU[I] v		8ADDU[I] v		16ADDU[I] v		
# 3x	CMP[I] v		CMPU[I] v		NEG[I] v		NEGU[I] v		# 3x
	SL[I] v		SLU[I] v		SR[I] v		SRU[I] v		
# 4x	BN[B] $v+\pi$		BZ[B] $v+\pi$		BP[B] $v+\pi$		BOD[B] $v+\pi$		# 4x
	BNN[B] $v+\pi$		BNZ[B] $v+\pi$		BNP[B] $v+\pi$		BEV[B] $v+\pi$		
# 5x	PBN[B] $3v-\pi$		PBZ[B] $3v-\pi$		PBP[B] $3v-\pi$		PBOD[B] $3v-\pi$		# 5x
	PBNZ[B] $3v-\pi$		PBNZ[B] $3v-\pi$		PBNP[B] $3v-\pi$		PBEV[B] $3v-\pi$		
# 6x	CSN[I] v		CSZ[I] v		CSP[I] v		CSOD[I] v		# 6x
	CSNN[I] v		CSNZ[I] v		CSNP[I] v		CSEV[I] v		
# 7x	ZSN[I] v		ZSZ[I] v		ZSP[I] v		ZSOD[I] v		# 7x
	ZSNN[I] v		ZSNZ[I] v		ZSNP[I] v		ZSEV[I] v		
# 8x	LDB[I] $\mu+v$		LDBU[I] $\mu+v$		LDW[I] $\mu+v$		LDWU[I] $\mu+v$		# 8x
	LDT[I] $\mu+v$		LDTU[I] $\mu+v$		LDO[I] $\mu+v$		LDOU[I] $\mu+v$		
# 9x	LDSF[I] $\mu+v$		LDHT[I] $\mu+v$		CSWAP[I] $2\mu+2v$		LDUNC[I] $\mu+v$		# 9x
	LDVTS[I] v		PRELD[I] v		PREGO[I] v		GO[I] $3v$		
# Ax	STB[I] $\mu+v$		STBU[I] $\mu+v$		STW[I] $\mu+v$		STWU[I] $\mu+v$		# Ax
	STT[I] $\mu+v$		STTU[I] $\mu+v$		STO[I] $\mu+v$		STOU[I] $\mu+v$		
# Bx	STSF[I] $\mu+v$		STHT[I] $\mu+v$		STCO[I] $\mu+v$		STUNC[I] $\mu+v$		# Bx
	SYNCD[I] v		PREST[I] v		SYNCID[I] v		PUSHGO[I] $3v$		
# Cx	OR[I] v		ORN[I] v		NOR[I] v		XOR[I] v		# Cx
	AND[I] v		ANDN[I] v		NAND[I] v		NXOR[I] v		
# Dx	BDIF[I] v		WDIF[I] v		TDIF[I] v		ODIF[I] v		# Dx
	MUX[I] v		SADD[I] v		MOR[I] v		MXOR[I] v		
# Ex	SETH v	SETMH v	SETML v	SETL v	INCH v	INCMH v	INCML v	INCL v	# Ex
	ORH v	ORMH v	ORML v	ORL v	ANDNH v	ANDNMH v	ANDNML v	ANDNL v	
# Fx	JMP[B] v		PUSHJ[B] v		GETA[B] v		PUT[I] v		# Fx
	POP $3v$	RESUME $5v$	[UN]SAVE $20\mu+v$		SYNC v	SWYM v	GET v	TRIP $5v$	
	# 8	# 9	# A	# B	# C	# D	# E	# F	

$\pi = 2v$ if the branch is taken, $\pi = 0$ if the branch is not taken

increases by 1 whenever specified opcodes are executed; and a *virtual translation register*, rV, which defines a mapping from the “virtual” 64-bit addresses used in programs to the “actual” physical locations of installed memory. These special registers help make MMIX a complete, viable machine that could actually be built and run successfully; but they are not of importance to us in this book. The *MMIXware* document explains them fully.

- GETA \$X,RA (get address): $u(\$X) \leftarrow RA$.

This instruction loads a relative address into register \$X, using the same conventions as the relative addresses in branch commands. For example, GETA \$0,@ will set \$0 to the address of the instruction itself.

Table 2
SPECIAL REGISTERS OF MMIX

		code	saved?	put?
rA	arithmetic status register	21	✓	✓
rB	bootstrap register (trip)	0	✓	✓
rC	cycle counter	8		
rD	dividend register	1	✓	✓
rE	epsilon register	2	✓	✓
rF	failure location register	22		✓
rG	global threshold register	19	✓	✓
rH	himult register	3	✓	✓
rI	interval counter	12		
rJ	return-jump register	4	✓	✓
rK	interrupt mask register	15		
rL	local threshold register	20	✓	✓
rM	multiplex mask register	5	✓	✓
rN	serial number	9		
rO	register stack offset	10		
rP	prediction register	23	✓	✓
rQ	interrupt request register	16		
rR	remainder register	6	✓	✓
rS	register stack pointer	11		
rT	trap address register	13		
rU	usage counter	17		
rV	virtual translation register	18		
rW	where-interrupted register (trip)	24	✓	✓
rX	execution register (trip)	25	✓	✓
rY	Y operand (trip)	26	✓	✓
rZ	Z operand (trip)	27	✓	✓
rBB	bootstrap register (trap)	7		✓
rTT	dynamic trap address register	14		
rWW	where-interrupted register (trap)	28		✓
rXX	execution register (trap)	29		✓
rYY	Y operand (trap)	30		✓
rZZ	Z operand (trap)	31		✓

- SWYM X, Y, Z or SWYM X, YZ or SWYM XYZ (sympathize with your machinery).

The last of MMIX's 256 opcodes is, fortunately, the simplest of all. In fact, it is often called a no-op, because it performs no operation. It does, however, keep the machine running smoothly, just as real-world swimming helps to keep programmers healthy. Bytes X, Y, and Z are ignored.

Timing. In later parts of this book we will often want to compare different MMIX programs to see which is faster. Such comparisons aren't easy to make, in general, because the MMIX architecture can be implemented in many different ways. Although MMIX is a mythical machine, its mythical hardware exists in cheap, slow versions as well as in costly high-performance models. The running time of a program depends not only on the clock rate but also on the number of

functional units that can be active simultaneously and the degree to which they are pipelined; it depends on the techniques used to prefetch instructions before they are executed; it depends on the size of the random-access memory that is used to give the illusion of 2^{64} virtual bytes; and it depends on the sizes and allocation strategies of caches and other buffers, etc., etc.

For practical purposes, the running time of an MMIX program can often be estimated satisfactorily by assigning a fixed cost to each operation, based on the approximate running time that would be obtained on a high-performance machine with lots of main memory; so that's what we will do. Each operation will be assumed to take an integer number of v , where v (pronounced "oops")* is a unit that represents the clock cycle time in a pipelined implementation. Although the value of v decreases as technology improves, we always keep up with the latest advances because we measure time in units of v , not in nanoseconds. The running time in our estimates will also be assumed to depend on the number of memory references or *mems* that a program uses; this is the number of load and store instructions. For example, we will assume that each LDO (load octa) instruction costs $\mu + v$, where μ is the average cost of a memory reference. The total running time of a program might be reported as, say, $35\mu + 1000v$, meaning "35 mems plus 1000 oops." The ratio μ/v has been increasing steadily for many years; nobody knows for sure whether this trend will continue, but experience has shown that μ and v deserve to be considered independently.

Table 1, which is repeated also in the endpapers of this book, displays the assumed running time together with each opcode. Notice that most instructions take just $1v$, while loads and stores take $\mu+v$. A branch or probable branch takes $1v$ if predicted correctly, $3v$ if predicted incorrectly. Floating point operations usually take $4v$ each, although FDIV and FSQRT cost $40v$. Integer multiplication takes $10v$; integer division weighs in at $60v$.

Even though we will often use the assumptions of Table 1 for seat-of-the-pants estimates of running time, we must remember that the actual running time might be quite sensitive to the ordering of instructions. For example, integer division might cost only one cycle if we can find 60 other things to do between the time we issue the command and the time we need the result. Several LDB (load byte) instructions might need to reference memory only once, if they refer to the same octabyte. Yet the result of a load command is usually not ready for use in the immediately following instruction. Experience has shown that some algorithms work well with cache memory, and others do not; therefore μ is not really constant. Even the location of instructions in memory can have a significant effect on performance, because some instructions can be fetched together with others. Therefore the MMIXware package includes not only a simple simulator, which calculates running times by the rules of Table 1, but also a comprehensive *meta-simulator*, which runs MMIX programs under a wide range of different technological assumptions. Users of the meta-simulator can specify the

* The Greek letter *upsilon* (υ) is wider than an italic letter *vee* (v), but the author admits that this distinction is rather subtle. Readers who prefer to say *vee* instead of *oops* are free to do as they wish. The symbol is, however, an *upsilon*.

characteristics of the memory bus and the parameters of such things as caches for instructions and data, virtual address translation, pipelining and simultaneous instruction issue, branch prediction, etc. Given a configuration file and a program file, the meta-simulator determines precisely how long the specified hardware would need to run the program. Only the meta-simulator can be trusted to give reliable information about a program's actual behavior in practice; but such results can be difficult to interpret, because infinitely many configurations are possible. That's why we often resort to the much simpler estimates of Table 1.

No benchmark result should ever be taken at face value.
— BRIAN KERNIGHAN and CHRISTOPHER VAN WYK (1998)

MMIX versus reality. A person who understands the rudiments of MMIX programming has a pretty good idea of what today's general-purpose computers can do easily; MMIX is very much like all of them. But MMIX has been idealized in several ways, partly because the author has tried to design a machine that is somewhat "ahead of its time" so that it won't become obsolete too quickly. Therefore a brief comparison between MMIX and the computers actually being built at the turn of the millennium is appropriate. The main differences between MMIX and those machines are:

- Commercial machines do not ignore the low-order bits of memory addresses, as MMIX does when accessing $M_8[A]$; they usually insist that A be a multiple of 8. (We will find many uses for those precious low-order bits.)
- Commercial machines are usually deficient in their support of integer arithmetic. For example, they almost never produce the true quotient $\lfloor x/y \rfloor$ and true remainder $x \bmod y$ when x is negative or y is negative; they often throw away the upper half of a product. They don't treat left and right shifts as strict equivalents of multiplication and division by powers of 2. Sometimes they do not implement division in hardware at all; and when they do handle division, they usually assume that the upper half of the 128-bit dividend is zero. Such restrictions make high-precision calculations more difficult.
- Commercial machines do not perform FINT and FREM efficiently.
- Commercial machines do not (yet?) have the powerful MOR and MXOR operations. They usually have a half dozen or so ad hoc instructions that handle only the most common special cases of MOR.
- Commercial machines rarely have more than 64 general-purpose registers. The 256 registers of MMIX significantly decrease program length, because many variables and constants of a program can live entirely in those registers instead of in memory. Furthermore, MMIX's register stack is more flexible than the comparable mechanisms in existing computers.

All of these pluses for MMIX have associated minuses, because computer design always involves tradeoffs. The primary design goal for MMIX was to keep the machine as simple and clean and consistent and forward-looking as possible, without sacrificing speed and realism too greatly.

*And now I see with eye serene
The very pulse of the machine.*

— WILLIAM WORDSWORTH, *She Was a Phantom of Delight* (1804)

Summary. MMIX is a programmer-friendly computer that operates on 64-bit quantities called octabytes. It has the general characteristics of a so-called RISC (“reduced instruction set computer”); that is, its instructions have only a few different formats (OP X, Y, Z or OP X, YZ or OP XYZ), and each instruction either transfers data between memory and a register or involves only registers. Table 1 summarizes the 256 opcodes and their default running times; Table 2 summarizes the special registers that are sometimes important.

The following exercises give a quick review of the material in this section. Most of them are quite simple, and the reader should try to do nearly all of them.

EXERCISES

1. [00] The binary form of 2009 is $(11111011001)_2$; what is 2009 in hexadecimal?
2. [05] Which of the letters {A, B, C, D, E, F, a, b, c, d, e, f} are *odd* when considered as (a) hexadecimal digits? (b) ASCII characters?
3. [10] Four-bit quantities — half-bytes, or hexadecimal digits — are often called *nybbles*. Suggest a good name for *two-bit* quantities, so that we have a complete binary nomenclature ranging from bits to octabytes.
4. [15] A kilobyte (kB or KB) is 1000 bytes, and a megabyte (MB) is 1000 kB. What are the official names and abbreviations for larger numbers of bytes?
5. [M13] If α is any string of 0s and 1s, let $s(\alpha)$ and $u(\alpha)$ be the integers that it represents when regarded as a signed or unsigned binary number. Prove that, if x is any integer, we have

$$x = s(\alpha) \quad \text{if and only if} \quad x \equiv u(\alpha) \pmod{2^n} \text{ and } -2^{n-1} \leq x < 2^{n-1},$$

where n is the length of α .

- 6. [M20] Prove or disprove the following rule for negating an n -bit number in two’s complement notation: “Complement all the bits, then add 1.” (For example, #0...01 becomes #f...fe, then #f...ff; also #f...ff becomes #0...00, then #0...01.)
- 7. [M15] Could the formal definitions of LDHT and STHT have been stated as

$$s(\$X) \leftarrow s(M_4[A]) \times 2^{32} \quad \text{and} \quad s(M_4[A]) \leftarrow \lfloor s(\$X)/2^{32} \rfloor,$$

thus treating the numbers as signed rather than unsigned?

8. [10] If registers \$Y and \$Z represent numbers between 0 and 1 in which the binary radix point is assumed to be at the left of each register, (7) illustrates the fact that MULU forms a product in which the assumed radix point appears at the left of register rH. Suppose, on the other hand, that \$Z is an integer, with the radix point assumed at its right, while \$Y is a fraction between 0 and 1 as before. Where does the radix point lie after MULU in such a case?

9. [M10] Does the equation $s(\$Y) = s(\$X) \cdot s(\$Z) + s(rR)$ always hold after the instruction DIV \$X,\$Y,\$Z has been performed?

10. [M16] Give an example of DIV in which overflow occurs.
11. [M16] True or false: (a) Both MUL \$X,\$Y,\$Z and MULU \$X,\$Y,\$Z produce the same result in \$X. (b) If register rD is zero, both DIV \$X,\$Y,\$Z and DIVU \$X,\$Y,\$Z produce the same result in \$X.
- 12. [M20] Although ADDU \$X,\$Y,\$Z never signals overflow, we might want to know if a carry occurs at the left when adding \$Y to \$Z. Show that the carry can be computed with two further instructions.
13. [M21] Suppose MMIX had no ADD command, only its unsigned counterpart ADDU. How could a programmer tell whether overflow occurred when computing $s(Y) + s(Z)$?
14. [M21] Suppose MMIX had no SUB command, only its unsigned counterpart SRU. How could a programmer tell whether overflow occurred when computing $s(Y) - s(Z)$?
15. [M25] The product of two signed octabytes always lies between -2^{126} and 2^{126} , so it can always be expressed as a signed 16-byte quantity. Explain how to calculate the upper half of such a signed product.
16. [M23] Suppose MMIX had no MUL command, only its unsigned counterpart MULU. How could a programmer tell whether overflow occurred when computing $s(Y) \times s(Z)$?
- 17. [M22] Prove that unsigned integer division by 3 can always be done by multiplication: If register \$Y contains any unsigned integer y , and if register \$1 contains the constant #aaaaaaaaaaaaaaab, then the sequence

```
MULU $0,$Y,$1; GET $0,rH; SRU $X,$0,1
```

puts $\lfloor y/3 \rfloor$ into register \$X.

18. [M23] Continuing the previous exercise, prove or disprove that the instructions

```
MULU $0,$Y,$1; GET $0,rH; SRU $X,$0,2
```

put $\lfloor y/5 \rfloor$ in \$X if \$1 is an appropriate constant.

- 19. [M26] Continuing exercises 17 and 18, prove or disprove the following statement: Unsigned integer division by a constant can always be done using “high multiplication” followed by a right shift. More precisely, if $2^e < z < 2^{e+1}$ we can compute $\lfloor y/z \rfloor$ by computing $\lfloor ay/2^{64+e} \rfloor$, where $a = \lceil 2^{64+e}/z \rceil$, for $0 \leq y < 2^{64}$.
20. [16] Show that two cleverly chosen MMIX instructions will multiply by 25 faster than the single instruction MUL \$X,\$Y,25, if we assume that overflow will not occur.
21. [15] Describe the effects of SL, SLU, SR, and SRU when the unsigned value in register \$Z is 64 or more.
- 22. [15] Mr. B. C. Dull wrote a program in which he wanted to branch to location Case1 if the signed number in register \$1 was less than the signed number in register \$2. His solution was to write ‘SUB \$0,\$1,\$2; BN \$0,Case1’.
- What terrible mistake did he make? What should he have written instead?
- 23. [10] Continuing the previous exercise, what should Dull have written if his problem had been to branch if $s($1)$ was *less than or equal to* $s($2)$?
24. [M10] If we represent a subset S of $\{0, 1, \dots, 63\}$ by the bit vector

$$([0 \in S], [1 \in S], \dots, [63 \in S]),$$

the bitwise operations \wedge and \vee correspond respectively to set intersection ($S \cap T$) and set union ($S \cup T$). Which bitwise operation corresponds to set difference ($S \setminus T$)?

25. [10] The *Hamming distance* between two bit vectors is the number of positions in which they differ. Show that two MMIX instructions suffice to set register \$X equal to the Hamming distance between v(\$Y) and v(\$Z).

26. [10] What's a good way to compute 64 bit differences, $v(\$X) \leftarrow v(\$Y) \dot{-} v(\$Z)$?

► **27.** [20] Show how to use BDIF to compute the *maximum* and *minimum* of eight bytes at a time: $b(\$X) \leftarrow \max(b(\$Y), b(\$Z))$, $b(\$W) \leftarrow \min(b(\$Y), b(\$Z))$.

28. [16] How would you calculate eight *absolute pixel differences* $|b(\$Y) - b(\$Z)|$ simultaneously?

29. [21] The operation of *saturating addition* on n -bit pixels is defined by the formula

$$y + z = \min(2^n - 1, y + z).$$

Show that a sequence of three MMIX instructions will set $b(\$X) \leftarrow b(\$Y) + b(\$Z)$.

► **30.** [25] Suppose register \$0 contains eight ASCII characters. Find a sequence of three MMIX instructions that counts the number of *blank spaces* among those characters. (You may assume that auxiliary constants have been preloaded into other registers. A blank space is ASCII code #20.)

31. [22] Continuing the previous exercise, show how to count the number of characters in \$0 that have *odd parity* (an odd number of 1 bits).

32. [M20] True or false: If $C = A \bullet B$ then $C^T = B^T \bullet A^T$. (See (11).)

33. [20] What is the shortest sequence of MMIX instructions that will *cyclically shift* a register eight bits to the right? For example, #9e3779b97f4a7c16 would become #169e3779b97f4a7c.

► **34.** [21] Given eight bytes of ASCII characters in \$Z, explain how to convert them to the corresponding eight wyde characters of Unicode, using only two MMIX instructions to place the results in \$X and \$Y. How would you go the other way (back to ASCII)?

► **35.** [22] Show that two cleverly chosen MOR instructions will reverse the left-to-right order of all 64 bits in a given register \$Y.

► **36.** [20] Using only two instructions, create a mask that has #ff in all byte positions where \$Y differs from \$Z, #00 in all byte positions where \$Y equals \$Z.

► **37.** [HM30] (*Finite fields*.) Explain how to use MXOR for arithmetic in a field of 256 elements; each element of the field should be represented by a suitable octabyte.

38. [20] What does the following little program do?

```
SETL $1,0; SR $2,$0,56; ADD $1,$1,$2; SLU $0,$0,8; PBNZ $0,0-4*3.
```

► **39.** [20] Which of the following equivalent sequences of code is faster, based on the timing information of Table 1?

a) BN \$0,0+4*2; ADDU \$1,\$2,\$3 versus ADDU \$4,\$2,\$3; CSNN \$1,\$0,\$4.

b) BN \$0,0+4*3; SET \$1,\$2; JMP 0+4*2; SET \$1,\$3 versus CSNN \$1,\$0,\$2; CSN \$1,\$0,\$3.

c) BN \$0,0+4*3; ADDU \$1,\$2,\$3; JMP 0+4*2; ADDU \$1,\$4,\$5 versus ADDU \$1,\$2,\$3; ADDU \$6,\$4,\$5; CSN \$1,\$0,\$6.

d, e, f) Same as (a), (b), and (c), but with PBN in place of BN.

40. [10] What happens if you GO to an address that is not a multiple of 4?

- 41.** [20] True or false:
- The instructions CSOD \$X,\$Y,0 and ZSEV \$X,\$Y,\$X have exactly the same effect.
 - The instructions CMPU \$X,\$Y,0 and ZSNZ \$X,\$Y,1 have exactly the same effect.
 - The instructions MOR \$X,\$Y,1 and AND \$X,\$Y,#ff have exactly the same effect.
 - The instructions MXOR \$X,\$Y,#80 and SR \$X,\$Y,56 have exactly the same effect.
- 42.** [20] What is the best way to set register \$1 to the *absolute value* of the number in register \$0, if \$0 holds (a) a signed integer? (b) a floating point number?
- **43.** [28] Given a nonzero octabyte in \$Z, what is the fastest way to count how many leading and trailing zero bits it has? (For example, #13fd8124f32434a2 has three leading zeros and one trailing zero.)
- **44.** [M25] Suppose you want to emulate 32-bit arithmetic with MMIX. Show that it is easy to add, subtract, multiply, and divide signed tetrabytes, with overflow occurring whenever the result does not lie in the interval $[-2^{31} \dots 2^{31}]$.
- 45.** [10] Think of a way to remember the sequence DVWIOUZX.
- 46.** [05] The all-zeros tetrabyte #00000000 halts a program when it occurs as an MMIX instruction. What does the all-ones tetrabyte #ffffffff do?
- 47.** [05] What are the symbolic names of opcodes #DF and #55?
- 48.** [11] The text points out that opcodes LDO and LDOU perform exactly the same operation, with the same efficiency, regardless of the operand bytes X, Y, and Z. What other pairs of opcodes are equivalent in this sense?
- **49.** [22] After the following “number one” program has been executed, what changes to registers and memory have taken place? (For example, what is the final setting of \$1? of rA? of rB?)

```

NEG    $1,1
STCO   1,$1,1
CMPU   $1,$1,1
STB    $1,$1,$1
LDOU   $1,$1,$1
INCH   $1,1
16ADDU $1,$1,$1
MULU   $1,$1,$1
PUT    rA,1
STW    $1,$1,1
SADD   $1,$1,1
FLOT   $1,$1
PUT    rB,$1
XOR    $1,$1,1
PBOD   $1,0-4*1
NOR    $1,$1,$1
SR     $1,$1,1
SRU   $1,$1,1  ■

```

- **50.** [14] What is the execution time of the program in the preceding exercise?
- 51.** [14] Convert the “number one” program of exercise 49 to a sequence of tetrabytes in hexadecimal notation.
- 52.** [22] For each MMIX opcode, consider whether there is a way to set the X, Y, and Z bytes so that the result of the instruction is precisely equivalent to SWYM (except that

the execution time may be longer). Assume that nothing is known about the contents of any registers or any memory locations. Whenever it is possible to produce a no-op, state how it can be done. *Examples:* INCL is a no-op if X = 255 and Y = Z = 0. BZ is a no-op if Y = 0 and Z = 1. MULU can never be a no-op, since it affects rH.

- 53. [15] List all MMIX opcodes that can possibly change the value of rH.
- 54. [20] List all MMIX opcodes that can possibly change the value of ra.
- 55. [21] List all MMIX opcodes that can possibly change the value of rL.
- 56. [28] Location #2000000000000000 contains a signed integer number, x . Write two programs that compute x^{13} in register \$0. One program should use the minimum number of MMIX memory locations; the other should use the minimum possible execution time. Assume that x^{13} fits into a single octabyte, and that all necessary constants have been preloaded into global registers.
- 57. [20] When a program changes one or more of its own instructions in memory, it is said to have *self-modifying code*. MMIX insists that a SYNCID command be issued before such modified commands are executed. Explain why self-modifying code is usually undesirable in a modern computer.
- 58. [50] Write a book about operating systems, which includes the complete design of an NNIX kernel for the MMIX architecture.

Them fellers is a-mommixin' everything.

— V. RANDOLPH and G. P. WILSON, *Down in the Holler* (1953)

1.3.2'. The MMIX Assembly Language

A symbolic language is used to make MMIX programs considerably easier to read and to write, and to save the programmer from worrying about tedious clerical details that often lead to unnecessary errors. This language, MMIXAL (“MMIX Assembly Language”), is an extension of the notation used for instructions in the previous section. Its main features are the optional use of alphabetic names to stand for numbers, and a label field to associate names with memory locations and register numbers.

MMIXAL can readily be comprehended if we consider first a simple example. The following code is part of a larger program; it is a subroutine to find the maximum of n elements $X[1], \dots, X[n]$, according to Algorithm 1.2.10M.

Program M (*Find the maximum*). Initially n is in register \$0, and the address of $X[0]$ is in register x0, a global register defined elsewhere.

Assembled code	Line no.	LABEL	OP	EXPR	Times	Remarks
	01	j	IS	\$0		j
	02	m	IS	\$1		m
	03	kk	IS	\$2		$8k$
	04	xk	IS	\$3		$X[k]$
	05	t	IS	\$255		Temp storage
	06		LOC	#100		
#100: #39 02 00 03	07	Maximum	SL	kk,\$0,3	1	<i>M1. Initialize.</i> $k \leftarrow n, j \leftarrow n$.
#104: #8c 01 fe 02	08		LDO	m,x0,kk	1	$m \leftarrow X[n]$.
#108: #f0 00 00 06	09		JMP	DecrK	1	To M2 with $k \leftarrow n - 1$.

#10c: #8c 03 fe 02	10	Loop	LDO	xk,x0,kk	$n - 1$	<u>M3. Compare.</u>
#110: #30 ff 03 01	11		CMP	t,xk,m	$n - 1$	$t \leftarrow [X[k] > m] - [X[k] < m]$.
#114: #5c ff 00 03	12		PBNP	t,DecrK	$n - 1$	To M5 if $X[k] \leq m$.
#118: #c1 01 03 00	13	ChangeM	SET	m,xk	A	<u>M4. Change m. $m \leftarrow X[k]$.</u>
#11c: #3d 00 02 03	14		SR	j,kk,3	A	$j \leftarrow k$.
#120: #25 02 02 08	15	DecrK	SUB	kk,kk,8	n	<u>M5. Decrease k. $k \leftarrow k - 1$</u> .
#124: #55 00 ff fa	16		PBP	kk,Loop	n	<u>M2. All tested? To M3 if $k > 0$</u> .
#128: #f8 02 00 00	17		POP	2,0	1	Return to main program. ■

This program is an example of several things simultaneously:

- a) The columns headed “LABEL”, “OP”, and “EXPR” are of principal interest; they contain a program in the MMIXAL symbolic machine language, and we shall explain the details of this program below.
- b) The column headed “Assembled code” shows the actual numeric machine language that corresponds to the MMIXAL program. MMIXAL has been designed so that any MMIXAL program can easily be translated into numeric machine language; the translation is usually carried out by another computer program called an *assembly program* or *assembler*. Thus, programmers can do all of their machine language programming in MMIXAL, never bothering to determine the equivalent numeric codes by hand. Virtually all MMIX programs in this book are written in MMIXAL.
- c) The column headed “Line no.” is not an essential part of the MMIXAL program; it is merely included with MMIXAL examples in this book so that we can readily refer to parts of the program.
- d) The column headed “Remarks” gives explanatory information about the program, and it is cross-referenced to the steps of Algorithm 1.2.10M. The reader should compare that algorithm (page 96) with the program above. Notice that a little “programmer’s license” was used during the transcription into MMIX code; for example, step M2 has been put last.
- e) The column headed “Times” will be instructive in many of the MMIX programs we will be studying in this book; it represents the *profile*, the number of times the instruction on that line will be executed during the course of the program. Thus, line 10 will be performed $n - 1$ times, etc. From this information we can determine the length of time required to perform the subroutine; it is $n\mu + (5n + 4A + 5)v$, where A is the quantity that was analyzed carefully in Section 1.2.10. (The PBNP instruction costs $(n - 1 + 2A)v$.)

Now let’s discuss the MMIXAL part of Program M. Line 01, ‘j IS \$0’, says that symbol j stands for register \$0; lines 02–05 are similar. The effect of lines 01 and 03 can be seen on line 14, where the numeric equivalent of the instruction ‘SR j, kk, 3’ appears as #3d 00 02 03, that is, ‘SR \$0, \$2, 3’.

Line 06 says that the locations for succeeding lines should be chosen sequentially, beginning with #100. Therefore the symbol Maximum that appears in the label field of line 07 becomes equivalent to the number #100; the symbol Loop in line 10 is three tetrabytes further along, so it is equivalent to #10c.

On lines 07 through 17 the OP field contains the symbolic names of MMIX instructions: SL, LDO, etc. But the symbolic names IS and LOC, found in

the OP column of lines 01–06, are somewhat different; IS and LOC are called *pseudo-operations*, because they are operators of MMIXAL but not operators of MMIX. Pseudo-operations provide special information about a symbolic program, without being instructions of the program itself. Thus the line ‘j IS \$0’ only talks *about* Program M; it does not signify that any variable is to be set equal to the contents of register \$0 when the program is run. Notice that no instructions are assembled for lines 01–06.

Line 07 is a “shift left” instruction that sets $k \leftarrow n$ by setting kk $\leftarrow 8n$. This program works with the value of $8k$, not k , because $8k$ is needed for octabyte addresses in lines 08 and 10.

Line 09 jumps the control to line 15. The assembler, knowing that this JMP instruction is in location #108 and that DecrK is equivalent to #120, computes the relative offset $(\#120 - \#108)/4 = 6$. Similar relative addresses are computed for the branch commands in lines 12 and 16.

The rest of the symbolic code is self-explanatory. As mentioned earlier, Program M is intended to be part of a larger program; elsewhere the sequence

```
SET    $2,100
PUSHJ $1,Maximum
STO   $1,Max
```

would, for example, jump to Program M with n set to 100. Program M would then find the largest of the elements $X[1], \dots, X[100]$ and would return to the instruction ‘STO \$1,Max’ with the maximum value in \$1 and with its position, j , in \$2. (See exercise 3.)

Let’s look now at a program that is *complete*, not merely a subroutine. If the following program is named Hello, it will print out the famous message ‘Hello, world’ and stop.

Program H (*Hail the world*).

Assembled code	Line	LABEL	OP	EXPR	Remarks
	01	argv	IS	\$1	The argument vector
	02		LOC	#100	
#100: #8f ff 01 00	03	Main	LDOU	\$255, argv, 0	\$255 \leftarrow address of program name.
#104: #00 00 07 01	04		TRAP	0, Fputs, StdOut	Print that name.
#108: #f4 ff 00 03	05		GETA	\$255, String	\$255 \leftarrow address of “, world”.
#10c: #00 00 07 01	06		TRAP	0, Fputs, StdOut	Print that string.
#110: #00 00 00 00	07		TRAP	0, Halt, 0	Stop.
#114: #2c 20 77 6f	08	String	BYTE	“, world”, #a, 0	String of characters
#118: #72 6c 64 0a	09				with newline
#11c: #00	10				and terminator

Readers who have access to an MMIX assembler and simulator should take a moment to prepare a short computer file containing the LABEL OP EXPR portions of Program H before reading further. Name the file ‘Hello.mms’ and assemble it by saying, for example, ‘mmixal Hello.mms’. (The assembler will produce a file called ‘Hello.mmo’; the suffix .mms means “MMIX symbolic” and .mmo means “MMIX object.”) Now invoke the simulator by saying ‘mmix Hello’.

The MMIX simulator implements some of the simplest features of a hypothetical operating system called NNX. If an object file called, say, `foo.mmo` is present, NNX will launch it when a command line such as

(1)

is given. You can obtain the corresponding behavior by invoking the simulator with the command line ‘`mmix <options> foo bar xyzzy`’, where `<options>` is a sequence of zero or more special requests. For example, option `-P` will print a profile of the program after it has halted.

An MMIX program always begins at symbolic location `Main`. At that time register `$0` contains the number of *command line arguments*, namely the number of words on the command line. Register `$1` contains the memory address of the first such argument, which is always the name of the program. The operating system has placed all of the arguments into consecutive octabytes, starting at the address in `$1` and ending with an octabyte of all zeros. Each argument is represented as a *string*, meaning that it is the address in memory of a sequence of zero or more nonzero bytes followed by a byte that is zero; the nonzero bytes are the *characters* of the string.

For example, the command line (1) would cause `$0` to be initially 3, and we might have

<code>\$1 = #4000000000000008</code>	Pointer to the first string
<code>M₈[#4000000000000008] = #4000000000000028</code>	First argument, the string "foo"
<code>M₈[#4000000000000010] = #4000000000000030</code>	Second argument, the string "bar"
<code>M₈[#4000000000000018] = #4000000000000038</code>	Third argument, the string "xyzzy"
<code>M₈[#4000000000000020] = #0000000000000000</code>	Null pointer after the last argument
<code>M₈[#4000000000000028] = #666f6f0000000000</code>	'f', 'o', 'o', 0, 0, 0, 0, 0
<code>M₈[#4000000000000030] = #6261720000000000</code>	'b', 'a', 'r', 0, 0, 0, 0, 0
<code>M₈[#4000000000000038] = #78797a7a79000000</code>	'x', 'y', 'z', 'z', 'y', 0, 0, 0

NNX sets up each argument string so that its characters begin at an octabyte boundary; strings in general can, however, start anywhere within an octabyte.

The first instruction of Program H, in line 03, puts the string pointer `M8[$1]` into register `$255`; this string is the program name ‘Hello’. Line 04 is a special TRAP instruction, which asks the operating system to put string `$255` into the *standard output* file. Similarly, lines 05 and 06 ask NNX to contribute ‘, world’ and a newline character to the standard output. The symbol `Fputs` is predefined to equal 7, and the symbol `StdOut` is predefined to equal 1. Line 07, ‘`TRAP 0, Halt, 0`’, is the normal way to terminate a program. We will discuss all such special TRAP commands at the end of this section.

The characters of the string output by lines 05 and 06 are generated by the `BYTE` command in line 08. `BYTE` is a pseudo-operation of MMIXAL, not an operation of MMIX; but `BYTE` is different from pseudo-ops like `IS` and `LOC`, because it does assemble data into memory. In general, `BYTE` assembles a sequence of expressions into one-byte constants. The construction “, world” in line 08 is MMIXAL’s shorthand for the list

‘, ’, ‘ ’, ‘w’, ‘o’, ‘r’, ‘l’, ‘d’

of seven one-character constants. The constant `#a` on line 08 is the ASCII *newline* character, which causes a new line to begin when it appears in a file being printed. The final ‘,0’ on line 08 terminates the string. Thus line 08 is a list of nine expressions, and it leads to the nine bytes shown at the left of lines 08–10.

Our third example introduces a few more features of the assembly language. The object is to compute and print a table of the first 500 prime numbers, with 10 columns of 50 numbers each. The table should appear as follows, when the standard output of our program is listed as a text file:

```
First Five Hundred Primes
0002 0233 0547 0877 1229 1597 1993 2371 2749 3187
0003 0239 0557 0881 1231 1601 1997 2377 2753 3191
0005 0241 0563 0883 1237 1607 1999 2381 2767 3203
:
:
0229 0541 0863 1223 1583 1987 2357 2741 3181 3571
```

We will use the following method.

Algorithm P (*Print table of 500 primes*). This algorithm has two distinct parts: Steps P1–P8 prepare an internal table of 500 primes, and steps P9–P11 print the answer in the form shown above.

- P1. [Start table.] Set `PRIME[1] ← 2`, $n \leftarrow 3$, $j \leftarrow 1$. (In this program, n runs through the odd numbers that are candidates for primes; j keeps track of how many primes have been found so far.)
- P2. [n is prime.] Set $j \leftarrow j + 1$, `PRIME[j] ← n`.
- P3. [500 found?] If $j = 500$, go to step P9.
- P4. [Advance n .] Set $n \leftarrow n + 2$.
- P5. [$k \leftarrow 2$.] Set $k \leftarrow 2$. (`PRIME[k]` will run through n 's possible prime divisors.)
- P6. [`PRIME[k] \nmid n?`] Divide n by `PRIME[k]`; let q be the quotient and r the remainder. If $r = 0$ (hence n is not prime), go to P4.
- P7. [`PRIME[k]` large?] If $q \leq \text{PRIME}[k]$, go to P2. (In such a case, n must be prime; the proof of this fact is interesting and a little unusual—see exercise 11.)
- P8. [Advance k .] Increase k by 1, and go to P6.
- P9. [Print title.] Now we are ready to print the table. Output the title line and set $m \leftarrow 1$.
- P10. [Print line.] Output a line that contains `PRIME[m]`, `PRIME[50 + m]`, ..., `PRIME[450 + m]` in the proper format.
- P11. [500 printed?] Increase m by 1. If $m \leq 50$, return to P10; otherwise the algorithm terminates. ■

Program P (*Print table of 500 primes*). This program has deliberately been written in a slightly clumsy fashion in order to illustrate most of the features of MMIXAL in a single program.

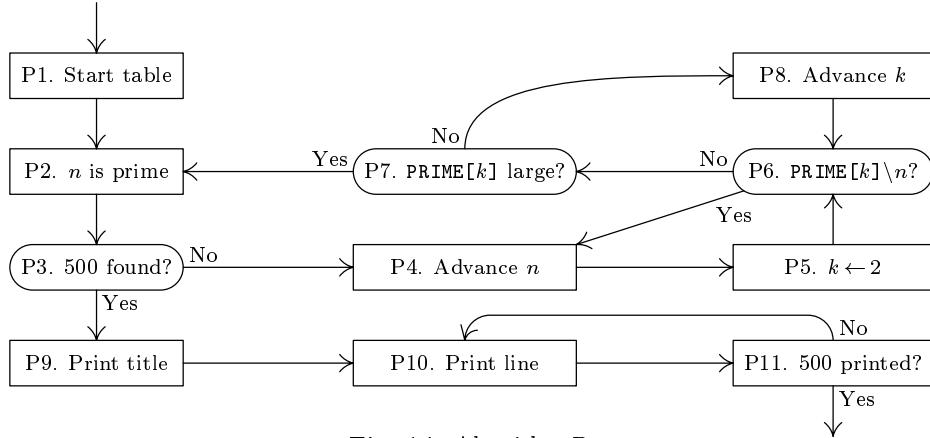


Fig. 14. Algorithm P.

01	% Example program ... Table of primes	
02	L IS 500	The number of primes to find
03	t IS \$255	Temporary storage
04	n GREG 0	Prime candidate
05	q GREG 0	Quotient
06	r GREG 0	Remainder
07	jj GREG 0	Index for PRIME[j]
08	kk GREG 0	Index for PRIME[k]
09	pk GREG 0	Value of PRIME[k]
10	mm IS kk	Index for output lines
11	LOC Data_Segment	
12	PRIME1 WYDE 2	PRIME[1] = 2
13	LOC PRIME1+2*L	
14	p top GREG @	Address of PRIME[501]
15	j0 GREG PRIME1+2-@	Initial value of jj
16	BUF OCTA 0	Place to form decimal string
17		
18	LOC #100	
19	Main SET n,3	<u>P1. Start table.</u> n ← 3.
20	SET jj,j0	j ← 1.
21	2H STWU n,p top,jj	<u>P2. n is prime.</u> PRIME[j+1] ← n.
22	INCL jj,2	j ← j + 1.
23	3H BZ jj,2F	<u>P3. 500 found?</u>
24	4H INCL n,2	<u>P4. Advance n.</u>
25	5H SET kk,j0	<u>P5. k ← 2.</u>
26	6H LDWU pk,p top,kk	<u>P6. PRIME[k]\n?</u>
27	DIV q,n,pk	q ← ⌊n/PRIME[k]⌋.
28	GET r,rR	r ← n mod PRIME[k].
29	BZ r,4B	To P4 if r = 0.
30	7H CMP t,q,pk	<u>P7. PRIME[k] large?</u>
31	BNP t,2B	To P2 if q ≤ PRIME[k].
32	8H INCL kk,2	<u>P8. Advance k.</u> k ← k + 1.
33	JMP 6B	To P6.

34	GREG	@	Base address
35	Title	BYTE "First Five Hundred Primes"	
36	NewLn	BYTE #a,0	Newline and string terminator
37	Blanks	BYTE " ",0	String of three blanks
38	2H	LDA t,Title	<u>P9. Print title.</u>
39		TRAP 0,Fputs,StdOut	
40		NEG mm,2	Initialize m .
41	3H	ADD mm,mm,j0	<u>P10. Print line.</u>
42		LDA t,Blanks	Output " ".
43		TRAP 0,Fputs,StdOut	
44	2H	LDWU pk,ptop,mm	$pk \leftarrow$ prime to be printed.
45	0H	GREG #2030303030000000	" 0000",0,0,0
46		STOU 0B,BUF	Prepare buffer for decimal conversion.
47		LDA t,BUF+4	$t \leftarrow$ position of units digit.
48	1H	DIV pk,pk,10	$pk \leftarrow \lfloor pk/10 \rfloor$.
49		GET r,rR	$r \leftarrow$ next digit.
50		INCL r,'0'	$r \leftarrow$ ASCII digit r .
51		STBU r,t,0	Store r in the buffer.
52		SUB t,t,1	Move one byte to the left.
53		PBNZ pk,1B	Repeat on remaining digits.
54		LDA t,BUF	Output " " and four digits.
55		TRAP 0,Fputs,StdOut	
56		INCL mm,2*L/10	Advance by 50 wydes.
57		PBN mm,2B	
58		LDA t,NewLn	Output a newline.
59		TRAP 0,Fputs,StdOut	
60		CMP t,mm,2*(L/10-1)	<u>P11. 500 printed?</u>
61		PBNZ t,3B	To P10 if not done.
62		TRAP 0,Halt,0	■

The following points of interest should be noted about this program:

1. Line 01 begins with a percent sign and line 17 is blank. Such “comment” lines are merely explanatory; they have no effect on the assembled program.

Each non-comment line has three fields called **LABEL**, **OP**, and **EXPR**, separated by spaces. The **EXPR** field contains one or more symbolic expressions separated by commas. Comments may follow the **EXPR** field.

2. As in Program M, the pseudo-operation **IS** sets the equivalent of a symbol. For example, in line 02 the equivalent of **L** is set to 500, which is the number of primes to be computed. Notice that in line 03, the equivalent of **t** is set to \$255, a *register number*, while **L**’s equivalent was 500, a *pure number*. Some symbols have register number equivalents, ranging from \$0 to \$255; others have pure equivalents, which are octabytes. We will generally use symbolic names that begin with a lowercase letter to denote registers, and names that begin with an uppercase letter to denote pure values, although MMIXAL does not enforce this convention.

3. The pseudo-op **GREG** on line 04 allocates a *global register*. Register \$255 is always global; the first **GREG** causes \$254 to be global, and the next **GREG** does

the same for \$253, etc. Lines 04–09 therefore allocate six global registers, and they cause the symbols **n**, **q**, **r**, **jj**, **kk**, **pk** to be respectively equivalent to \$254, \$253, \$252, \$251, \$250, \$249. Line 10 makes **mm** equivalent to \$250.

If the EXPR field of a GREG definition is zero, as it is on lines 04–09, the global register is assumed to have a dynamically varying value when the program is run. But if a nonzero expression is given, as on lines 14, 15, 34, and 45, the global register is assumed to be constant throughout a program's execution. MMIXAL uses such global registers as *base addresses* when subsequent instructions refer to memory. For example, consider the instruction 'LDA t,BUF+4' in line 47. MMIXAL is able to discover that global register ptop holds the address of BUF; therefore 'LDA t,BUF+4' can be assembled as 'LDA t,ptop,4'. Similarly, the LDA instructions on lines 38, 42, and 58 make use of the nameless base address introduced by the instruction 'GREG @' on line 34. (Recall from Section 1.3.1' that @ denotes the current location.)

4. A good assembly language should mimic the way a programmer *thinks* about machine programs. One example of this philosophy is the automatic allocation of global registers and base addresses. Another example is the idea of *local symbols* such as the symbol 2H, which appears in the label field of lines 21, 38, and 44.

Local symbols are special symbols whose equivalents can be *redefined* as many times as desired. A global symbol like PRIME1 has but one significance throughout a program, and if it were to appear in the label field of more than one line an error would be indicated by the assembler. But local symbols have a different nature; we write, for example, 2H ("2 here") in the LABEL field, and 2F ("2 forward") or 2B ("2 backward") in the EXPR field of an MMIXAL line:

2B means the closest *previous* label 2H;
2F means the closest *following* label 2H.

Thus the 2F in line 23 refers to line 38; the 2B in line 31 refers back to line 21; and the 2B in line 57 refers to line 44. The symbols 2F and 2B never refer to their *own* line. For example, the MMIXAL instructions

2H	IS	\$10
2H	BZ	2B,2F
2H	IS	2B-4

are virtually equivalent to the single instruction

BZ \$10,@-4.

The symbols 2F and 2B should never be used in the LABEL field; the symbol 2H should never be used in the EXPR field. If 2B occurs before any appearance of 2H, it denotes zero. There are ten local symbols, which can be obtained by replacing '2' in these examples by any digit from 0 to 9.

The idea of local symbols was introduced by M. E. Conway in 1958, in connection with an assembly program for the UNIVAC I. Local symbols free us from the obligation to choose a symbolic name when we merely want to refer to

an instruction a few lines away. There often is no appropriate name for nearby locations, so programmers have tended to introduce meaningless symbols like X_1 , X_2 , X_3 , etc., with the potential danger of duplication.

5. The reference to `Data_Segment` on line 11 introduces another new idea. In most embodiments of MMIX, the 2^{64} -byte virtual address space is broken into two parts, called *user space* (addresses `#0000000000000000 .. #7fffffffffffffff`) and *kernel space* (addresses `#8000000000000000 .. #fffffffffffffff`). The “negative” addresses of kernel space are reserved for the operating system.

User space is further subdivided into four segments of 2^{61} bytes each. First comes the *text segment*; the user’s program generally resides here. Then comes the *data segment*, beginning at virtual address `#2000000000000000`; this is for variables whose memory locations are allocated once and for all by the assembler, and for other variables allocated by the user without the help of the system library. Next is the *pool segment*, beginning at `#4000000000000000`; command line arguments and other dynamically allocated data go here. Finally the *stack segment*, which starts at `#6000000000000000`, is used by the MMIX hardware to maintain the register stack governed by `PUSH`, `POP`, `SAVE`, and `UNSAVE`. Three symbols,

```
Data_Segment = #2000000000000000,
Pool_Segment = #4000000000000000,
Stack_Segment = #6000000000000000,
```

are predefined for convenience in MMIXAL. Nothing should be assembled into the pool segment or the stack segment, although a program may refer to data found there. References to addresses near the beginning of a segment might be more efficient than references to addresses that come near the end; for example, MMIX might not be able to access the last byte of the text segment, $M[#1fffffffffffff]$, as fast as it can read the first byte of the data segment.

Our programs for MMIX will always consider the text segment to be *read-only*: Everything in memory locations less than `#2000000000000000` will remain constant once a program has been assembled and loaded. Therefore Program P puts the prime table and the output buffer into the data segment.

6. The text and data segments are entirely zero at the beginning of a program, except for instructions and data that have been loaded in accordance with the MMIXAL specification of the program. If two or more bytes of data are destined for the same cell of memory, the loader will fill that cell with their bitwise exclusive-or.

7. The symbolic expression ‘`PRIME1+2*L`’ on line 13 indicates that MMIXAL has the ability to do arithmetic on octabytes. See also the more elaborate example ‘`2*(L/10-1)`’ on line 60.

8. As a final note about Program P, we can observe that its instructions have been organized so that registers are counted towards zero, and tested against zero, whenever possible. For example, register jj holds a quantity that is related to the positive variable j of Algorithm P, but jj is normally negative; this change

makes it easy for the machine to decide when j has reached 500 (line 23). Lines 40–61 are particularly noteworthy in this regard, although perhaps a bit tricky. The binary-to-decimal conversion routine in lines 45–55, based on division by 10, is simple but not the fastest possible. More efficient methods are discussed in Section 4.4.

It may be of interest to note a few of the statistics observed when Program P was actually run. The division instruction in line 27 was executed 9538 times. The total time to perform steps P1–P8 (lines 19–33) was $10036\mu + 641543v$; steps P9–P11 cost an additional $2804\mu + 124559v$, not counting the time taken by the operating system to handle TRAP requests.

Language summary. Now that we have seen three examples of what can be done in **MMIXAL**, it is time to discuss the rules more carefully, observing in particular the things that *cannot* be done. The following comparatively few rules define the language.

1. A *symbol* is a string of letters and/or digits, beginning with a letter. The underscore character ‘_’ is regarded as a letter, for purposes of this definition, and so are all Unicode characters whose code value exceeds 126. *Examples:* PRIME1, Data_Segment, Main, __, pâté.

The special constructions *dH*, *dF*, and *dB*, where *d* is a single digit, are effectively replaced by unique symbols according to the “local symbol” convention explained above.

2. A *constant* is either

- a) a *decimal constant*, consisting of one or more decimal digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, representing an unsigned octabyte in radix 10 notation; or
- b) a *hexadecimal constant*, consisting of a hash mark # followed by one or more hexadecimal digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F}, representing an unsigned octabyte in radix 16 notation; or
- c) a *character constant*, consisting of a quote character ‘ followed by any character other than newline, followed by another quote ’; this represents the ASCII or Unicode value of the quoted character.

Examples: 65, #41, 'A', 39, #27, '', 31639, #7B97, '算'.

A *string constant* is a double-quote character " followed by one or more characters other than newline or double-quote, followed by another double-quote ". This construction is equivalent to a sequence of character constants for the individual characters, separated by commas.

3. Each appearance of a symbol in an **MMIXAL** program is said to be either a “defined symbol” or a “future reference.” A *defined symbol* is a symbol that has appeared in the **LABEL** field of a preceding line of this **MMIXAL** program. A *future reference* is a symbol that has not yet been defined in this way.

A few symbols, like **rR** and **ROUND_NEAR** and **V_BIT** and **W_Handler** and **Fputs**, are predefined because they refer to constants associated with the **MMIX**

hardware or with its rudimentary operating system. Such symbols can be re-defined, because MMIXAL does not assume that every programmer knows all their names. But no symbol should appear as a label more than once.

Every defined symbol has an equivalent value, which is either *pure* (an unsigned octabyte) or a *register number* (\$0 or \$1 or ... or \$255).

4. A *primary* is either

- a) a symbol; or
- b) a constant; or
- c) the character @, denoting the current location; or
- d) an expression enclosed in parentheses; or
- e) a unary operator followed by a primary.

The unary operators are + (affirmation, which does nothing), - (negation, which subtracts from zero), ~ (complementation, which changes all 64 bits), and \$ (registerization, which converts a pure value to a register number).

5. A *term* is a sequence of one or more primaries separated by strong binary operators; an *expression* is a sequence of one or more terms separated by weak binary operators. The *strong binary operators* are * (multiplication), / (division), // (fractional division), % (remainder), << (left shift), >> (right shift), and & (bitwise and). The *weak binary operators* are + (addition), - (subtraction), | (bitwise or), and ^ (bitwise exclusive-or). These operations act on unsigned octabytes; $x//y$ denotes $\lfloor 2^{64}x/y \rfloor$ if $x < y$, and it is undefined if $x \geq y$. Binary operators of the same strength are performed from left to right; thus $a/b/c$ is $(a/b)/c$ and $a-b+c$ is $(a-b)+c$.

Example: #ab<<32+k&~(k-1) is an expression, the sum of terms #ab<<32 and k&~(k-1). The latter term is the bitwise and of primaries k and ~(k-1). The latter primary is the complement of (k-1), a parenthesized expression that is the difference of two terms k and 1. The term 1 is also a primary, and also a constant, in fact it is a decimal constant. If symbol k is equivalent to #cdef00, say, the entire expression #ab<<32+k&~(k-1) is equivalent to #ab00000100.

Binary operations are allowed only on pure numbers, except in cases like \$1+2 = \$3 and \$3-\$1 = 2. Future references cannot be combined with anything else; an expression like 2F+1 is always illegal, because 2F never corresponds to a defined symbol.

6. An *instruction* consists of three fields:

- a) the **LABEL** field, which is either blank or a symbol;
- b) the **OP** field, which is either an **MMIX** opcode or an **MMIXAL** pseudo-op;
- c) the **EXPR** field, which is a list of one or more expressions separated by commas. The **EXPR** field can also be blank, in which case it is equivalent to the single expression 0.

7. Assembly of an instruction takes place in three steps:

- a) The current location @ is aligned, if necessary, by increasing it to the next multiple of
 - 8, if OP is `OCTA`;
 - 4, if OP is `TETRA` or an MMIX opcode;
 - 2, if OP is `WYDE`.

b) The symbol in `LABEL`, if present, is defined to be @ , unless $\text{OP} = \text{IS}$ or $\text{OP} = \text{GREG}$.

- c) If OP is a pseudo-operation, see rule 8. Otherwise OP is an MMIX instruction; the `OP` and `EXPR` fields define a tetrabyte as explained in Section 1.3.1', and @ advances by 4. Some MMIX opcodes have three operands in the `EXPR` field, others have two, and others have only one.

If OP is `ADD`, say, MMIXAL will expect three operands, and will check that the first and second operands are register numbers. If the third operand is pure, MMIXAL will change the opcode from #20 ("add") to #21 ("add immediate"), and will check that the immediate value is less than 256.

If OP is `SETH`, say, MMIXAL will expect two operands. The first operand should be a register number; the second should be a pure value less than 65536.

An OP like `BNZ` takes two operands: a register and a pure number. The pure number should be expressible as a relative address; in other words, its value should be expressible as $\text{@} + 4k$ where $-65536 \leq k < 65536$.

Any OP that refers to memory, like `LDB` or `G0`, has a two-operand form `$X,A` as well as the three-operand forms `$X,$Y,$Z` or `$X,$Y,Z`. The two-operand option can be used when the memory address A is expressible as the sum $\text{$Y} + \text{$Z}$ of a base address and a one-byte value; see rule 8(b).

8. MMIXAL includes the following pseudo-operations.

- a) $\text{OP} = \text{IS}$: The `EXPR` should be a single expression; the symbol in `LABEL`, if present, is made equivalent to the value of this expression.
- b) $\text{OP} = \text{GREG}$: The `EXPR` should be a single expression with a pure equivalent, x . The symbol in `LABEL`, if present, is made equivalent to the largest previously unallocated global register number, and this global register will contain x when the program begins. If $x \neq 0$, the value of x is considered to be a *base address*, and the program should not change that global register.
- c) $\text{OP} = \text{LOC}$: The `EXPR` should be a single expression with a pure equivalent, x . The value of @ is set to x . For example, the instruction '`T LOC @+1000`' defines symbol `T` to be the address of the first of a sequence of 1000 bytes, and advances @ to the byte following that sequence.
- d) $\text{OP} = \text{BYTE}, \text{WYDE}, \text{TETRA}$, or `OCTA`: The `EXPR` field should be a list of pure expressions that each fit in 1, 2, 4, or 8 bytes, respectively.

9. MMIXAL restricts future references so that the assembly process can work quickly in one pass over the program. A future reference is permitted only

- a) in a relative address: as the operand of `JMP`, or as the second operand of a branch, probable branch, `PUSHJ`, or `GETA`; or
- b) in an expression assembled by `OCTA`.

```
% Example program ... Table of primes
L IS 500      The number of primes to find
t IS $255     Temporary storage
n GREG        ;; Prime candidate
q GREG /* Quotient */
r GREG // Remainder
jj GREG 0     Index for PRIME[j]
:
PBN mm,2B
LDA t,NewLn; TRAP 0,Fputs,StdOut
CMP t,mm,2*(L/10-1) ; PBNZ t,3B;      TRAP 0,Halt,0
```

Fig. 15. Program P as a computer file: The assembler tolerates many formats.

MMIXAL also has a few additional features relevant to system programming that do not concern us here. Complete details of the full language appear in the *MMIXware* document, together with the complete logic of a working assembler.

A free format can be used when presenting an MMIXAL program to the assembler (see Fig. 15). The **LABEL** field starts at the beginning of a line and continues up to the first blank space. The next nonblank character begins the **OP** field, which continues to the next blank, etc. The whole line is a comment if the first nonblank character is not a letter or digit; otherwise comments start after the **EXPR** field. Notice that the **GREG** definitions for **n**, **q**, and **r** in Fig. 15 have a blank **EXPR** field (which is equivalent to the single expression ‘0’); therefore the comments on those lines need to be introduced by some sort of special delimiter. But no such delimiter is necessary on the **GREG** line for **jj**, because an explicit **EXPR** of 0 appears there.

The final lines of Fig. 15 illustrate the fact that two or more instructions can be placed on a single line of input to the assembler, if they are separated by semicolons. If an instruction following a semicolon has a nonblank label, the label must immediately follow the ‘;’.

A consistent format would obviously be better than the hodgepodge of different styles shown in Fig. 15, because computer files are easier to read when they aren’t so chaotic. But the assembler itself is very forgiving; it doesn’t mind occasional sloppiness.

Primitive input and output. Let us conclude this section by discussing the special **TRAP** operations supported by the MMIX simulator. These operations provide basic input and output functions on which facilities at a much higher level could be built. A two-instruction sequence of the form

$$\text{SET } \$255, \langle\text{arg}\rangle; \text{ TRAP } 0, \langle\text{function}\rangle, \langle\text{handle}\rangle \quad (2)$$

is usually used to invoke such a function, where **⟨arg⟩** points to a parameter and **⟨handle⟩** identifies the relevant file. For example, Program H uses

`GETA $255,String; TRAP 0,Fputs,StdOut`

to put a string into the standard output file, and Program P is similar.

After the TRAP has been serviced by the operating system, register \$255 will contain a return value. In each case *this value will be negative if and only if an error occurred*. Programs H and P do not check for file errors, because they assume that the correctness or incorrectness of the standard output will speak for itself; but error detection and error recovery are usually important in well-written programs.

- **Fopen(handle, name, mode).** Each of the ten primitive input/output traps applies to a *handle*, which is a one-byte integer. Fopen associates *handle* with an external file whose name is the string *name*, and prepares to do input and/or output on that file. The third parameter, *mode*, must be one of the values **TextRead**, **TextWrite**, **BinaryRead**, **BinaryWrite**, or **BinaryReadWrite**, all of which are predefined in MMIXAL. In the three ...Write modes, any previous file contents are discarded. The value returned is 0 if the handle was successfully opened, otherwise -1.

The calling sequence for Fopen is

LDA \$255,Arg; TRAP 0,Fopen,(handle) (3)

where Arg is a two-octabyte sequence

Arg OCTA <name>,(mode) (4)

that has been placed elsewhere in memory. For example, to call the function **Fopen(5, "foo", BinaryWrite)** in an MMIXAL program, we could put

```
Arg OCTA 1F,BinaryWrite
1H     BYTE  "foo",0
```

into, say, the data segment, and then give the instructions

LDA \$255,Arg; TRAP 0,Fopen,5.

This would open handle 5 for writing a new file of binary output,* to be named "foo".

Three handles are already open at the beginning of each program: The standard input file **StdIn** (handle 0) has mode **TextRead**; the standard output file **StdOut** (handle 1) has mode **TextWrite**; the standard error file **StdErr** (handle 2) also has mode **TextWrite**.

- **Fclose(handle).** If *handle* has been opened, Fclose causes it to be closed, hence no longer associated with any file. Again the result is 0 if successful, or -1 if the file was already closed or unclosable. The calling sequence is simply

TRAP 0,Fclose,(handle) (5)

because there is no need to put anything in \$255.

* Different computer systems have different notions of what constitutes a text file and what constitutes a binary file. Each MMIX simulator adopts the conventions of the operating system on which it resides.

- **Fread**(*handle, buffer, size*). The file handle should have been opened with mode `TextRead`, `BinaryRead`, or `BinaryReadWrite`. The next *size* bytes are read from the file into MMIX's memory starting at address *buffer*. The value $n - size$ is returned, where n is the number of bytes successfully read and stored, or $-1 - size$ if an error occurred. The calling sequence is

LDA \$255,Arg; TRAP 0,Fread,<handle> (6)

with two octabytes for the other arguments

Arg OCTA <buffer>,<size> (7)

as in (3) and (4).

- **Fgets**(*handle, buffer, size*). The file handle should have been opened with mode `TextRead`, `BinaryRead`, or `BinaryReadWrite`. One-byte characters are read into MMIX's memory starting at address *buffer*, until either *size*–1 characters have been read and stored or a newline character has been read and stored; the next byte in memory is then set to zero. If an error or end of file occurs before reading is complete, the memory contents are undefined and the value -1 is returned; otherwise the number of characters successfully read and stored is returned. The calling sequence is the same as (6) and (7), except of course that **Fgets** replaces **Fread** in (6).

- **Fgetws**(*handle, buffer, size*). This command is the same as **Fgets**, except that it applies to wyde characters instead of one-byte characters. Up to *size*–1 wyde characters are read; a wyde newline is `#000a`.

- **Fwrite**(*handle, buffer, size*). The file handle should have been opened with one of the modes `TextWrite`, `BinaryWrite`, or `BinaryReadWrite`. The next *size* bytes are written from MMIX's memory starting at address *buffer*. The value $n - size$ is returned, where n is the number of bytes successfully written. The calling sequence is analogous to (6) and (7).

- **Fputs**(*handle, string*). The file handle should have been opened with mode `TextWrite`, `BinaryWrite`, or `BinaryReadWrite`. One-byte characters are written from MMIX's memory to the file, starting at address *string*, up to but not including the first byte equal to zero. The number of bytes written is returned, or -1 on error. The calling sequence is

SET \$255,<string>; TRAP 0,Fputs,<handle>. (8)

- **Fputws**(*handle, string*). This command is the same as **Fputs**, except that it applies to wyde characters instead of one-byte characters.

- **Fseek**(*handle, offset*). The file handle should have been opened with mode `BinaryRead`, `BinaryWrite`, or `BinaryReadWrite`. This operation causes the next input or output operation to begin at *offset* bytes from the beginning of the file, if $offset \geq 0$, or at $-offset - 1$ bytes before the end of the file, if $offset < 0$. (For example, $offset = 0$ “rewinds” the file to its very beginning; $offset = -1$

moves forward all the way to the end.) The result is 0 if successful, or -1 if the stated positioning could not be done. The calling sequence is

SET \$255,⟨offset⟩; TRAP 0,Fseek,⟨handle⟩. (9)

An `Fseek` command must be given when switching from input to output or from output to input in `BinaryReadWrite` mode.

- `Ftell(handle)`. The given file handle should have been opened with mode `BinaryRead`, `BinaryWrite`, or `BinaryReadWrite`. This operation returns the current file position, measured in bytes from the beginning, or -1 if an error has occurred. The calling sequence is simply

TRAP 0,Ftell,⟨handle⟩. (10)

Complete details about all ten of these input/output functions appear in the *MMIXware* document, together with a reference implementation. The symbols

Fopen = 1,	Fwrite = 6,	TextRead = 0,
Fclose = 2,	Fputs = 7,	TextWrite = 1,
Fread = 3,	Fputws = 8,	BinaryRead = 2,
Fgets = 4,	Fseek = 9,	BinaryWrite = 3,
Fgetws = 5,	Ftell = 10,	BinaryReadWrite = 4

(11)

are predefined in `MMIXAL`; also `Halt = 0`.

EXERCISES — First set

1. [05] (a) What is the meaning of ‘4B’ in line 29 of Program P? (b) Would the program still work if the label of line 24 were changed to ‘2H’ and the `EXPR` field of line 29 were changed to ‘r,2B’?
2. [10] Explain what happens if an `MMIXAL` program contains several instances of the line

9H IS 9B+1

and no other occurrences of `9H`.

- 3. [23] What is the effect of the following program?

```

LOC    Data_Segment
X0    IS    @
N     IS    100
x0    GREG  X0
<Insert Program M here>
Main   GETA   t,9F; TRAP 0,Fread,StdIn
        SET    $0,N<<3
1H     SR     $2,$0,3; PUSHJ $1,Maximum
        LDO    $3,x0,$0
        SL    $2,$2,3
        STO    $1,x0,$0; STO $3,x0,$2
        SUB    $0,$0,1<<3; PBNZ $0,1B
        GETA   t,9F; TRAP 0,Fwrite,StdOut
        TRAP   0,Halt,0
9H     OCTA   X0+1<<3,N<<3    ■

```

4. [10] What is the value of the constant #112233445566778899?
5. [11] What do you get from ‘BYTE 3+"pills"+6’?
- 6. [15] True or false: The single instruction TETRA ⟨expr1⟩,⟨expr2⟩ always has the same effect as the pair of instructions TETRA ⟨expr1⟩; TETRA ⟨expr2⟩.
7. [05] John H. Quick (a student) was shocked, shocked to find that the instruction GETA \$0, $\mathbb{Q}+1$ gave the same result as GETA \$0, \mathbb{Q} . Explain why he should not have been surprised.
- 8. [15] What’s a good way to align the current location \mathbb{Q} so that it is a multiple of 16, increasing it by 0..15 as necessary?
9. [10] What changes to Program P will make it print a table of 600 primes?
- 10. [25] Assemble Program P by hand. (It won’t take as long as you think.) What are the actual numerical contents of memory, corresponding to that symbolic program?
11. [HM20] (a) Show that every nonprime $n > 1$ has a divisor d with $1 < d \leq \sqrt{n}$.
 (b) Use this fact to show that n is prime if it passes the test in step P7 of Algorithm P.
12. [15] The GREG instruction on line 34 of Program P defines a base address that is used for the string constants Title, NewLn, and Blank on lines 38, 42, and 58. Suggest a way to avoid using this extra global register, without making the program run slower.
13. [20] Unicode characters make it possible to print the first 500 primes as

أول خمس ميليات للأرقام الأولية

٢١٨٧	٢٧٤٩	٢٣٧١	١٩٩٣	١٥٩٧	١٢٢٩	٠٨٧٧	٠٥٤٧	٠٢٢٣	٠٠٠٢
٢١٩١	٢٧٥٣	٢٣٧٧	١٩٩٧	١٦٠١	١٢٢١	٠٨٨١	٠٥٥٧	٠٢٢٩	٠٠٠٣
٢٢٠٣	٢٧٦٧	٢٣٨١	١٩٩٩	١٦٠٧	١٢٢٧	٠٨٨٣	٠٥٦٣	٠٢٤١	٠٠٠٥
⋮									
٢٥٧١	٣١٨١	٢٧٤١	٢٢٥٧	١٩٨٧	١٥٨٣	١٢٢٢	٠٨٦٣	٠٥٤١	٠٢٢٩

with “authentic” Arabic numerals. One simply uses wyde characters instead of bytes, translating the English title and then substituting Arabic-Indic digits #0660 –#0669 for the ASCII digits #30 –#39. (Arabic script is written from right to left, but numbers still appear with their least significant digits at the right. The bidirectional presentation rules of Unicode automatically take care of the necessary reversals when the output is formatted.) What changes to Program P will accomplish this?

- 14. [21] Change Program P so that it uses floating point arithmetic for the divisibility test in step P6. (The FREM instruction always gives an exact result.) Use \sqrt{n} instead of q in step P7. Do these changes increase or decrease the running time?
- 15. [22] What does the following program do? (Do not run it on a computer, figure it out by hand!)

```
* Mystery Program
a    GREG   '*'
b    GREG   ','
c    GREG   Data_Segment
      LOC    #100
Main NEG    $1,1,75
      SET    $2,0
2H    ADD    $3,$1,75
3H    STB    b,c,$2
      ADD    $2,$2,1
```

```

SUB    $3,$3,1
PBP    $3,3B
STB    a,c,$2
INCL   $2,1
INCL   $1,1
PBN    $1,2B
SET    $255,c; TRAP 0,Fputs,StdOut
TRAP   0,Halt,0  ■

```

16. [46] MMIXAL was designed with simplicity and efficiency in mind, so that people can easily prepare machine language programs for MMIX when those programs are relatively short. Longer programs are usually written in a higher-level language like C or Java, ignoring details at the machine level. But sometimes there is a need to write large-scale programs specifically for a particular machine, and to have precise control over each instruction. In such cases we ought to have a machine-oriented language with a much richer structure than the line-for-line approach of a traditional assembler.

Design and implement a language called PL/MMIX, which is analogous to Niklaus Wirth's PL/360 language [JACM 15 (1968), 37–74]. Your language should also incorporate the ideas of literate programming [D. E. Knuth, *Literate Programming* (1992)].

EXERCISES — Second set

The next exercises are short programming problems, representing typical computer applications and covering a wide range of techniques. Every reader is encouraged to choose a few of these problems in order to get some experience using MMIX, as well as to practice basic programming skills. If desired, these exercises may be worked concurrently as the rest of Chapter 1 is being read. The following list indicates the types of programming techniques that are involved:

- The use of switching tables for multiway decisions: exercise 17.
- Computation with two-dimensional arrays: exercises 18, 28, and 35.
- Text and string manipulation: exercises 24, 25, and 35.
- Integer and scaled decimal arithmetic: exercises 21, 27, 30, and 32.
- Elementary floating point arithmetic: exercises 27 and 32.
- The use of subroutines: exercises 23, 24, 32, 33, 34, and 35.
- List processing: exercise 29.
- Real-time control: exercise 34.
- Typographic display: exercise 35.
- Loop and pipeline optimization: exercises 23 and 26.

Whenever an exercise in this book says “write an MMIX program” or “write an MMIX subroutine,” you need only write symbolic MMIXAL code for what is asked. This code will not be complete in itself; it will merely be a fragment of a (hypothetical) complete program. No input or output need be done in a code fragment, if the data is to be supplied externally; one need write only **LABEL**, **OP**, and **EXPR** fields of MMIXAL instructions, together with appropriate remarks. The numeric machine language, line number, and “Times” columns (see Program M) are not required unless specifically requested, nor will there be a **Main** label.

On the other hand, if an exercise says “write a *complete* MMIX program,” it implies that an executable program should be written in MMIXAL, including in particular the **Main** label. Such programs should preferably be tested with the help of an MMIX assembler and simulator.

- 17. [25] Register \$0 contains the address of a tetrabyte that purportedly is a valid, unprivileged MMIX instruction. (This means that $$0 \geq 0$ and that the X, Y, and Z bytes of $M_4[$0]$ obey all restrictions imposed by the OP byte, according to the rules of Section 1.3.1'. For example, a valid instruction with opcode FIX will have $Y \leq \text{ROUND_NEAR}$; a valid instruction with opcode PUT will have $Y = 0$ and either $X < 8$ or $18 < X < 32$. The opcode LDVTS is always privileged, for use by the operating system only. But most opcodes define instructions that are valid and unprivileged for all X, Y, and Z.) Write an MMIX subroutine that checks the given tetrabyte for validity in this sense; try to make your program as efficient as possible.

Note: Inexperienced programmers tend to tackle a problem like this by writing a long series of tests on the OP byte, such as “SR op,tetra,24; CMP t,op,#18; BN t,1F; CMP t,op,#98; BN t,2F; ...”. This is *not* good practice! The best way to make multiway decisions is to prepare an auxiliary *table* containing information that encapsulates the desired logic. For example, a table of 256 octabytes, one for each opcode, could be accessed by saying “SR t,tetra,21; LDO t,Table,t”, followed perhaps by a GO instruction if many different kinds of actions need to be done. A tabular approach often makes a program dramatically faster and more flexible.

- 18. [31] Assume that a 9×8 matrix of signed one-byte elements

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{18} \\ a_{21} & a_{22} & a_{23} & \dots & a_{28} \\ \vdots & & & & \vdots \\ a_{91} & a_{92} & a_{93} & \dots & a_{98} \end{pmatrix}$$

has been stored so that a_{ij} is in location $A + 8i + j$ for some constant A . The matrix therefore appears as follows in MMIX's memory:

$$\begin{pmatrix} M[A + 9] & M[A + 10] & M[A + 11] & \dots & M[A + 16] \\ M[A + 17] & M[A + 18] & M[A + 19] & \dots & M[A + 24] \\ \vdots & & & & \vdots \\ M[A + 73] & M[A + 74] & M[A + 75] & \dots & M[A + 80] \end{pmatrix}.$$

An $m \times n$ matrix is said to have a “saddle point” if some position is the smallest value in its row and the largest value in its column. In symbols, a_{ij} is a saddle point if

$$a_{ij} = \min_{1 \leq k \leq n} a_{ik} = \max_{1 \leq k \leq m} a_{kj}.$$

Write an MMIX program that computes the location of a saddle point (if there is at least one) or zero (if there is no saddle point), and puts this value in register \$0.

19. [M29] What is the *probability* that the matrix in the preceding exercise has a saddle point, assuming that the 72 elements are distinct and assuming that all $72!$ permutations are equally likely? What is the corresponding probability if we assume instead that the elements of the matrix are zeros and ones, and that all 2^{72} such matrices are equally likely?

20. [HM42] Two solutions are given for exercise 18 (see page 102), and a third is suggested; it is not clear which of them is better. Analyze the algorithms, using each of the assumptions of exercise 19, and decide which is the better method.

21. [25] The ascending sequence of all reduced fractions between 0 and 1 that have denominators $\leq n$ is called the “Farey series of order n .” For example, the Farey series of order 7 is

$$\frac{0}{1}, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, \frac{1}{1}.$$

If we denote this series by $x_0/y_0, x_1/y_1, x_2/y_2, \dots$, exercise 22 proves that

$$\begin{aligned} x_0 &= 0, & y_0 &= 1; & x_1 &= 1, & y_1 &= n; \\ x_{k+2} &= \lfloor (y_k + n)/y_{k+1} \rfloor x_{k+1} - x_k; \\ y_{k+2} &= \lfloor (y_k + n)/y_{k+1} \rfloor y_{k+1} - y_k. \end{aligned}$$

Write an MMIX subroutine that computes the Farey series of order n , by storing the values of x_k and y_k in tetrabytes X + 4k and Y + 4k, respectively. (The total number of terms in the series is approximately $3n^2/\pi^2$; thus we may assume that $n < 2^{32}$.)

22. [M30] (a) Show that the numbers x_k and y_k defined by the recurrence in the preceding exercise satisfy the relation $x_{k+1}y_k - x_ky_{k+1} = 1$. (b) Show that the fractions x_k/y_k are indeed the Farey series of order n , using the fact proved in (a).

23. [25] Write an MMIX subroutine that sets n consecutive bytes of memory to zero, given a starting address in \$0 and an integer $n \geq 0$ in \$1. Try to make your subroutine blazingly fast, when n is large; use an MMIX pipeline simulator to obtain realistic running-time statistics.

► **24.** [30] Write an MMIX subroutine that copies a string, starting at the address in \$0, to bytes of memory starting at the address in \$1. Strings are terminated by null characters (that is, bytes equal to zero). Assume that there will be no overlap in memory between the string and its copy. Your routine should minimize the number of memory references by loading and storing eight bytes at a time when possible, so that long strings are copied efficiently. Compare your program to the trivial byte-at-a-time code

```
SUBU $1,$1,$0;1H LDBU $2,$0,0; STBU $2,$0,$1; INCL $0,1; PBNZ $2,1B
```

which takes $(2n + 2)\mu + (4n + 7)v$ to copy a string of length n .

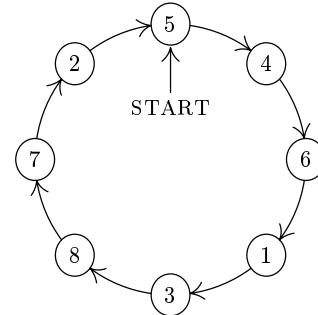
25. [26] A cryptanalyst wants to count how often each character occurs in a long string of ciphertext. Write an MMIX program that computes 255 frequency counts, one for each nonnull character; the first null byte ends the given string. Try for a solution that is efficient in terms of the “mems and oops” criteria of Table 1 in Section 1.3.1’.

► **26.** [32] Improve the solution to the previous exercise by optimizing its performance with respect to realistic configurations of the MMIX pipeline simulator.

27. [26] (*Fibonacci approximations.*) Equation 1.2.8–(15) states that the formula $F_n = \text{round}(\phi^n/\sqrt{5})$ holds for all $n \geq 0$, where ‘round’ denotes rounding to the nearest integer. (a) Write a complete MMIX program to test how well this formula behaves with respect to floating point arithmetic: Compute straightforward approximations to $\phi^n/\sqrt{5}$ for $n = 0, 1, 2, \dots$, and find the smallest n for which the approximation does not round to F_n . (b) Exercise 1.2.8–28 proves that $F_n = \text{round}(\phi F_{n-1})$ for all $n \geq 3$. Find the smallest $n \geq 3$ for which this equation fails when we compute ϕF_{n-1} approximately by *fixed point* multiplication of unsigned octabytes. (See Eq. 1.3.1’–(7).)

28. [26] A *magic square of order n* is an arrangement of the numbers 1 through n^2 in a square array in such a way that the sum of each row and column is $n(n^2 + 1)/2$, and so is the sum of the two main diagonals. Figure 16 shows a magic square of order 7.

22	47	16	41	10	35	04
05	23	48	17	42	11	29
30	06	24	49	18	36	12
13	31	07	25	43	19	37
38	14	32	01	26	44	20
21	39	08	33	02	27	45
46	15	40	09	34	03	28

Fig. 16. A magic square.**Fig. 17.** Josephus's problem, $n = 8$, $m = 4$.

The rule for generating it is easily seen: Start with 1 just below the middle square, then go down and to the right diagonally until reaching a filled square; if you run off the edge, “wrap around” by imagining an entire plane tiled with squares. When you reach a nonempty position, drop down two spaces from the most-recently-filled square and continue. This method works whenever n is odd.

Using memory allocated in a fashion like that of exercise 18, write a complete MMIX program to generate a 19×19 magic square by the method above, and to format the result in the standard output file. [This algorithm is due to Ibn al-Haytham, who was born in Basra about 965 and died in Cairo about 1040. Many other magic square constructions make good programming exercises; see W. W. Rouse Ball, *Mathematical Recreations and Essays*, revised by H. S. M. Coxeter (New York: Macmillan, 1939), Chapter 7.]

29. [30] (*The Josephus problem.*) There are n men arranged in a circle. Beginning at a particular position, we count around the circle and brutally execute every m th man; the circle closes as men die. For example, the execution order when $n = 8$ and $m = 4$ is 54613872, as shown in Fig. 17: The first man is fifth to go, the second man is fourth, etc. Write a complete MMIX program that prints out the order of execution when $n = 24$, $m = 11$. Try to design a clever algorithm that works at high speed when m and n are large (it may save your life). Reference: W. Ahrens, *Mathematische Unterhaltungen und Spiele 2* (Leipzig: Teubner, 1918), Chapter 15.

30. [31] We showed in Section 1.2.7 that the sum $1 + \frac{1}{2} + \frac{1}{3} + \dots$ becomes infinitely large. But if it is calculated with finite accuracy by a computer, the sum actually exists, in some sense, because the terms eventually get so small that they contribute nothing to the sum if added one by one. For example, suppose we calculate the sum by rounding to one decimal place; then we have $1 + 0.5 + 0.3 + 0.2 + 0.2 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.0 + \dots = 3.7$.

More precisely, let $r_n(x)$ be the number x rounded to n decimal places, rounding to an even digit in case of ties. For the purposes of this problem we can use the formula $r_n(x) = \lceil 10^n x - \frac{1}{2} \rceil / 10^n$. Then we wish to find

$$S_n = r_n(1) + r_n\left(\frac{1}{2}\right) + r_n\left(\frac{1}{3}\right) + \dots;$$

we know that $S_1 = 3.7$, and the problem is to write a complete MMIX program that calculates and prints S_n for $1 \leq n \leq 10$.

Note: There is a much faster way to do this than the simple procedure of adding $r_n(1/m)$, one number at a time, until $r_n(1/m)$ becomes zero. For example, we have $r_5(1/m) = 0.00001$ for all values of m from 66667 to 199999; it's wise to avoid calculating $1/m$ all 133333 times! An algorithm along the following lines is better.

- H1.** Start with $m_1 = 1$, $S \leftarrow 1$, $k \leftarrow 1$.
- H2.** Calculate $r \leftarrow r_n(1/(m_k + 1))$, and stop if $r = 0$.
- H3.** Find m_{k+1} , the largest m for which $r_n(1/m) = r$.
- H4.** Set $S \leftarrow S + (m_{k+1} - m_k)r$, $k \leftarrow k + 1$, and return to H2. ■

- 31.** [HM30] Using the notation of the preceding exercise, prove or disprove the formula

$$\lim_{n \rightarrow \infty} (S_{n+1} - S_n) = \ln 10.$$

- **32.** [31] The following algorithm, due to the Neapolitan astronomer Aloysius Lilius and the German Jesuit mathematician Christopher Clavius in the late 16th century, is used by most Western churches to determine the date of Easter Sunday for any year after 1582.

Algorithm E (Date of Easter). Let Y be the year for which Easter date is desired.

- E1.** [Golden number.] Set $G \leftarrow (Y \bmod 19) + 1$. (G is the so-called “golden number” of the year in the 19-year Metonic cycle.)
- E2.** [Century.] Set $C \leftarrow \lfloor Y/100 \rfloor + 1$. (When Y is not a multiple of 100, C is the century number; for example, 1984 is in the twentieth century.)
- E3.** [Corrections.] Set $X \leftarrow \lfloor 3C/4 \rfloor - 12$, $Z \leftarrow \lfloor (8C + 5)/25 \rfloor - 5$. (Here X is the number of years, such as 1900, in which leap year was dropped in order to keep in step with the sun; Z is a special correction designed to synchronize Easter with the moon’s orbit.)
- E4.** [Find Sunday.] Set $D \leftarrow \lfloor 5Y/4 \rfloor - X - 10$. (March $((-D) \bmod 7)$ will actually be a Sunday.)
- E5.** [Epact.] Set $E \leftarrow (11G + 20 + Z - X) \bmod 30$. If $E = 25$ and the golden number G is greater than 11, or if $E = 24$, increase E by 1. (This number E is the *epact*, which specifies when a full moon occurs.)
- E6.** [Find full moon.] Set $N \leftarrow 44 - E$. If $N < 21$ then set $N \leftarrow N + 30$. (Easter is supposedly the first Sunday following the first full moon that occurs on or after March 21. Actually perturbations in the moon’s orbit do not make this strictly true, but we are concerned here with the “calendar moon” rather than the actual moon. The N th of March is a calendar full moon.)
- E7.** [Advance to Sunday.] Set $N \leftarrow N + 7 - ((D + N) \bmod 7)$.
- E8.** [Get month.] If $N > 31$, the date is $(N - 31)$ APRIL; otherwise the date is N MARCH. ■

Write a subroutine to calculate and print Easter date given the year, assuming that the year is less than 100000. The output should have the form “*dd MONTH, yyyy*” where *dd* is the day and *yyyy* is the year. Write a complete MMIX program that uses this subroutine to prepare a table of the dates of Easter from 1950 through 2000.

- 33.** [M30] Some computers—not MMIX!—give a negative remainder when a negative number is divided by a positive number. Therefore a program for calculating the date of Easter by the algorithm in the previous exercise might fail when the quantity $(11G + 20 + Z - X)$ in step E5 is negative. For example, in the year 14250 we obtain $G = 1$, $X = 95$, $Z = 40$; so if we had $E = -24$ instead of $E = +6$ we would get

the ridiculous answer “42 APRIL”. [See CACM 5 (1962), 556.] Write a complete MMIX program that finds the *earliest* year for which this error would actually cause the wrong date to be calculated for Easter.

- 34. [33] Assume that an MMIX computer has been wired up to the traffic signals at the corner of Del Mar Boulevard and Berkeley Avenue, via special “files” named `/dev/lights` and `/dev/sensor`. The computer activates the lights by outputting one byte to `/dev/lights`, specifying the sum of four two-bit codes as follows:

```
Del Mar traffic light:    #00 off, #40 green, #80 amber, #c0 red;
Berkeley traffic light:  #00 off, #10 green, #20 amber, #30 red;
Del Mar pedestrian light: #00 off, #04 WALK, #0c DON'T WALK;
Berkeley pedestrian light: #00 off, #01 WALK, #03 DON'T WALK.
```

Cars or pedestrians wishing to travel on Berkeley across the boulevard must activate a sensor; if this condition never occurs, the light for Del Mar should remain green. When MMIX reads a byte from `/dev/sensor`, the input is nonzero if and only if the sensor has been activated since the previous input.

Cycle times are as follows:

```
Del Mar traffic light is green ≥ 30 sec, amber 8 sec;
Berkeley traffic light is green 20 sec, amber 5 sec.
```

When a traffic light is green or amber for one direction, the other direction has a red light. When the traffic light is green, the corresponding `WALK` light is on, except that `DON'T WALK` flashes for 12 sec just before a green light turns to amber, as follows:

```
DON'T WALK  1/2 sec} repeat 8 times;
off          1/2 sec}
DON'T WALK  4 sec (and remains on through amber and red cycles).
```

If the sensor is activated while the Berkeley light is green, the car or pedestrian will pass on that cycle. But if it is activated during the amber or red portions, another cycle will be necessary after the Del Mar traffic has passed.

Write a complete MMIX program that controls these lights, following the stated protocol. Assume that the special clock register `rC` increases by 1 exactly ρ times per second, where the integer ρ is a given constant.

35. [37] This exercise is designed to give some experience in the many applications of computers for which the output is to be displayed graphically rather than in the usual tabular form. The object is to “draw” a crossword puzzle diagram.

You are given as input a matrix of zeros and ones. An entry of zero indicates a white square; a one indicates a black square. The output should generate a diagram of the puzzle, with the appropriate squares numbered for words across and down.

For example, given the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

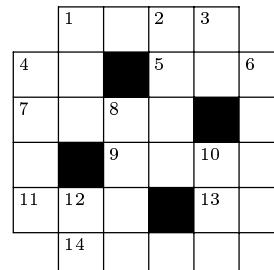


Fig. 18. Diagram corresponding to the matrix in exercise 35.

the corresponding puzzle diagram would be as shown in Fig. 18. A square is numbered if it is a white square and either (a) the square below it is white and there is no white square immediately above, or (b) the square to its right is white and there is no white square immediately to its left. If black squares occur at the edges, they should be removed from the diagram. This is illustrated in Fig. 18, where the black squares at the corners were dropped. A simple way to accomplish this is to artificially insert rows and columns of -1 's at the top, bottom, and sides of the given input matrix, then to change every $+1$ that is adjacent to a -1 into a -1 until no $+1$ remains next to any -1 .

Figure 18 was produced by the METAPOST program shown in Fig. 19. Simple changes to the uses of `line` and `black`, and to the coordinates in the `for` loop, will produce any desired diagram.

Write a complete MMIX program that reads a 25×25 matrix of zeros and ones in the standard input file and writes a suitable METAPOST program on the standard output file. The input should consist of 25 lines, each consisting of 25 digits followed by “newline”; for example, the first line corresponding to the matrix above would be ‘100001111111111111111111’, using extra 1s to extend the original 6×6 array. The diagram will not necessarily be symmetrical, and it might have long paths of black squares that are connected to the outside in strange ways.

```

beginfig(18)
transform t; t=identity rotated -90 scaled 17pt;
def line(expr i,j,ii,jj) =
  draw ((i,j)--(ii,jj)) transformed t;
enddef;
def black(expr i,j) =
  fill ((i,j)--(i+1,j)--(i+1,j+1)--(i,j+1)--cycle) transformed t;
enddef;
line (1,2,1,6); line (2,1,2,7); line (3,1,3,7); line (4,1,4,7);
line (5,1,5,7); line (6,1,6,7); line (7,2,7,6);
line (2,1,6,1); line (1,2,7,2); line (1,3,7,3); line (1,4,7,4);
line (1,5,7,5); line (1,6,7,6); line (2,7,6,7);
numeric n; n=0;
for p = (1,2),(1,4),(1,5), (2,1),(2,4),(2,6),
    (3,1),(3,3), (4,3),(4,5), (5,1),(5,2),(5,5), (6,2):
  n:=n+1; label.lrt(decimal n infont "cmr8", p transformed t);
endfor
black(2,3); black(3,5); black(4,2); black(5,4);
endfig;

```

Fig. 19. The METAPOST program that generated Fig. 18.

1.3.3'. Applications to Permutations

 The MIX programs in the former Section 1.3.3 will all be converted to MMIX programs, and so will the MIX programs in Chapters 2, 3, 4, 5, and 6. Anyone who wishes to help with this instructive conversion project is invited to join the MMIXmasters (see page v).

1.4'. SOME FUNDAMENTAL PROGRAMMING TECHNIQUES

1.4.1'. Subroutines

WHEN A CERTAIN task is to be performed at several different places in a program, we usually don't want to repeat the coding over and over. To avoid this situation, the coding (called a *subroutine*) can be put into one place only, and a few extra instructions can be added to restart the main routine properly after the subroutine is finished. Transfer of control between subroutines and main programs is called *subroutine linkage*.

Each machine has its own peculiar way to achieve efficient subroutine linkage, usually by using special instructions. Our discussion will be based on MMIX machine language, but similar remarks will apply to subroutine linkage on most other general-purpose computers.

Subroutines are used to save space in a program. They do not save any time, other than the time implicitly saved by having less space—for example, less time to load the program, and better use of high-speed memory on machines with several grades of memory. The extra time taken to enter and leave a subroutine is usually negligible, except in critical innermost loops.

Subroutines have several other advantages. They make it easier to visualize the structure of a large and complex program; they form a logical segmentation of the entire problem, and this usually makes debugging of the program easier. Many subroutines have additional value because they can be used by people other than the programmer of the subroutine.

Most computer installations have built up a large library of useful subroutines, and such a library greatly facilitates the programming of standard computer applications that arise. A programmer should not think of this as the *only* purpose of subroutines, however; subroutines should not always be regarded as general-purpose programs to be used by the community. Special-purpose subroutines are just as important, even when they are intended to appear in only one program. Section 1.4.3' contains several typical examples.

The simplest subroutines are those that have only one entrance and one exit, such as the **Maximum** subroutine we have already considered (see Program M in Section 1.3.2' and exercise 1.3.2'-3). Let's look at that program again, recasting it slightly so that a fixed number of cells, 100, is searched for the maximum:

```
* Maximum of X[1..100]
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
Max100 SETL kk,100*8 M1. Initialize.
      LDO   m,x0,kk
      JMP   1F
3H    LDO   xk,x0,kk M3. Compare.
      CMP   t,xk,m
      PBNP  t,5F
4H    SET   m,xk      M4. Change m.
1H    SR    j,kk,3
5H    SUB   kk,kk,8   M5. Decrease k.
      PBP   kk,3B      M2. All tested?
6H    POP   2,0       Return to main program. ■
```

(1)

This subroutine is assumed to be part of a larger program in which the symbol **t** has been defined to stand for register \$255, and the symbol **x0** has been defined to stand for a global register such that $X[k]$ appears in location $x0 + 8k$. In that larger program, the single instruction “**PUSHJ \$1,Max100**” will cause register \$1 to be set to the current maximum value of $\{X[1], \dots, X[100]\}$, and the position of the maximum will appear in \$2. Linkage in this case is achieved by the **PUSHJ** instruction that invokes the subroutine, together with “**POP 2,0**” at the subroutine’s end. These MMIX instructions cause local registers to be renumbered while the subroutine is active; furthermore, the **PUSHJ** inserts a return address into special register **rJ**, and the **POP** jumps to this location.

We can also accomplish subroutine linkage in a simpler, rather different way, by using MMIX’s **GO** instruction instead of pushing and popping. We might, for instance, use the following code in place of (1):

```
* Maximum of X[1..100]
j GREG ;m GREG ;kk GREG ;xk GREG
          GREG @           Base address
GoMax100 SETL  kk,100*8  M1. Initialize.
          LDO   m,x0,kk
          JMP   1F
3H      ...          (Continue as in (1))
          PBP   kk,3B    M2. All tested?
6H      GO    kk,$0,0  Return to main program. ■
```

(2)

Now the instruction “**GO \$0,GoMax100**” will transfer control to the subroutine, placing the address of the following instruction into \$0; the subsequent “**GO kk,\$0,0**” at the subroutine’s end will return to this address. In this case the maximum value will appear in global register **m**, and its location will be in global register **j**. Two additional global registers, **kk** and **xk**, have also been set aside for use by this subroutine. Furthermore, the “**GREG @**” provides a base address so that we can **GO** to **GoMax100** in a single instruction; otherwise a two-step sequence like “**GETA \$0,GoMax100; GO \$0,\$0,0**” would be necessary. Subroutine linkage like (2) is commonly used on machines that have no built-in register stack mechanism.

It is not hard to obtain *quantitative* statements about the amount of code saved and the amount of time lost when subroutines are used. Suppose that a piece of coding requires k tetrabytes and that it appears in m places in the program. Rewriting this as a subroutine, we need a **PUSHJ** or **GO** instruction in each of the m places where the subroutine is called, plus a single **POP** or **GO** instruction to return control. This gives a total of $m + k + 1$ tetrabytes, rather than mk , so the amount saved is

$$(m - 1)(k - 1) - 2. \quad (3)$$

If k is 1 or m is 1 we cannot possibly save any space by using subroutines; this, of course, is obvious. If k is 2, m must be greater than 3 in order to gain, etc.

The amount of time lost is the time taken for the **PUSHJ**, **POP**, and/or **GO** instructions in the linkage. If the subroutine is invoked t times during a run of the

program, and if we assume that running time is governed by the approximations in Table 1.3.1'-1, the extra cost is $4tv$ in case (1), or $6tv$ in case (2).

These estimates must be taken with a grain of salt, because they were given for an idealized situation. Many subroutines cannot be called simply with a single **PUSHJ** or **G0** instruction. Furthermore, if code is replicated in many parts of a program without using a subroutine approach, each instance can be customized to take advantage of special characteristics of the particular part of the program in which it lies. With a subroutine, on the other hand, the code must be written for the most general case; this will often add several additional instructions.

When a subroutine is written to handle a general case, it is expressed in terms of *parameters*. Parameters are values that govern a subroutine's actions; they are subject to change from one call of the subroutine to another.

The coding in the outside program that transfers control to a subroutine and gets it properly started is known as the *calling sequence*. Particular values of parameters, supplied when the subroutine is called, are known as *arguments*. With our **GoMax100** subroutine, the calling sequence is simply “**G0 \$0, GoMax100**”, but a longer calling sequence is generally necessary when arguments must be supplied.

For example, we might want to generalize (2) to a subroutine that finds the maximum of the first n elements of an array, given *any* constant n , by placing n in the instruction stream with the two-step calling sequence

$$\text{G0 } \$0, \text{GoMax}; \quad \text{TETRA } n. \quad (4)$$

The **GoMax** subroutine could then take the form

```
* Maximum of X[1..n]
j GREG ;m GREG ;kk GREG ;xk GREG
          GREG @           Base address
GoMax LDT   kk,$0,0  Fetch the argument.
      SL    kk,kk,3
      LDO   m,x0,kk
      JMP   1F
3H     ...          (Continue as in (1))
      PBP   kk,3B
6H     G0    kk,$0,4  Return to caller. ■
```

(5)

Still better would be to communicate the parameter n by putting it into a register. We could, for example, use the two-step calling sequence

$$\text{SET } \$1,n; \quad \text{G0 } \$0, \text{GoMax} \quad (6)$$

together with a subroutine of the form

```
GoMax SL    kk,$1,3  Fetch the argument.
      LDO   m,x0,kk
      ...
6H     G0    kk,$0,0  Return. ■
```

(7)

This variation is faster than (5), and it allows n to vary dynamically without modifying the instruction stream.

Notice that the address of array element $X[0]$ is also essentially a parameter to subroutines (1), (2), (5), and (7). The operation of putting this address into register $x0$ may be regarded as part of the calling sequence, in cases when the array is different each time.

If the calling sequence occupies c tetrabytes of memory, formula (3) for the amount of space saved changes to

$$(m - 1)(k - c) - \text{constant} \quad (8)$$

and the time lost for subroutine linkage is slightly increased.

A further correction to the formulas above can be necessary because certain registers might need to be saved and restored. For example, in the `GoMax` subroutine we must remember that by writing “SET \$1,n; GO \$0,GoMax” we are not only computing the maximum value in register m and its position in register j , we are also changing the values of global registers kk and xk . We have implemented (2), (5), and (7) with the implicit assumption that registers kk and xk are for the exclusive use of the maximum-finding routine, but many computers are not blessed with a large number of registers. Even MMIX will run out of registers if a lot of subroutines are present simultaneously. We might therefore want to revise (7) so that it will work with $kk \equiv \$2$ and $xk \equiv \$3$, say, without clobbering the contents of those registers. We could do this by writing

```

j GREG ;m GREG ;kk IS $2 ;xk IS $3
      GREG   @          Base address
GoMax STO    kk,Tempkk Save previous register contents.
      STO    xk,Tempxk
      SL     kk,$1,3    Fetch the argument.
      LDO    m,x0,kk
      ...
      LDO    kk,Tempkk Restore previous register contents.
      LDO    xk,Tempxk
6H     GO    $0,$0,0   Return.  ■

```

(9)

and by setting aside two octabytes called `Tempkk` and `Tempxk` in the data segment. Of course this change adds potentially significant overhead cost to each use of the subroutine.

A subroutine may be regarded as an *extension* of the computer’s machine language. For example, whenever the `GoMax` subroutine is present in memory we have a single machine instruction (namely, “GO \$0,GoMax”) that is a maximum-finder. It is important to define the effect of each subroutine just as carefully as the machine language operators themselves have been defined; a programmer should therefore be sure to write down the relevant characteristics, even though nobody else will be making use of the routine or its specification. In the case of `GoMax` as given in (7) or (9), the characteristics are as follows:

Calling sequence: `GO $0,GoMax`.
 Entry conditions: $\$1 = n \geq 1$; $x0 = \text{address of } X[0]$.
 Exit conditions: $m = \max_{1 \leq k \leq n} X[k] = X[j]$.

(10)

A specification should mention all changes to quantities that are external to the subroutine. If registers **kk** and **xk** are not considered “private” to the variant of **GoMax** in (7), we should include the fact that those registers are affected, as part of that subroutine’s exit conditions. The subroutine also changes register **t**, namely register \$255; but that register is conventionally used for temporary quantities of only momentary significance, so we needn’t bother to list it explicitly.

Now let’s consider *multiple entrances* to subroutines. Suppose we have a program that requires the general subroutine **GoMax**, but it usually wants to use the special case **GoMax100** in which $n = 100$. The two can be combined as follows:

```
GoMax100 SET $1,100 First entrance
GoMax    ... Second entrance; continue as in (7) or (9). ■ (11)
```

We could also add a *third* entrance, say **GoMax50**, by putting the code

```
GoMax50 SET $1,50; JMP GoMax
```

in some convenient place.

A subroutine might also have *multiple exits*, meaning that it is supposed to return to one of several different locations, depending on conditions that it has detected. For example, we can extend subroutine (11) yet again by assuming that an upper bound parameter is given in global register **b**; the subroutine is now supposed to exit to one of the two tetrabytes following the **GO** instruction that calls it:

Calling sequence for general n	Calling sequence for $n = 100$
SET \$1, n ; GO \$0,GoMax	GO \$0,GoMax100
Exit here if $m \leq 0$ or $m \geq b$.	Exit here if $m \leq 0$ or $m \geq b$.
Exit here if $0 < m < b$.	Exit here if $0 < m < b$.

(In other words, we skip the tetrabyte after the **GO** when the maximum value is positive and less than the upper bound. A subroutine like this would be useful in a program that often needs to make such distinctions after computing a maximum value.) The implementation is easy:

```
* Maximum of X[1..n] with bounds check
j GREG ;m GREG ;kk GREG ;xk GREG
          GREG   @      Base address
GoMax100 SET    $1,100  Entrance for  $n = 100$ 
GoMax    SL     kk,$1,3  Entrance for general  $n$ 
          LDO    m,x0,kk
          JMP    1F
3H      ...           (Continue as in (1))                               (12)
          PBP    kk,3B
          BNP    m,1F      Branch if  $m \leq 0$ .
          CMP    kk,m,b
          BN     kk,2F      Branch if  $m < b$ .
1H      GO     kk,$0,0  Take first exit if  $m \leq 0$  or  $m \geq b$ .
2H      GO     kk,$0,4  Otherwise take second exit. ■
```

Notice that this program combines the instruction-stream linking technique of (5) with the register-setting technique of (7). The location to which a subroutine exits is, strictly speaking, a parameter; hence the locations of multiple exits must be supplied as arguments. When a subroutine accesses one of its parameters all the time, the corresponding argument is best passed in a register, but when an argument is constant and not always needed it is best kept in the instruction stream.

Subroutines may call on other subroutines. Indeed, complicated programs often have subroutine calls nested more than five deep. The only restriction that must be followed when using the GO-type linkage described above is that all temporary storage locations and registers must be distinct; thus no subroutine may call on any other subroutine that is (directly or indirectly) calling on it. For example, consider the following scenario:

[Main program]	[Subroutine A]	[Subroutine B]	[Subroutine C]	
:	A :	B :	C :	
GO \$0,A	GO \$1,B	GO \$2,C	GO \$0,A	(13)
:	⋮	⋮	⋮	
	GO \$0,\$0,0	GO \$1,\$1,0	GO \$2,\$2,0	

If the main program calls A, which calls B, which calls C, and then C calls on A, the address in \$0 referring to the main program is destroyed, and there is no way to return to that program.

Using a memory stack. Recursive situations like (13) do not often arise in simple programs, but a great many important applications do have a natural recursive structure. Fortunately there is a straightforward way to avoid interference between subroutine calls, by letting each subroutine keep its local variables on a *stack*. For example, we can set aside a global register called sp (the “stack pointer”) and use GO \$0,Sub to invoke each subroutine. If the code for the subroutine has the form

```
Sub  STO  $0,sp,0
      ADD  sp,sp,8
      ...
      SUB  sp,sp,8
      LDO  $0,sp,0
      GO   $0,$0,0
```

(14)

register \$0 will always contain the proper return address; the problem of (13) no longer arises. (Initially we set sp to an address in the data segment, following all other memory locations needed.) Moreover, the STO/ADD and SUB/LDO instructions of (14) can be omitted if Sub is a so-called *leaf subroutine*—a subroutine that doesn’t call any other subroutines.

A stack can be used to hold parameters and other local variables besides the return addresses stored in (14). Suppose, for example, that subroutine Sub needs 20 octabytes of local data, in addition to the return address; then we can

use a scheme like this:

Sub	STO fp,sp,0	Save the old frame pointer.
	SET fp,sp	Establish a new frame pointer.
	INCL sp,8*22	Advance the stack pointer.
	STO \$0,fp,8	Save the return address.
...		
	LDO \$0,fp,8	Restore the return address.
	SET sp,fp	Restore the stack pointer.
	LDO fp,sp,0	Restore the frame pointer.
	GO \$0,\$0,0	Return to caller. ■

(15)

Here *fp* is a global register called the *frame pointer*. Within the “...” part of the subroutine, local quantity number *k* is equivalent to the octabyte in memory location *fp* + 8*k* + 8, for $1 \leq k \leq 20$. The instructions at the beginning are said to “push” local quantities onto the “top” of the stack; the instructions at the end “pop” those quantities off, leaving the stack in the condition it had when the subroutine was entered.

Using the register stack. We have discussed *G0*-type subroutine linkage at length because many computers have no better alternative. But MMIX has built-in instructions **PUSHJ** and **POP**, which handle subroutine linkage in a more efficient way, avoiding most of the overhead in schemes like (9) and (15). These instructions allow us to keep most parameters and local variables entirely in registers, instead of storing them into a memory stack and loading them again later. With **PUSHJ** and **POP**, most of the details of stack maintenance are done automatically by the machine.

The basic idea is quite simple, once the general idea of a stack is understood. MMIX has a *register stack* consisting of octabytes $S[0], S[1], \dots, S[\tau - 1]$ for some number $\tau \geq 0$. The topmost *L* octabytes in the stack (namely $S[\tau - L], S[\tau - L + 1], \dots, S[\tau - 1]$) are the current *local registers* $\$0, \$1, \dots, \$L$; the other $\tau - L$ octabytes of the stack are currently inaccessible to the program, and we say they have been “pushed down.” The current number of local registers, *L*, is kept in MMIX’s special register *rL*, although a programmer rarely needs to know this. Initially $L = 2$, $\tau = 2$, and local registers $\$0$ and $\$1$ represent the command line as in Program 1.3.2’H.

MMIX also has *global registers*, namely $\$G, \$G+1, \dots, \$255$; the value of *G* is kept in special register *rG*, and we always have $0 \leq L \leq G \leq 255$. (In fact, we also always have $G \geq 32$.) Global registers are *not* part of the register stack.

Registers that are neither local nor global are called *marginal*. These registers, namely $\$L, \$L+1, \dots, \$G-1$, have the value zero whenever they are used as input operands to an MMIX instruction.

The register stack grows when a marginal register is given a value. This marginal register becomes local, and so do all marginal registers with smaller numbers. For example, if eight local registers are currently in use, the instruction **ADD \$10,\$20,5** causes $\$8, \9 , and $\$10$ to become local; more precisely, if *rL* = 8, the instruction **ADD \$10,\$20,5** sets $\$8 \leftarrow 0, \$9 \leftarrow 0, \$10 \leftarrow 5$, and *rL* $\leftarrow 11$. (Register $\$20$ remains marginal.)

If $\$X$ is a local register, the instruction `PUSHJ $X,Sub` decreases the number of local registers and changes their effective register numbers: Local registers previously called $\$(X+1), \$(X+2), \dots, \$(L-1)$ are called $\$0, \$1, \dots, \$(L-X-2)$ inside the subroutine, and the value of L decreases by $X + 1$. Thus the register stack remains unchanged, but $X + 1$ of its entries have become inaccessible; the subroutine cannot damage those entries, and it has $X+1$ newly marginal registers to play with.

If $X \geq G$, so that $\$X$ is a global register, the action of `PUSHJ $X,Sub` is similar, but a new entry is placed on the register stack and then $L+1$ registers are pushed down instead of $X+1$. In this case L is zero when the subroutine begins; all of the formerly local registers have been pushed down, and the subroutine starts out with a clean slate.

The register stack shrinks only when a `POP` instruction is given, or when a program explicitly decreases the number of local registers with an instruction such as `PUT rL,5`. The purpose of `POP X,YZ` is to make the items pushed down by the most recent `PUSHJ` accessible again, as they were before, and to remove items from the register stack if they are no longer necessary. In general the X field of a `POP` instruction is the number of values “returned” by the subroutine, if $X \leq L$. If $X > 0$, the main value returned is $\$(X-1)$; this value is removed from the register stack, together with all entries above it, and the return value is placed in the position specified by the `PUSHJ` command that invoked the subroutine. The behavior of `POP` is similar when $X > L$, but in this case the register stack remains intact and zero is placed in the position of the `PUSHJ`.

The rules we have just stated are a bit complicated, because many different cases can arise in practice. A few examples will, however, make everything clear. Suppose we are writing a routine `A` and we want to call subroutine `B`; suppose further that routine `A` has 5 local registers that should not be accessible to `B`. These registers are $\$0, \$1, \$2, \3 , and $\$4$. We reserve the next register, $\$5$, for the main result of subroutine `B`. If `B` has, say, three parameters, we set $\$6 \leftarrow \text{arg}0$, $\$7 \leftarrow \text{arg}1$, and $\$8 \leftarrow \text{arg}2$, then issue the command `PUSHJ $5,B`; this invokes `B` and the arguments are now found in $\$0, \1 , and $\$2$.

If `B` returns no result, it will conclude with the command `POP 0,YZ`; this will restore $\$0, \$1, \$2, \3 , and $\$4$ to their former values and set $L \leftarrow 5$.

If `B` returns a single result x , it will place x in $\$0$ and conclude with the command `POP 1,YZ`. This will restore $\$0, \$1, \$2, \3 , and $\$4$ as before; it will also set $\$5 \leftarrow x$ and $L \leftarrow 6$.

If `B` returns two results x and a , it will place the main result x in $\$1$ and the auxiliary result a in $\$0$. Then `POP 2,YZ` will restore $\$0$ through $\$4$ and set $\$5 \leftarrow x$, $\$6 \leftarrow a$, $L \leftarrow 7$. Similarly, if `B` returns ten results (x, a_0, \dots, a_8) , it will place the main result x in $\$9$ and the others in the first nine registers: $\$0 \leftarrow a_0$, $\$1 \leftarrow a_1, \dots, \$8 \leftarrow a_8$. Then `POP 10,YZ` will restore $\$0$ through $\$4$ and set $\$5 \leftarrow x$, $\$6 \leftarrow a_0, \dots, \$14 \leftarrow a_8$. (The curious permutation of registers that arises when two or more results are returned may seem strange at first. But it makes sense, because it leaves the register stack unchanged except for the main result. For example, if subroutine `B` wants `arg0, arg1, and arg2` to reappear in

$\$6$, $\$7$, and $\$8$ after it has finished its work, it can leave them as auxiliary results in $\$0$, $\$1$, and $\$2$ and then say `POP 4,YZ.`)

The YZ field of a `POP` instruction is usually zero, but in general the instruction `POP X,YZ` returns to the instruction that is $YZ+1$ tetrabytes after the `PUSHJ` that invoked the current subroutine. This generality is useful for subroutines with multiple exits. More precisely, a `PUSHJ` subroutine in location $@$ sets special register rJ to $@ + 4$ before jumping to the subroutine; a `POP` instruction then returns to location $rJ + 4YZ$.

We can now recast the programs previously written with `GO` linkage so that they use `PUSH/POP` linkage instead. For example, the two-entrance, two-exit subroutine for maximum-finding in (12) takes the following form when **MMIX**'s register stack mechanism is used:

```
* Maximum of X[1..n] with bounds check
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
Max100 SET    $0,100   Entrance for n = 100
Max     SL      kk,$0,3   Entrance for general n
        LDO     m,x0,kk
        JMP     1F
        ...
        BNZ     kk,2F          (Continue as in (12))
1H     POP     2,0       Take first exit if max ≤ 0 or max ≥ b.
2H     POP     2,1       Otherwise take second exit. ■
```

(16)

Calling sequence for general n

```
SET $A,n; PUSHJ $R,Max (A = R+1)
Exit here if $R ≤ 0 or $R ≥ b.
Exit here if 0 < $R < b.
```

Calling sequence for $n = 100$

```
PUSHJ $R,Max100
Exit here if $R ≤ 0 or $R ≥ b.
Exit here if 0 < $R < b.
```

The local result register $$R$ in the `PUSHJ` of this calling sequence is arbitrary, depending on the number of local variables the caller wishes to retain. The local argument register $$A$ is then $(R+1)$. After the call, $$R$ will contain the main result (the maximum value) and $$A$ will contain the auxiliary result (the array index of that maximum). If there are several arguments and/or auxiliaries, they are conventionally called A_0, A_1, \dots , and we conventionally assume that $A_0 = R+1, A_1 = R+2, \dots$ when `PUSH/POP` calling sequences are written down.

A comparison of (12) and (16) shows only mild advantages for (16): The new form does not need to allocate global registers for j , m , kk , and xk , nor does it need a global base register for the address of the `GO` command. (Recall from Section 1.3.1' that `GO` takes an absolute address, while `PUSHJ` has a relative address.) A `GO` instruction is slightly slower than `PUSHJ`; it is no slower than `POP`, according to Table 1.3.1'-1, although high-speed implementations of **MMIX** could implement `POP` more efficiently. Programs (12) and (16) both have the same length.

The advantages of `PUSH/POP` linkage over `GO` linkage begin to manifest themselves when we have *non-leaf* subroutines (namely, subroutines that call other subroutines, possibly themselves). Then the `GO`-based code of (14) can be re-

placed by

```
Sub  GET  retadd,rJ
      ...
      PUT  rJ,retadd
      POP  X,0
```

(17)

where `retadd` is a local register. (For example, `retadd` might be \$5; its register number is generally greater than or equal to the number of returned results `X`, so the `POP` instruction will automatically remove it from the register stack.) Now the costly memory references of (14) are avoided.

A non-leaf subroutine with many local variables and/or parameters is significantly better off with a register stack than with the memory stack scheme of (15), because we can often perform the computations entirely in registers. We should note, however, that MMIX's register stack applies only to local variables that are *scalar*, not to local array variables that must be accessed by address computation. Subroutines that need non-scalar local variables should use a scheme like (15) for all such variables, while keeping scalars on the register stack. Both approaches can be used simultaneously, with `fp` and `sp` updated only by subroutines that need a memory stack.

If the register stack becomes extremely large, MMIX will automatically store its bottom entries in the stack segment of memory, using a behind-the-scenes procedure that we will study in Section 1.4.3'. (Recall from Section 1.3.2' that the stack segment begins at address #6000 0000 0000 0000.) MMIX stores register stack items in memory also when a `SAVE` command saves a program's entire current context. Saved stack items are automatically restored from memory when a `POP` command needs them or when an `UNSAVE` command restores a saved context. But in most cases MMIX is able to push and pop local registers without actually accessing memory, and without actually changing the contents of very many internal machine registers.

Stacks have many other uses in computer programs; we will study their basic properties in Section 2.2.1. We will get a further taste of nested subroutines and recursive procedures in Section 2.3, when we consider operations on trees. Chapter 8 studies recursion in detail.

***Assembly language features.** The MMIX assembly language supports the writing of subroutines in three ways that were not mentioned in Section 1.3.2'. The most important of these is the `PREFIX` operation, which makes it easy to define "private" symbols that will not interfere with symbols defined elsewhere in a large program. The basic idea is that a symbol can have a structured form like `Sub:X` (meaning symbol `X` of subroutine `Sub`), possibly carried to several levels like `Lib:Sub:X` (meaning symbol `X` of subroutine `Sub` in library `Lib`).

Structured symbols are accommodated by extending rule 1 of MMIXAL in Section 1.3.2' slightly, allowing the colon character ':' to be regarded as a "letter" that can be used to construct symbols. Every symbol that does not begin with a colon is implicitly extended by placing the *current prefix* in front of it. The current prefix is initially ':', but the user can change it with the

PREFIX command. For example,

ADD x,y,z	means ADD :x,:y,:z
PREFIX Foo:	current prefix is :Foo:
ADD x,y,z	means ADD :Foo:x,:Foo:y,:Foo:z
PREFIX Bar:	current prefix is :Foo:Bar:
ADD :x,y,:z	means ADD :x,:Foo:Bar:y,:z
PREFIX :	current prefix reverts to :
ADD x,Foo:Bar:y,Foo:z	means ADD :x,:Foo:Bar:y,:Foo:z

One way to use this idea is to replace the opening lines of (16) by

```

PREFIX Max:
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
x0 IS :x0 ;b IS :b ;t IS :t      External symbols
:Max100 SET    $0,100   Entrance for n = 100
:Max     SL      kk,$0,3   Entrance for general n          (18)
LDO      m,x0,kk
JMP      1F
...
(Continue as in (16))

```

and to add “PREFIX :” at the end. Then the symbols `j`, `m`, `kk`, and `xk` are free for use in the rest of the program or in the definition of other subroutines. Further examples of the use of prefixes appear in Section 1.4.3’.

MMIXAL also includes a pseudo-operation called LOCAL. The assembly command “LOCAL \$40” means, for example, that an error message should be given at the end of assembly if GREG commands allocate so many registers that \$40 will be global. (This feature is needed only when a subroutine uses more than 32 local registers, because “LOCAL \$31” is always implicitly true.)

A third feature for subroutine support, BSPEC ... ESPEC, is also provided. It allows information to be passed to the object file so that debugging routines and other system programs know what kind of linkage is being used by each subroutine. This feature is discussed in the *MMIXware* document; it is primarily of interest in the output of compilers.

Strategic considerations. When ad hoc subroutines are written for special-purpose use, we can afford to use GREG instructions liberally, so that plenty of global registers are filled with basic constants that make our program run fast. Comparatively few local registers are needed, unless the subroutines are used recursively.

But when dozens or hundreds of general-purpose subroutines are written for inclusion in a large library, with the idea of allowing any user program to include whatever subroutines it needs, we obviously can’t allow each subroutine to allocate a substantial number of globals. Even one global variable per subroutine might be too much.

Thus we want to use GREG generously when we have only a few subroutines, but we want to use it sparingly when the number of subroutines is potentially huge. In the latter case we probably can make good use of local variables without too much loss of efficiency.

Let's conclude this section by discussing briefly how we might go about writing a complex and lengthy program. How can we decide what kind of subroutines we will need? What calling sequences should be used? One successful way to determine this is to use an iterative procedure:

Step 0 (Initial idea). First we decide vaguely upon the general plan of attack that the program will use.

Step 1 (A rough sketch of the program). We start now by writing the “outer levels” of the program, in any convenient language. A somewhat systematic way to go about this has been described very nicely by E. W. Dijkstra, *Structured Programming* (Academic Press, 1972), Chapter 1, and by N. Wirth, *CACM* **14** (1971), 221–227. First we break the whole program into a small number of pieces, which might be thought of temporarily as subroutines although they are called only once. These pieces are successively refined into smaller and smaller parts, having correspondingly simpler jobs to do. Whenever some computational task arises that seems likely to occur elsewhere or that has already occurred elsewhere, we define a subroutine (a real one) to do that job. We do not write the subroutine at this point; we continue writing the main program, assuming that the subroutine has performed its task. Finally, when the main program has been sketched, we tackle the subroutines in turn, trying to take the most complex subroutines first and then their sub-subroutines, etc. In this manner we will come up with a list of subroutines. The actual function of each subroutine has probably already changed several times, so that the first parts of our sketch will by now be incorrect; but that is no problem, since we are merely making a sketch. We now have a reasonably good idea about how each subroutine will be called and how general-purpose it should be. We should consider extending the generality of each subroutine, at least a little.

Step 2 (First working program). The next step goes in the opposite direction from step 1. We now write in computer language, say MMIXAL or PL/MMIX or—most probably—a higher-level language. We start this time with the lowest level subroutines, and do the main program last. As far as possible, we try never to write any instructions that call a subroutine before the subroutine itself has been coded. (In step 1, we tried the opposite, never considering a subroutine until all of its calls had been written.)

As more and more subroutines are written during this process, our confidence gradually grows, since we are continually extending the power of the machine we are programming. After an individual subroutine is coded, we should immediately prepare a complete description of what it does, and what its calling sequences are, as in (10). It is also important to be sure that global variables are not used for two conflicting purposes at the same time; when preparing the sketch in step 1, we didn't have to worry about such problems.

Step 3 (Reexamination). The result of step 2 should be very nearly a working program, but we may be able to improve it. A good way is to reverse direction again, studying for each subroutine *all* of the places it is called. Perhaps the subroutine should be enlarged to do some of the more common things that

are always done by the outside routine just before or after the subroutine is called. Perhaps several subroutines should be merged into one; or perhaps a subroutine is called only once and should not be a subroutine at all. Perhaps a subroutine is never called and can be dispensed with entirely.

At this point, it is often a good idea to scrap everything and start over again at step 1, or even at step 0! This is not intended to be a facetious remark; the time spent in getting this far has not been wasted, for we have learned a great deal about the problem. With hindsight, we will probably have discovered several improvements that could be made to the program's overall organization. There's no reason to be afraid to go back to step 1—it will be much easier to go through steps 2 and 3 again, now that a similar program has been done already. Moreover, we will quite probably save as much debugging time later on as it will take to rewrite everything. Some of the best computer programs ever written owe much of their success to the fact that all the work was unintentionally lost, at about this stage, and the authors were forced to begin again.

On the other hand, there is probably never a point when a complex computer program cannot be improved somehow, so steps 1 and 2 should not be repeated indefinitely. When significant improvements can clearly be made, the additional time required to start over is well spent, but eventually a point of diminishing returns is reached.

Step 4 (Debugging). After a final polishing of the program, including perhaps the allocation of storage and other last-minute details, it is time to look at it in still another direction from the three that were used in steps 1, 2, and 3: Now we study the program in the order in which the computer will *perform* it. This may be done by hand or, of course, by machine. The author has found it quite helpful at this point to make use of system routines that trace each instruction the first two times it is executed; it is important to rethink the ideas underlying the program and to check that everything is actually taking place as expected.

Debugging is an art that needs much further study, and the way to approach it is highly dependent on the facilities available at each computer installation. A good start towards effective debugging is often the preparation of appropriate test data. The most successful debugging techniques are typically designed and built into the program itself: Many of today's best programmers devote nearly half of their programs to facilitating the debugging process in the other half. The first half, which usually consists of fairly straightforward routines that display relevant information in a readable format, will eventually be of little importance, but the net result is a surprising gain in productivity.

Another good debugging practice is to keep a record of every mistake made. Even though this will probably be quite embarrassing, such information is invaluable to anyone doing research on the debugging problem, and it will also help you learn how to cope with future errors.

Note: The author wrote most of the preceding comments in 1964, after he had successfully completed several medium-sized software projects but before he had developed a mature programming style. Later, during the 1980s, he

learned that an additional technique, called *structured documentation* or *literate programming*, is probably even more important. A summary of his current beliefs about the best way to write programs of all kinds appears in the book *Literate Programming* (Cambridge University Press, first published in 1992). Incidentally, Chapter 11 of that book contains a detailed record of all bugs removed from the TeX program during the period 1978–1991.

Up to a point it is better to let the snags [bugs] be there than to spend such time in design that there are none (how many decades would this course take?).

— A. M. TURING, Proposals for ACE (1945)

EXERCISES

1. [20] Write a subroutine `GoMaxR` that generalizes Algorithm 1.2.10M by finding the maximum value of $\{X[a], X[a+r], X[a+2r], \dots, X[n]\}$, where r and n are positive parameters and a is the smallest positive number with $a \equiv n$ (modulo r), namely $a = 1 + (n - 1) \bmod r$. Give a special entrance `GoMax` for the case $r = 1$, using a G0-style calling sequence so that your subroutine is a generalization of (7).
2. [20] Convert the subroutine of exercise 1 from G0 linkage to `PUSHJ/POP` linkage.
3. [15] How can scheme (15) be simplified when `Sub` is a leaf subroutine?
4. [15] The text in this section speaks often of `PUSHJ`, but Section 1.3.1' mentions also a command called `PUSHGO`. What is the difference between `PUSHJ` and `PUSHGO`?
5. [0] True or false: The number of marginal registers is $G - L$.
6. [10] What is the effect of the instruction `DIVU $5,$5,$5` if $\$5$ is a marginal register?
7. [10] What is the effect of the instruction `INCML $5,#abcd` if $\$5$ is a marginal register?
8. [15] Suppose the instruction `SET $15,0` is performed when there are 10 local registers. This increases the number of local registers to 16; but the newly local registers (including $\$15$) are all zero, so they still behave essentially as if they were marginal. Is the instruction `SET $15,0` therefore entirely redundant in such a case?
9. [20] When a trip interrupt has been enabled for some exceptional condition like arithmetic overflow, the trip handler might be called into action at unpredictable times. We don't want to clobber any of the interrupted program's registers; yet a trip handler can't do much unless it has "elbow room." Explain how to use `PUSHJ` and `POP` so that plenty of local registers are safely available to a handler.
- ▶ 10. [20] True or false: If an MMIX program never uses the instructions `PUSHJ`, `PUSHGO`, `POP`, `SAVE`, or `UNSAVE`, all 256 registers $\$0, \$1, \dots, \$255$ are essentially equivalent, in the sense that the distinction between local, global, and marginal registers is irrelevant.
11. [20] Guess what happens if a program issues more `POP` instructions than `PUSH` instructions.
- ▶ 12. [10] True or false:
 - a) The current prefix in an MMIXAL program always begins with a colon.
 - b) The current prefix in an MMIXAL program always ends with a colon.
 - c) The symbols `:` and `::` are equivalent in MMIXAL programs.

- 13. [21] Write two MMIX subroutines to calculate the Fibonacci number $F_n \bmod 2^{64}$, given n . The first subroutine should call itself recursively, using the definition

$$F_n = n \quad \text{if } n \leq 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{if } n > 1.$$

The second subroutine should *not* be recursive. Both subroutines should use PUSH/POP linkage and should avoid global variables entirely.

- 14. [M21] What is the running time of the subroutines in exercise 13?
- 15. [21] Convert the recursive subroutine of exercise 13 to GO-style linkage, using a memory stack as in (15) instead of MMIX's register stack. Compare the efficiency of the two versions.
- 16. [25] (*Nonlocal goto statements.*) Sometimes we want to jump out of a subroutine, to a location that is not in the calling routine. For example, suppose subroutine A calls subroutine B, which calls subroutine C, which calls itself recursively a number of times before deciding that it wants to exit directly to A. Explain how to handle such situations when using MMIX's register stack. (We can't simply JMP from C to A; the stack must be properly popped.)

1.4.2'. Coroutines

Subroutines are special cases of more general program components, called *coroutines*. In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines, which *call on each other*.

To understand the coroutine concept, let us consider another way of thinking about subroutines. The viewpoint adopted in the previous section was that a subroutine was merely an extension of the computer hardware, introduced to save lines of coding. This may be true, but another point of view is also possible: We may consider the main program and the subroutine as a *team* of programs, each member of the team having a certain job to do. The main program, in the course of doing its job, will activate the subprogram; the subprogram will perform its own function and then activate the main program. We might stretch our imagination to believe that, from the subroutine's point of view, when it exits *it* is calling the *main* routine; the main routine continues to perform its duty, then "exits" to the subroutine. The subroutine acts, then calls the main routine again.

This egalitarian philosophy may sound far-fetched, but it actually rings true with respect to coroutines. There is no way to distinguish which of two coroutines is subordinate to the other. Suppose a program consists of coroutines A and B; when programming A, we may think of B as our subroutine, but when programming B, we may think of A as our subroutine. Whenever a coroutine is activated, it resumes execution of its program at the point where the action was last suspended.

The coroutines A and B might, for example, be two programs that play chess. We can combine them so that they will play against each other.

Such coroutine linkage is easy to achieve with MMIX if we set aside two global registers, **a** and **b**. In coroutine A, the instruction "GO a,b,0" is used to

activate coroutine B; in coroutine B, the instruction “`G0 b,a,0`” is used to activate coroutine A. This scheme requires only $3v$ of time to transfer control each way.

The essential difference between routine-subroutine and coroutine-coroutine linkage can be seen by comparing the `G0`-type linkage of the previous section with the present scheme: A subroutine is always initiated *at its beginning*, which is usually a fixed place; the main routine or a coroutine is always initiated *at the place following* where it last terminated.

Coroutines arise most naturally in practice when they are connected with algorithms for input and output. For example, suppose it is the duty of coroutine A to read a file and to perform some transformation on the input, reducing it to a sequence of items. Another coroutine, which we will call B, does further processing of those items, and outputs the answers; B will periodically call for the successive input items found by A. Thus, coroutine B jumps to A whenever it wants the next input item, and coroutine A jumps to B whenever an input item has been found. The reader may say, “Well, B is the main program and A is merely a *subroutine* for doing the input.” This, however, becomes less true when the process A is very complicated; indeed, we can imagine A as the main routine and B as a subroutine for doing the output, and the above description remains valid. The usefulness of the coroutine idea emerges midway between these two extremes, when both A and B are complicated and each one calls the other in numerous places. It is not easy to find short, simple examples of coroutines that illustrate the importance of the idea; the most useful coroutine applications are generally quite lengthy.

In order to study coroutines in action, let us consider a contrived example. Suppose we want to write a program that translates one code into another. The input code to be translated is a sequence of 8-bit characters terminated by a period, such as

(1)

```
a2b5e3426fg0zyw3210pq89r.
```

This code appears on the standard input file, interspersed with whitespace characters in an arbitrary fashion. For our purposes a “whitespace character” will be any byte whose value is less than or equal to #20, the ASCII code for ‘ ’. All whitespace characters in the input are ignored; the other characters should be interpreted as follows, when they are read in sequence: (1) If the next character is one of the decimal digits 0 or 1 or ⋯ or 9, say n , it indicates $(n+1)$ repetitions of the following character, whether the following character is a digit or not. (2) A nondigit simply denotes itself. The output of our program is to consist of the resulting sequence separated into groups of three characters each, until a period appears; the last group may have fewer than three characters. For example, (1) should be translated into

(2)

```
abb bee eee e44 446 66f gzy w22 220 0pq 999 999 999 r.
```

Notice that `3426f` does not mean 3427 repetitions of the letter `f`; it means 4 fours and 3 sixes followed by `f`. If the input sequence is ‘1.’, the output is simply ‘.’, not ‘..’, because the first period terminates the output. The goal of

our program is to produce a sequence of lines on the standard output file, with 16 three-character groups per line (except, of course, that the final line might be shorter). The three-character groups should be separated by blank spaces, and each line should end as usual with the ASCII newline character #a.

To accomplish this translation, we will write two routines and a subroutine. The program begins by giving symbolic names to three global registers, one for temporary storage and the others for coroutine linkage.

```
01 * An example of coroutines
02 t      IS $255 Temporary data of short duration
03 in    GREG 0 Address for resuming the first coroutine
04 out   GREG 0 Address for resuming the second coroutine |
```

The next step is to set aside the memory locations used for working storage.

```
05 * Input and output buffers
06          LOC Data_Segment
07          GREG @           Base address
08 OutBuf TETRA "           ",#a,0 (see exercise 3)
09 Period  BYTE  ','
10 InArgs  OCTA InBuf,1000
11 InBuf   LOC  #100 |
```

Now we turn to the program itself. The subroutine we need, called `NextChar`, is designed to find non-whitespace characters of the input, and to return the next such character:

```
12 * Subroutine for character input
13 inptr   GREG 0           (the current input position)
14 1H      LDA   t,InArgs   Fill the input buffer.
15          TRAP  0,Fgets,StdIn
16          LDA   inptr,InBuf Start at beginning of buffer.
17 0H      GREG  Period
18          CSN   inptr,t,0B If error occurred, read a ','.
19 NextChar LDBU $0,inptr,0 Fetch the next character.
20          INCL  inptr,1
21          BZ    $0,1B       Branch if at end of buffer.
22          CMPU  t,$0,','
23          BNP   t,NextChar Branch if character is whitespace.
24          POP   1,0         Return to caller. |
```

This subroutine has the following characteristics:

Calling sequence: `PUSHJ $R,NextChar`.

Entry conditions: `inptr` points to the first unread character.

Exit conditions: `$R` = next non-whitespace character of input;
`inptr` is ready for the next entry to `NextChar`.

The subroutine also changes register `t`, namely register `$255`; but we usually omit that register from such specifications, as we did in 1.4.1'-(10).

Our first coroutine, called **In**, finds the characters of the input code with the proper replication. It begins initially at location **In1**:

```

25 * First coroutine
26 count    GREG 0          (the repetition counter)
27 1H      GO   in,out,0   Send a character to the Out coroutine.
28 In1     PUSHJ $0,NextChar Get a new character.
29       CMPU t,$0,'9'      Branch if it exceeds '9'.
30       PBP   t,1B         Branch if it is less than '0'.
31       SUB   count,$0,'0' Get another character.
32       BN    count,1B      Branch if it is less than '0'.
33       PUSHJ $0,NextChar Get another character.
34 1H      GO   in,out,0   Send it to Out.
35       SUB   count,count,1 Decrease the repetition counter.
36       PBNN  count,1B      Repeat if necessary.
37       JMP   In1          Otherwise begin a new cycle. ■

```

This coroutine has the following characteristics:

Calling sequence (from **Out**): **GO out,in,0**.

Exit conditions (to **Out**): **\$0** = next input character with proper replication.

Entry conditions

(upon return): **\$0** unchanged from its value at exit.

Register **count** is private to **In** and need not be mentioned.

The other coroutine, called **Out**, puts the code into three-character groups and sends them to the standard output file. It begins initially at **Out1**:

```

38 * Second coroutine
39 outptr   GREG 0          (the current output position)
40 1H      LDA   t,OutBuf   Empty the output buffer.
41       TRAP  0,Fputs,StdOut
42 Out1    LDA   outptr,OutBuf Start at beginning of buffer.
43 2H      GO   out,in,0   Get a new character from In.
44       STBU $0,outptr,0   Store it as the first of three.
45       CMP   t,$0,'.'      Branch if it was '.'.
46       BZ    t,1F         Otherwise get another character.
47       GO   out,in,0   Get a new character from In.
48       STBU $0,outptr,1   Store it as the second of three.
49       CMP   t,$0,'.'      Branch if it was '.'.
50       BZ    t,2F         Otherwise get another character.
51       GO   out,in,0   Get a new character from In.
52       STBU $0,outptr,2   Store it as the third of three.
53       CMP   t,$0,'.'      Branch if it was '.'.
54       BZ    t,3F         Otherwise advance to next group.
55       INCL  outptr,4
56 0H      GREG OutBuf+4*16
57       CMP   t,outptr,0B
58       PBNZ t,2B         Branch if fewer than 16 groups.
59       JMP   1B          Otherwise finish the line.

```

```

60 3H      INCL  outptr,1      Move past a stored character.
61 2H      INCL  outptr,1      Move past a stored character.
62 0H      GREG  #a          (newline character)
63 1H      STBU  OB,outptr,1  Store newline after period.
64 0H      GREG  0          (null character)
65           STBU  OB,outptr,2  Store null after newline.
66           LDA   t,OutBuf
67           TRAP  0,Fputs,StdOut Output the final line.
68           TRAP  0,Halt,0     Terminate the program. ■

```

The characteristics of `Out` are designed to complement those of `In`:

Calling sequence (from `In`): `G0 in,out,0`.

Exit conditions (to `In`): `$0` unchanged from its value at entry.

Entry conditions

(upon return): `$0 = next input character with proper replication.`

To complete the program, we need to get everything off to a good start. Initialization of coroutines tends to be a little tricky, although not really difficult.

```

69 * Initialization
70 Main    LDA   inptr,InBuf  Initialize NextChar.
71           GETA  in,In1       Initialize In.
72           JMP   Out1        Start with Out (see exercise 2). ■

```

This completes the program. The reader should study it carefully, noting in particular how each coroutine can be read and written independently as though the other coroutine were its subroutine.

We learned in Section 1.4.1' that MMIX's `PUSHJ` and `POP` instructions are superior to the `G0` command with respect to subroutine linkage. But with coroutines the opposite is true: Pushing and popping are quite unsymmetrical, and MMIX's register stack can get hopelessly entangled if two or more coroutines try to use it simultaneously. (See exercise 6.)

There is an important relation between coroutines and *multipass algorithms*. For example, the translation process we have just described could have been done in two distinct passes: We could first have done just the `In` coroutine, applying it to the entire input and writing each character with the proper amount of replication into an intermediate file. After this was finished, we could have read that file and done just the `Out` coroutine, taking the characters in groups of three. This would be called a "two-pass" process. (Intuitively, a "pass" denotes a complete scan of the input. This definition is not precise, and in many algorithms the number of passes taken is not at all clear; but the intuitive concept of "pass" is useful in spite of its vagueness.)

Figure 22(a) illustrates a four-pass process. Quite often we will find that the same process can be done in just one pass, as shown in part (b) of the figure, if we substitute four coroutines `A`, `B`, `C`, `D` for the respective passes `A`, `B`, `C`, `D`. Coroutine `A` will jump to `B` when pass `A` would have written an item of output on File 1; coroutine `B` will jump to `A` when pass `B` would have read an item of input from File 1, and `B` will jump to `C` when pass `B` would have written an item

of output on File 2; etc. UNIX® users will recognize this as a “pipe,” denoted by “PassA | PassB | PassC | PassD”. The programs for passes B, C, and D are sometimes referred to as “filters.”

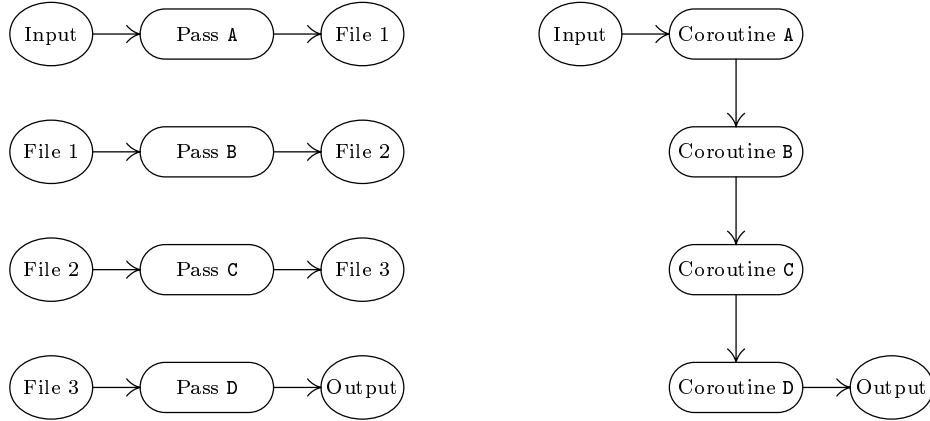


Fig. 22. Passes: (a) a four-pass algorithm, and (b) a one-pass algorithm.

Conversely, a process done by n coroutines can often be transformed into an n -pass process. Due to this correspondence it is worthwhile to compare multipass algorithms with one-pass algorithms.

a) *Psychological difference.* A multipass algorithm is generally easier to create and to understand than a one-pass algorithm for the same problem. A process that has been broken into a sequence of small steps, which happen one after the other, is easier to comprehend than an involved process in which many transformations take place simultaneously.

Also, if a very large problem is being tackled and if many people are supposed to cooperate in producing a computer program, a multipass algorithm provides a natural way to divide up the job.

These advantages of a multipass algorithm are present in coroutines as well, since each coroutine can be written essentially separate from the others. The linkage makes an apparently multipass algorithm into a single-pass process.

b) *Time difference.* The time required to pack, write, read, and unpack the intermediate data that flows between passes (for example, the information in the files of Fig. 22) is avoided in a one-pass algorithm. For this reason, a one-pass algorithm will be faster.

c) *Space difference.* The one-pass algorithm requires space to hold all the programs in memory simultaneously, while a multipass algorithm requires space for only one at a time. This requirement may affect the speed, even to a greater extent than indicated in statement (b). For example, many computers have a limited amount of “fast memory” and a larger amount of slower memory; if each

pass just barely fits into the fast memory, the result will be considerably faster than if we use coroutines in a single pass (since the use of coroutines would presumably force most of the program to appear in the slower memory or to be repeatedly swapped in and out of fast memory).

Occasionally there is a need to design algorithms for several computer configurations at once, some of which have larger memory capacity than others. In such cases it is possible to write the program in terms of coroutines, and to let the memory size govern the number of passes: Load together as many coroutines as feasible, and supply input or output subroutines for the missing links.

Although this relationship between coroutines and passes is important, we should keep in mind that coroutine applications cannot always be split into multipass algorithms. If coroutine B gets input from A and also sends back crucial information to A, as in the example of chess play mentioned earlier, the sequence of actions can't be converted into pass A followed by pass B.

Conversely, it is clear that some multipass algorithms cannot be converted to coroutines. Some algorithms are inherently multipass; for example, the second pass may require cumulative information from the first pass, like the total number of occurrences of a certain word in the input. There is an old joke worth noting in this regard:

Little old lady, riding a bus. “Little boy, can you tell me how to get off at Pasadena Street?”

Little boy. “Just watch me, and get off two stops before I do.”

(The joke is that the little boy gives a two-pass algorithm.)

So much for multipass algorithms. Coroutines also play an important role in discrete system simulation; see Section 2.2.5. When several more-or-less independent coroutines are controlled by a master process, they are often called *threads* of a computation. We will see further examples of coroutines in numerous places throughout this series of books. The important idea of *replicated coroutines* is discussed in Chapter 8, and some interesting applications of this idea may be found in Chapter 10.

EXERCISES

1. [10] Explain why short, simple examples of coroutines are hard for the author of a textbook to find.
- ▶ 2. [20] The program in the text starts up the `Out` coroutine first. What would happen if `In` were the first to be executed instead—that is, if lines 71 and 72 were changed to “`GETA out,Out1; JMP In1`”?
3. [15] Explain the `TETRA` instruction on line 08 of the program in the text. (There are exactly fifteen blank spaces between the double-quote marks.)
4. [20] Suppose two coroutines A and B want to treat MMIX's remainder register `rR` as if it were their private property, although both coroutines do division. (In other words, when one coroutine jumps to the other, it wants to be able to assume that the contents of `rR` will not have been altered when the other coroutine returns.) Devise a coroutine linkage that allows them this freedom.

5. [20] Could MMIX do reasonably efficient coroutine linkage by using its PUSH and POP instructions, without any GO commands?
6. [20] The program in the text uses MMIX's register stack only in a very limited way, namely when In calls NextChar. Discuss to what extent two cooperating routines could both make use of the register stack.
- 7. [30] Write an MMIX program that *reverses* the translation done by the program in the text. That is, your program should convert a file containing three-character groups like (2) into a file containing code like (1). The output should be as short a string of characters as possible, except for newlines; thus, for example, the zero before the z in (1) would not really be produced from (2).

1.4.3'. Interpretive Routines

In this section we will investigate a common type of program known as an *interpretive routine*, often called an *interpreter* for short. An interpretive routine is a computer program that performs the instructions of another program, where the other program is written in some machine-like language. By a machine-like language, we mean a way of representing instructions, where the instructions typically have operation codes, addresses, etc. (This definition, like most definitions of today's computer terms, is not precise, nor should it be; we cannot draw the line exactly and say just which programs are interpreters and which are not.)

Historically, the first interpreters were built around machine-like languages designed specially for simple programming; such languages were easier to use than a real machine language. The rise of symbolic languages for programming soon eliminated the need for interpretive routines of that kind, but interpreters have by no means begun to die out. On the contrary, their use has continued to grow, to the extent that an effective use of interpretive routines may be regarded as one of the essential characteristics of modern programming. The new applications of interpreters are made chiefly for the following reasons:

- a) a machine-like language is able to represent a complicated sequence of decisions and actions in a compact, efficient manner; and
- b) such a representation provides an excellent way to communicate between passes of a multipass process.

In such cases, special purpose machine-like languages are developed for use in a particular program, and programs in those languages are often generated only by computers. (Today's expert programmers are also good machine designers: They not only create an interpretive routine, they also define a *virtual machine* whose language is to be interpreted.)

The interpretive technique has the further advantage of being relatively machine-independent, since only the interpreter must be revised when changing computers. Furthermore, helpful debugging aids can readily be built into an interpretive system.

Examples of type (a) interpreters appear in several places later in this series of books; see, for example, the recursive interpreter in Chapter 8 and the "Parsing

Machine" in Chapter 10. We typically need to deal with situations in which a great many special cases arise, all similar, but having no really simple pattern.

For example, consider writing an algebraic compiler in which we want to generate efficient machine-language instructions that add two quantities together. There might be ten classes of quantities (constants, simple variables, subscripted variables, fixed or floating point, signed or unsigned, etc.) and the combination of all pairs yields 100 different cases. A long program would be required to do the proper thing in each case. The interpretive solution to this problem is to make up an ad hoc language whose "instructions" fit in one byte. Then we simply prepare a table of 100 "programs" in this language, where each program ideally fits in a single word. The idea is then to pick out the appropriate table entry and to perform the program found there. This technique is simple and efficient.

An example interpreter of type (b) appears in the article "Computer-Drawn Flowcharts" by D. E. Knuth, *CACM* **6** (1963), 555–563. In a multipass program, the earlier passes must transmit information to the later passes. This information is often transmitted most efficiently in a machine-like language, as a set of instructions for the later pass; the later pass is then nothing but a special purpose interpretive routine, and the earlier pass is a special purpose "compiler." This philosophy of multipass operation may be characterized as *telling* the later pass what to do, whenever possible, rather than simply presenting it with a lot of facts and asking it to *figure out* what to do.

Another example of a type-(b) interpreter occurs in connection with compilers for special languages. If the language includes many features that are not easily done on the machine except by subroutine, the resulting object programs will be very long sequences of subroutine calls. This would happen, for example, if the language were concerned primarily with multiple precision arithmetic. In such a case the object program would be considerably shorter if it were expressed in an interpretive language. See, for example, the book *ALGOL 60 Implementation*, by B. Randell and L. J. Russell (New York: Academic Press, 1964), which describes a compiler to translate from ALGOL 60 into an interpretive language, and which also describes the interpreter for that language; and see "An ALGOL 60 Compiler," by Arthur Evans, Jr., *Ann. Rev. Auto. Programming* **4** (1964), 87–124, for examples of interpretive routines used *within* a compiler. The rise of microprogrammed machines and of special-purpose integrated circuit chips has made this interpretive approach even more valuable.

The *T_EX* program, which produced the pages of the book you are now reading, converted a file that contained the text of this section into an interpretive language called DVI format, designed by D. R. Fuchs in 1979. [See D. E. Knuth, *T_EX: The Program* (Reading, Mass.: Addison-Wesley, 1986), Part 31.] The DVI file that *T_EX* produced was then processed by an interpreter called *dvips*, written by T. G. Rokicki, and converted to a file of instructions in another interpretive language called PostScript® [Adobe Systems Inc., *PostScript Language Reference*, 3rd edition (Reading, Mass.: Addison-Wesley, 1999)]. The PostScript file was sent to the publisher, who sent it to a commercial printer, who used a PostScript interpreter to produce printing plates. This three-pass

operation illustrates interpreters of type (b); *TeX* itself also includes a small interpreter of type (a) to process the so-called ligature and kerning information for characters that are being printed [*TeX: The Program*, §545].

There is another way to look at a program written in interpretive language: It may be regarded as a series of subroutine calls, one after another. Such a program may in fact be expanded into a long sequence of calls on subroutines, and, conversely, such a sequence can usually be packed into a coded form that is readily interpreted. The advantages of interpretive techniques are the compactness of representation, the machine independence, and the increased diagnostic capability. An interpreter can often be written so that the amount of time spent in interpretation of the code itself and branching to the appropriate routine is negligible.

***An MMIX simulator.** When the language presented to an interpretive routine is the machine language of another computer, the interpreter is often called a *simulator* (or sometimes an *emulator*).

In the author's opinion, entirely too much programmers' time has been spent in writing such simulators and entirely too much computer time has been wasted in using them. The motivation for simulators is simple: A computer installation buys a new machine and still wants to run programs written for the old machine (rather than rewriting the programs). However, this usually costs more and gives poorer results than if a special task force of programmers were given temporary employment to do the reprogramming. For example, the author once participated in such a reprogramming project, and a serious error was discovered in the original program, which had been in use for several years; the new program worked at five times the speed of the old, besides giving the right answers for a change! (Not all simulators are bad; for example, it is usually advantageous for a computer manufacturer to simulate a new machine before it has been built, so that software for the new machine may be developed as soon as possible. But that is a very specialized application.) An extreme example of the inefficient use of computer simulators is the true story of machine *A* simulating machine *B* running a program that simulates machine *C*. This is the way to make a large, expensive computer give poorer results than its cheaper cousin.

In view of all this, why should such a simulator rear its ugly head in this book? There are three reasons:

- a) The simulator we will describe below is a good example of a typical interpretive routine; the basic techniques employed in interpreters are illustrated here. It also illustrates the use of subroutines in a moderately long program.
- b) We will describe a simulator of the MMIX computer, written in (of all things) the MMIX language. This will reinforce our knowledge of the machine. It also will facilitate the writing of MMIX simulators for other computers, although we will not plunge deeply into the details of 64-bit integer or floating point arithmetic.
- c) Our simulation of MMIX explains how the register stack can be implemented efficiently in hardware, so that pushing and popping are accomplished with very little work. Similarly, the simulator presented here clarifies the `SAVE` and `UNSAVE` operators, and it provides details about the behavior of trip interrupts. Such

things are best understood by looking at a reference implementation, so that we can see how the machine really works.

Computer simulators as described in this section should be distinguished from *discrete system simulators*. Discrete system simulators are important programs that will be discussed in Section 2.2.5.

Now let's turn to the task of writing an MMIX simulator. We begin by making a tremendous simplification: Instead of attempting to simulate all the things that happen simultaneously in a pipelined computer, we will interpret only one instruction at a time. Pipeline processing is extremely instructive and important, but it is beyond the scope of this book; interested readers can find a complete program for a full-fledged pipeline “meta-simulator” in the *MMIXware* document. We will content ourselves here with a simulator that is blithely unaware of such things as cache memory, virtual address translation, dynamic instruction scheduling, reorder buffers, etc., etc. Moreover, we will simulate only the instructions that ordinary MMIX user programs can do; privileged instructions like LDVTS, which are reserved for the operating system, will be considered erroneous if they arise. Trap interrupts will not be simulated by our program unless they perform rudimentary input or output as described in Section 1.3.2’.

The input to our program will be a binary file that specifies the initial contents of memory, just as the memory would be set up by an operating system when running a user program (including command line data). We want to mimic the behavior of MMIX’s hardware, pretending that MMIX itself is interpreting the instructions that begin at symbolic location `Main`; thus, we want to implement the specifications that were laid down in Section 1.3.1’, in the run-time environment that was discussed in Section 1.3.2’. Our program will, for example, maintain an array of 256 octabytes $g[0], g[1], \dots, g[255]$ for the simulated global registers. The first 32 elements of this array will be the special registers listed in Table 1.3.1’–2; one of those special registers will be the simulated clock, `rC`. We will assume that each instruction takes a fixed amount of time, as specified by Table 1.3.1’–1; the simulated `rC` will increase by 2^{32} for each μ and by 1 for each v . Thus, for example, after we have simulated Program 1.3.2’P, the simulated `rC` will contain `#0000 3228 000b b091`, which represents $12840\mu + 766097v$.

The program is rather long, but it has many points of interest and we will study it in short easy pieces. It begins as usual by defining a few symbols and by specifying the contents of the data segment. We put the array of 256 simulated global registers first in that segment; for example, the simulated \$255 will be the octabyte $g[255]$, in memory location `Global+8*255`. This global array is followed by a similar array called the *local register ring*, where we will keep the top items of the simulated register stack. The size of this ring is set to 256, although 512 or any higher power of 2 would also work. (A large ring of local registers costs more, but it might be noticeably faster when a program uses the register stack heavily. One of the purposes of a simulator is to find out whether additional hardware would be worth the expense.) The main portion of the data segment, starting at `Chunk0`, will be devoted to the simulated memory.

```

001 * MMIX Simulator (Simplified)
002 t      IS  $255          Volatile register for temporary info
003 lring_size IS 256       Size of the local register ring
004      LOC  Data_Segment  Start at location #2000 0000 0000 0000
005 Global  LOC  @+8*256   256 octabytes for global registers
006 g      GREG Global    Base address for globals
007 Local   LOC  @+8*lring_size lring_size octabytes for local registers
008 l      GREG Local    Base address for locals
009          GREG @        Base address for IOArgs and Chunk0
010 IOArgs  OCTA 0,BinaryRead (See exercise 20)
011 Chunk0  IS  @        Beginning of simulated memory area
012          LOC  #100      Put everything else in the text segment. ■

```

One of the key subroutines we will need is called `MemFind`. Given a 64-bit address A , this subroutine returns the resulting address R where the simulated contents of $M_8[A]$ can be found. Of course 2^{64} bytes of simulated memory cannot be squeezed into a 2^{61} -byte data segment; but the simulator remembers all addresses that have occurred before, and it assumes that all locations not yet encountered are equal to zero.

Memory is divided into “chunks” of 2^{12} bytes each. `MemFind` looks at the leading $64 - 12 = 52$ bits of A to see what chunk it belongs to, and extends the list of known chunks, if necessary. Then it computes R by adding the trailing 12 bits of A to the starting address of the relevant simulated chunk. (The chunk size could be any power of 2, as long as each chunk contains at least one octabyte. Small chunks cause `MemFind` to search through longer lists of chunks-in-hand; large chunks cause `MemFind` to waste space for bytes that will never be accessed.)

Each simulated chunk is encapsulated in a “node,” which occupies $2^{12} + 24$ bytes of memory. The first octabyte of such a node, called the `KEY`, identifies the simulated address of the first byte in the chunk. The second octabyte, called the `LINK`, points to the next node on `MemFind`’s list; it is zero on the last node of the list. The `LINK` is followed by 2^{12} bytes of simulated memory called the `DATA`. Finally, each node ends with eight all-zero bytes, which are used as padding in the implementation of input-output (see exercises 15–17).

`MemFind` maintains its list of chunk nodes in order of use: The first node, pointed to by `head`, is the one that `MemFind` found on the previous call, and it links to the next-most-recently-used chunk, etc. If the future is like the past, `MemFind` will therefore not have to search far down its list. (Section 6.1 discusses such “self-organizing” list searches in detail.) Initially `head` points to `Chunk0`, whose `KEY` and `LINK` and `DATA` are all zero. The allocation pointer `alloc` is set initially to the place where the next chunk node will appear when it is needed, namely `Chunk0+nodesize`.

We implement `MemFind` with the `PREFIX` operation of `MMIXAL` discussed in Section 1.4.1’, so that the private symbols `head`, `key`, `addr`, etc., will not conflict with any symbols in the rest of the program. The calling sequence will be

SET arg,A; PUSHJ res,MemFind (1)

after which the resulting address R will appear in register `res`.

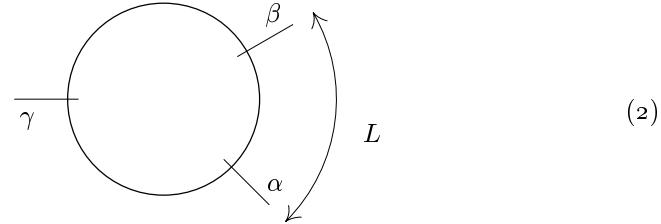
013	PREFIX :Mem:	(Begin private symbols for MemFind)
014	head GREG 0	Address of first chunk
015	curkey GREG 0	KEY(head)
016	alloc GREG 0	Address of next chunk to allocate
017	Chunk IS #1000	Bytes per chunk, must be a power of 2
018	addr IS \$0	The given address A
019	key IS \$1	Its chunk address
020	test IS \$2	Temporary register for key search
021	newlink IS \$3	The second most recently used node
022	p IS \$4	Temporary pointer register
023	t IS :t	External temporary register
024	KEY IS 0	
025	LINK IS 8	
026	DATA IS 16	
027	nodesize GREG Chunk+3*8	
028	mask GREG Chunk-1	
029	:MemFind ANDN key,addr,mask	
030	CMPU t,key,curkey	
031	PBZ t,4F	Branch if head is the right chunk.
032	BN addr,:Error	Disallow negative addresses A .
033	SET newlink,head	Prepare for the search loop.
034	1H SET p,head	$p \leftarrow \text{head}$.
035	LDOU head,p,LINK	$\text{head} \leftarrow \text{LINK}(p)$.
036	PBNZ head,2F	Branch if $\text{head} \neq 0$.
037	SET head,alloc	Otherwise allocate a new node.
038	STOU key,head,KEY	
039	ADDU alloc,alloc,nodesize	
040	JMP 3F	
041	2H LDOU test,head,KEY	
042	CMPU t,test,key	
043	BNZ t,1B	Loop back if $\text{KEY}(\text{head}) \neq \text{key}$.
044	3H LDOU t,head,LINK	Adjust pointers: $t \leftarrow \text{LINK}(\text{head})$,
045	STOU newlink,head,LINK	$\text{LINK}(\text{head}) \leftarrow \text{newlink}$,
046	SET curkey,key	$\text{curkey} \leftarrow \text{key}$,
047	STOU t,p,LINK	$\text{LINK}(p) \leftarrow t$.
048	SUBU t,addr,key	$t \leftarrow \text{chunk offset}$.
049	LDA \$0,head,DATA	$\$0 \leftarrow \text{address of DATA}(\text{head})$.
050	ADDU \$0,t,\$0	
051	POP 1,0	Return R .
052	PREFIX :	(End of the ':Mem:' prefix)
053	res IS \$2	Result register for PUSHJ
054	arg IS res+1	Argument register for PUSHJ

We come next to the most interesting aspect of the simulator, the implementation of MMIX's register stack. Recall from Section 1.4.1' that the register stack is conceptually a list of τ items $S[0], S[1], \dots, S[\tau - 1]$. The final item $S[\tau - 1]$ is said to be at the "top" of the stack, and MMIX's local registers $\$0, \$1, \dots, \$L-1$ are the topmost L items $S[\tau - L], S[\tau - L + 1], \dots, S[\tau - 1]$; here L is the value of special register rL. We could simulate the stack by simply keeping

it entirely in the simulated memory; but an efficient machine wants its registers to be instantly accessible, not in a relatively slow memory unit. Therefore we will simulate an efficient design that keeps the topmost stack items in an array of internal registers called the *local register ring*.

The basic idea is quite simple. Suppose the local register ring has ρ elements, $l[0], l[1], \dots, l[\rho - 1]$. Then we keep local register $\$k$ in $l[(\alpha + k) \bmod \rho]$, where α is an appropriate offset. (The value of ρ is chosen to be a power of 2, so that remainders mod ρ require no expensive computation. Furthermore we want ρ to be at least 256, so that there is room for all of the local registers.) A **PUSH** operation, which renames the local registers so that what once was, say, $\$3$ is now called $\$0$, simply increases the value of α by 3; a **POP** operation restores the previous state by decreasing α . Although the registers change their numbers, no data actually needs to be pushed down or popped up.

Of course we need to use memory as a backup when the register stack gets large. The status of the ring at any time is best visualized in terms of three variables, α , β , and γ :



Elements $l[\alpha], l[\alpha + 1], \dots, l[\beta - 1]$ of the ring are the current local registers $\$0, \$1, \dots, \$L - 1$; elements $l[\beta], l[\beta + 1], \dots, l[\gamma - 1]$ are currently unused; and elements $l[\gamma], l[\gamma + 1], \dots, l[\alpha - 1]$ contain items of the register stack that have been pushed down. If $\gamma \neq \alpha$, we can increase γ by 1 if we first store $l[\gamma]$ in memory. If $\gamma \neq \beta$, we can decrease γ by 1 if we then load $l[\gamma]$. MMIX has two special registers called the *stack pointer* rS and the *stack offset* rO, which hold the memory addresses where $l[\gamma]$ and $l[\alpha]$ will be stored, if necessary. The values of α , β , and γ are related to rL, rS, and rO by the formulas

$$\alpha = (rO/8) \bmod \rho, \quad \beta = (\alpha + rL) \bmod \rho, \quad \gamma = (rS/8) \bmod \rho. \quad (3)$$

The simulator keeps most of MMIX's special registers in the first 32 positions of the global register array. For example, the simulated remainder register rR is the octabyte in location **Global+8*rR**. But eight of the special registers, including rS, rO, rL, and rG, are potentially relevant to every simulated instruction, so the simulator maintains them separately in its own global registers. Thus, for example, register ss holds the simulated value of rS, and register 11 holds eight times the simulated value of rL:

055	ss	GREG	0	The simulated stack pointer, rS
056	oo	GREG	0	The simulated stack offset, rO
057	11	GREG	0	The simulated local threshold register, rL, times 8
058	gg	GREG	0	The simulated global threshold register, rG, times 8

```

059 aa GREG 0 The simulated arithmetic status register, rA
060 ii GREG 0 The simulated interval counter, rI
061 uu GREG 0 The simulated usage counter, rU
062 cc GREG 0 The simulated cycle counter, rC ■

```

Here is a subroutine that obtains the current value of the simulated register \$*k*, given *k*. The calling sequence is

SLU arg,*k*,3; PUSHJ res,GetReg (4)

after which the desired value will be in res.

```

063 lring_mask GREG 8*lring_size-1
064 :GetReg CMPU t,$0,gg Subroutine to get $k:
065 BN t,1F Branch if k < G.
066 LDOU $0,g,$0 Otherwise $k is global; load g[k].
067 POP 1,0 Return the result.
068 1H CMPU t,$0,11 t ← [k is local].
069 ADDU $0,$0,oo
070 AND $0,$0,lring_mask
071 LDOU $0,1,$0 Load l[( $\alpha + k$ ) mod  $\rho$ ].
072 CSNN $0,t,0 Zero it if $k is marginal.
073 POP 1,0 Return the result. ■

```

Notice the colon in the label field of line 064. This colon is redundant, because the current prefix is ‘:’ (see line 052); the colon on line 029 was, however, necessary for the external symbol MemFind, because at that time the current prefix was ‘:Mem:’. Colons in the label field, redundant or not, give us a handy way to advertise the fact that a subroutine is being defined.

The next subroutines, StackStore and StackLoad, simulate the operations of increasing γ by 1 and decreasing γ by 1 in the diagram (2). They return no result. StackStore is called only when $\gamma \neq \alpha$; StackLoad is called only when $\gamma \neq \beta$. Both of them must save and restore rJ, because they are not leaf subroutines.

```

074 :StackStore GET $0,rJ Save the return address.
075 AND t,ss,lring_mask
076 LDOU $1,1,t $1 ← l[ $\gamma$ ].
077 SET arg,ss
078 PUSHJ res,MemFind
079 STOU $1,res,0  $M_8[rS] \leftarrow \$1$ .
080 ADDU ss,ss,8 Increase rS by 8.
081 PUT rJ,$0 Restore the return address.
082 POP 0 Return to caller.

083 :StackLoad GET $0,rJ Save the return address.
084 SUBU ss,ss,8 Decrease rS by 8.
085 SET arg,ss
086 PUSHJ res,MemFind
087 LDOU $1,res,0 $1 ←  $M_8[rS]$ .
088 AND t,ss,lring_mask

```

```

089      STOU $1,1,t          l[γ] ← $1.
090      PUT   rJ,$0          Restore the return address.
091      POP   0              Return to caller. ■

```

(Register rJ on lines 074, 081, 083, and 090 is, of course, the *real* rJ, not the simulated rJ. When we simulate a machine on itself, we have to remember to keep such things straight!)

The **StackRoom** subroutine is called when we have just increased β . It checks whether $\beta = \gamma$ and, if so, it increases γ .

```

092 :StackRoom SUBU t,ss,oo
093      SUBU t,t,11
094      AND   t,t,lring_mask
095      PBNZ t,1F          Branch if (rS-rO)/8  $\not\equiv$  rL (modulo  $\rho$ ).
096      GET   $0,rJ          Oops, we're not a leaf subroutine.
097      PUSHJ res,StackStore Advance rS.
098      PUT   rJ,$0          Restore the return address.
099 1H      POP   0              Return to caller. ■

```

Now we come to the heart of the simulator, its main simulation loop. An interpretive routine generally has a central control section that is called into action between interpreted instructions. In our case, the program transfers to location **Fetch** when it is ready to simulate a new command. We keep the address @ of the next simulated instruction in the global register **inst_ptr**. **Fetch** usually sets **loc** \leftarrow **inst_ptr** and advances **inst_ptr** by 4; but if we are simulating a **RESUME** command that inserts the simulated rX into the instruction stream, **Fetch** sets **loc** \leftarrow **inst_ptr** - 4 and leaves **inst_ptr** unchanged. This simulator considers an instruction to be ineligible for execution unless its location **loc** is in the text segment (that is, **loc** < #2000 0000 0000 0000).

```

100 * The main loop
101 loc      GREG 0  Where the simulator is at
102 inst_ptr GREG 0  Where the simulator will be next
103 inst    GREG 0  The current instruction being simulated
104 resuming GREG 0  Are we resuming an instruction in rX?
105 Fetch  PBZ  resuming,1F  Branch if not resuming.
106      SUBU loc,inst_ptr,4  loc ← inst_ptr - 4.
107      LDTU inst,g,8*rX+4  inst ← right half of rX.
108      JMP   2F
109 1H      SET   loc,inst_ptr  loc ← inst_ptr.
110      SET   arg,loc
111      PUSHJ res,MemFind
112      LDTU inst,res,0      inst ← M4[loc].
113      ADDU inst_ptr,loc,4  inst_ptr ← loc + 4.
114 2H      CMPU t,loc,g
115      BNN   t,Error        Branch if loc  $\geq$  Data_Segment. ■

```

The main control routine does the things common to all instructions. It unpacks the current instruction into its various parts and puts the parts into

convenient registers for later use. Most importantly, it sets global register *f* to 64 bits of “info” corresponding to the current opcode. A master table, which starts at location *Info*, contains such information for each of MMIX’s 256 opcodes. (See Table 1 on page 88.) For example, *f* is set to an odd value if and only if the *Z* field of the current opcode is an “immediate” operand or the opcode is *JMP*; similarly *f* \wedge #40 is nonzero if and only if the instruction has a relative address. Later steps of the simulator will be able to decide quickly what needs to be done with respect to the current instruction because most of the relevant information appears in register *f*.

```

116  op      GREG  0  Opcode of the current instruction
117  xx      GREG  0  X field of the current instruction
118  yy      GREG  0  Y field of the current instruction
119  zz      GREG  0  Z field of the current instruction
120  yz      GREG  0  YZ field of the current instruction
121  f       GREG  0  Packed information about the current opcode
122  xxx     GREG  0  X field times 8
123  x       GREG  0  X operand and/or result
124  y       GREG  0  Y operand
125  z       GREG  0  Z operand
126  xptr    GREG  0  Location where x should be stored
127  exc     GREG  0  Arithmetic exceptions
128  Z_is_immed_bit IS #1  Flag bits possibly set in f
129  Z_is_source_bit IS #2
130  Y_is_immed_bit IS #4
131  Y_is_source_bit IS #8
132  X_is_source_bit IS #10
133  X_is_dest_bit  IS #20
134  Rel_addr_bit   IS #40
135  Mem_bit        IS #80
136  Info  IS  #1000
137  Done   IS  Info+8*256
138  info   GREG  Info          (Base address for the master info table)
139  c255   GREG  8*255        (A handy constant)
140  c256   GREG  8*256        (Another handy constant)
141      MOR   op,inst,#8  op ← inst  $\gg$  24.
142      MOR   xx,inst,#4  xx ← (inst  $\gg$  16)  $\wedge$  #ff.
143      MOR   yy,inst,#2  yy ← (inst  $\gg$  8)  $\wedge$  #ff.
144      MOR   zz,inst,#1  zz ← inst  $\wedge$  #ff.
145  OH GREG -#10000
146      ANDN  yz,inst,0B
147      SLU   xxx,xx,3
148      SLU   t,op,3
149      LDOU  f,info,t   f ← Info[op].
150      SET   x,0        x ← 0 (default value).
151      SET   y,0        y ← 0 (default value).
152      SET   z,0        z ← 0 (default value).
153      SET   exc,0      exc ← 0 (default value). ■

```

The first thing we do, after having unpacked the instruction into its various fields, is convert a relative address to an absolute address if necessary.

```

154      AND    t,f,Rel_addr_bit
155      PBZ    t,1F           Branch if not a relative address.
156      PBEV   f,2F           Branch if op isn't JMP or JMPC.
157  9H GREG -#1000000
158      ANDN   yz,inst,9B    yz ← inst ∧ #ffffff (namely XYZ).
159      ADDU   t,yz,9B       t ← XYZ − 224.
160      JMP    3F
161  2H ADDU   t,yz,0B       t ← YZ − 216.
162  3H CSOD   yz,op,t     Set yz ← t if op is odd ("backward").
163      SL     t,yz,2
164      ADDU   yz,loc,t     yz ← loc + yz ≪ 2. ■

```

The next task is critical for most instructions: We install the operands specified by the Y and Z fields into global registers y and z. Sometimes we also install a third operand into global register x, specified by the X field or coming from a special register like the simulated rD or rM.

```

165  1H          PBNN   resuming,Install_X  Branch unless resuming < 0.
                  ...
174  Install_X  AND    t,f,X_is_source_bit
175          PBZ    t,1F           Branch unless $X is a source.
176          SET    arg,xxx
177          PUSHJ  res,GetReg
178          SET    x,res         x ← $X.
179  1H          SRU    t,f,5
180          AND    t,t,#f8       t ← special register number, times 8.
181          PBZ    t,Install_Z
182          LDOU   x,g,t        If t ≠ 0, set x ← g[t].
183  Install_Z   AND    t,f,Z_is_source_bit
184          PBZ    t,1F           Branch unless $Z is a source.
185          SLU    arg,zz,3
186          PUSHJ  res,GetReg
187          SET    z,res         z ← $Z.
188          JMP    Install_Y
189  1H          CSOD   z,f,zz        If Z is immediate, z ← Z.
190          AND    t,op,#f0
191          CMPU   t,t,#e0
192          PBNZ   t,Install_Y  Branch unless # e0 ≤ op < # f0.
193          AND    t,op,#3
194          NEG    t,3,t
195          SLU    t,t,4
196          SLU    z,yz,t        z ← yz ≪ (48, 32, 16, or 0).
197          SET    y,x           y ← x.
198  Install_Y   AND    t,f,Y_is_immed_bit
199          PBZ    t,1F           Branch unless Y is immediate.
200          SET    y,yy
201          SLU    t,yy,40
202          ADDU   f,f,t        Insert Y into left half of f.

```

```

203 1H      AND    t,f,Y_is_source_bit
204  BZ    t,1F          Branch unless $Y is a source.
205  SLU   arg,yy,3
206  PUSHJ res,GetReg
207  SET    y,res        y ← $Y. ■

```

When the X field specifies a destination register, we set `xptr` to the memory address where we will eventually store the simulated result; this address will be either in the `Global` array or the `Local` ring. The simulated register stack grows at this point if the destination register must be changed from marginal to local.

```

208 1H      AND    t,f,X_is_dest_bit
209  BZ    t,1F          Branch unless $X is a destination.
210 XDest  CMPU  t,xxx,gg
211  BN    t,3F          Branch if $X is not global.
212  LDA   xptr,g,xxx    xptr ← address of g[X].
213  JMP   1F
214 2H      ADDU  t,oo,11
215  AND   t,t,lring_mask
216  STCO  0,1,t        l[( $\alpha + L$ ) mod  $\rho$ ] ← 0.
217  INCL  11,8          $L \leftarrow L + 1$ . ($L becomes local.)
218  PUSHJ res,StackRoom  Make sure  $\beta \neq \gamma$ .
219 3H      CMPU  t,xxx,11
220  BNN   t,2B          Branch if $X is not local.
221  ADD   t,xxx,oo
222  AND   t,t,lring_mask
223  LDA   xptr,l,t      xptr ← address of l[( $\alpha + X$ ) mod  $\rho$ ]. ■

```

Finally we reach the climax of the main control cycle: We simulate the current instruction by essentially doing a 256-way branch, based on the current opcode. The left half of register `f` is, in fact, an MMIX instruction that we *perform* at this point, by inserting it into the instruction stream via a `RESUME` command. For example, if we are simulating an `ADD` command, we put “`ADD x,y,z`” into the right half of `rX` and clear the exception bits of `rA`; the `RESUME` command will then cause the sum of registers `y` and `z` to be placed in register `x`, and `rA` will record whether overflow occurred. After the `RESUME`, control will pass to location `Done`, unless the inserted instruction was a branch or jump.

```

224 1H  AND    t,f,Mem_bit
225  PBZ   t,1F          Branch unless inst accesses memory.
226  ADDU  arg,y,z
227  CMPU  t,op,#AO     t ← [op is a load instruction].
228  BN    t,2F
229  CMPU  t,arg,g
230  BN    t>Error       Error if storing into the text segment.
231 2H  PUSHJ res,MemFind  res ← address of M[y + z].
232 1H  SRU   t,f,32
233  PUT   rX,t          rX ← left half of f.
234  PUT   rM,x          rM ← x (prepare for MUX).
235  PUT   rE,x          rE ← x (prepare for FCMPE, FUNE, FEQLE).

```

```

236 OH GREG #30000
237     AND t,aa,OB      t ← current rounding mode.
238     ORL t,U_BIT<<8   Enable underflow trip (see below).
239     PUT rA,t          Prepare rA for arithmetic.
240 OH GREG Done
241     PUT rW,OB          rW ← Done.
242     RESUME O           Execute the instruction in rX. ■

```

Some instructions can't be simulated by simply "performing themselves" like an ADD command and jumping to Done. For example, a MULU command must insert the high half of its computed product into the simulated rH. A branch command must change `inst_ptr` if the branch is taken. A PUSHJ command must push the simulated register stack, and a POP command must pop it. SAVE, UNSAVE, RESUME, TRAP, etc., all need special care; therefore the next part of the simulator deals with all cases that don't fit the nice " x equals y op z " pattern.

Let's start with multiplication and division, since they are easy:

```

243 MulU MULU x,y,z      Multiply y by z, unsigned.
244     GET t,rH            Set t ← upper half of the product.
245     STOU t,g,8*rH       g[rH] ← upper half product.
246     JMP XDone           Finish by storing x.
247 Div DIV x,y,z
    ...                   (For division, see exercise 6.) ■

```

If the simulated instruction was a branch command, say "BZ \$X,RA", the main control routine will have converted the relative address RA to an absolute address in register yz (line 164), and it will also have placed the contents of the simulated \$X into register x (line 178). The RESUME command will then execute the instruction "BZ x,BTaken" (line 242); and control will pass to BTaken instead of Done if the simulated branch is taken. BTaken adds $2v$ to the simulated running time, changes `inst_ptr`, and jumps to Update.

```

254 BTaken ADDU cc,cc,4    Increase rC by 4v.
255 PBTaken SUBU cc,cc,2    Decrease rC by 2v.
256     SET inst_ptr,yz    inst_ptr ← branch address.
257     JMP Update         Finish the command.
258 Go     SET x,inst_ptr   GO instruction: Set x ← loc + 4.
259     ADDU inst_ptr,y,z  inst_ptr ← (y + z) mod  $2^{64}$ .
260     JMP XDone           Finish by storing x. ■

```

(Line 257 could have jumped to Done, but that would be slower; a shortcut to Update is justified because a branch command doesn't store x and cannot cause an arithmetic exception. See lines 500–541 below.)

A PUSHJ or PUSHGO command pushes the simulated register stack down by increasing the α pointer of (2); this means increasing the simulated rO, namely register oo. If the command is "PUSHJ \$X,RA" and if \$X is local, we push $X + 1$ octabytes down by first setting $$X \leftarrow X$ and then increasing oo by $8(X + 1)$. (The value we have put in \$X will be used later by POP to determine how to restore oo to its former value. Simulated register \$X will then be set to the

result of the subroutine, as explained in Section 1.4.1'.) If \$X is global, we push $rL + 1$ octabytes down in a similar way.

```

261 PushGo ADDU yz,y,z      yz  $\leftarrow (y + z) \bmod 2^{64}$ .
262 PushJ SET inst_ptr,yz    inst_ptr  $\leftarrow yz$ .
263 CMPU t,xxx,gg
264 PBN t,1F                 Branch if $X is local.
265 SET xxx,11                Pretend that X = rL.
266 SRU xx,xxx,3
267 INCL 11,8                 Increase rL by 1.
268 PUSHJ 0,StackRoom        Make sure  $\beta \neq \gamma$  in (2).
269 1H ADDU t,xxx,oo
270 AND t,t,lring_mask
271 STOU xx,1,t                $l[(\alpha + X) \bmod \rho] \leftarrow X$ .
272 ADDU t,loc,4
273 STOU t,g,8*rJ              $g[rJ] \leftarrow loc + 4$ .
274 INCL xxx,8
275 SUBU 11,11,xxx            Decrease rL by X + 1.
276 ADDU oo,oo,xxx            Increase rO by 8(X + 1).
277 JMP Update                Finish the command. ■

```

Special routines are needed also to simulate POP, SAVE, UNSAVE, and several other opcodes including RESUME. Those routines deal with interesting details about MMIX, and we will consider them in the exercises; but we'll skip them for now, since they do not involve any techniques related to interpretive routines that we haven't seen already.

We might as well present the code for SYNC and TRIP, however, since those routines are so simple. (Indeed, there's nothing to do for "SYNC XYZ" except to check that $XYZ \leq 3$, since we aren't simulating cache memory.) Furthermore, we will take a look at the code for TRAP, which is interesting because it illustrates the important technique of a jump table for multiway switching:

```

278 Sync   BNZ xx,Error       Branch if X  $\neq 0$ .
279          CMPU t,yz,4
280          BNN t>Error        Branch if YZ  $\geq 4$ .
281          JMP Update        Finish the command.
282 Trip   SET xx,0           Initiate a trip to location 0.
283          JMP TakeTrip      (See exercise 13.)
284 Trap   STOU inst_ptr,g,8*rWW g[rWW]  $\leftarrow inst\_ptr$ .
285 OH GREG #8000000000000000
286          ADDU t,inst,OB
287          STOU t,g,8*rXX     g[rXX]  $\leftarrow inst + 2^{63}$ .
288          STOU y,g,8*rYY     g[rYY]  $\leftarrow y$ .
289          STOU z,g,8*rZZ     g[rZZ]  $\leftarrow z$ .
290          SRU y,inst,6
291          CMPU t,y,4*11
292          BNN t>Error        Branch if X  $\neq 0$  or Y  $> Ftell$ .
293          LDOU t,g,c255      t  $\leftarrow g[255]$ .

```

294	OH GREG @+4		
295	GO	y,0B,y	Jump to @ + 4 + 4Y.
296	JMP	SimHalt	Y = Halt: Jump to SimHalt.
297	JMP	SimFopen	Y = Fopen: Jump to SimFopen.
298	JMP	SimFclose	Y = Fclose: Jump to SimFcclose.
299	JMP	SimFread	Y = Fread: Jump to SimFread.
300	JMP	SimFgets	Y = Fgets: Jump to SimFgets.
301	JMP	SimFgetws	Y = Fgetws: Jump to SimFgetws.
302	JMP	SimFwrite	Y = Fwrite: Jump to SimFwrite.
303	JMP	SimFputs	Y = Fputs: Jump to SimFputs.
304	JMP	SimFputws	Y = Fputws: Jump to SimFputws.
305	JMP	SimFseek	Y = Fseek: Jump to SimFseek.
306	JMP	SimFtell	Y = Ftell: Jump to SimFtell.
307	TrapDone	STO t,g,8*rBB	Set g[rBB] \leftarrow t.
308		STO t,g,c255	A trap ends with g[255] \leftarrow g[rBB].
309		JMP Update	Finish the command. ■

(See exercises 15–17 for `SimFopen`, `SimFcclose`, `SimFread`, etc.)

Now let's look at the master `Info` table (Table 1), which allows the simulator to deal rather painlessly with 256 different opcodes. Each table entry is an octabyte consisting of (i) a four-byte MMIX instruction, which will be invoked by the `RESUME` instruction on line 242; (ii) two bytes that define the simulated running time, one byte for μ and one byte for v ; (iii) a byte that names a special register, if such a register ought to be loaded into `x` on line 182; and (iv) a byte that is the sum of eight 1-bit flags, expressing special properties of the opcode. For example, the info for opcode `FIX` is

```
FIX x,0,z; BYTE 0,4,0,#26;
```

it means that (i) the instruction `FIX x,0,z` should be performed, to round a floating point number to a fixed point integer; (ii) the simulated running time should be increased by $0\mu + 4v$; (iii) no special register is needed as an input operand; and (iv) the flag byte

```
#26 = X_is_dest_bit + Y_is_immed_bit + Z_is_source_bit
```

determines the treatment of registers `x`, `y`, and `z`. (The `Y_is_immed_bit` actually causes the `Y` field of the simulated instruction to be inserted into the `Y` field of “`FIX x,0,z`”; see line 202.)

One interesting aspect of the `Info` table is that the `RESUME` command of line 242 executes the instruction as if it were in location `Done-4`, since `rW = Done`. Therefore, if the instruction is a `JMP`, the address must be relative to `Done-4`; but MMIXAL always assembles `JMP` commands with an address relative to the assembled location `@`. We trick the assembler into doing the right thing by writing, for example, “`JMP Trap+@-0`”, where `0` is defined to equal `Done-4`. Then the `RESUME` command will indeed jump to location `Trap` as desired.

After we have executed the special instruction inserted by `RESUME`, we normally get to location `Done`. From here on everything is anticlimactic; but

Table 1
MASTER INFORMATION TABLE FOR SIMULATOR CONTROL

0 IS Done-4		LDB x,res,0; BYTE 1,1,0,#aa	(LDB)
LOC Info		LDB x,res,0; BYTE 1,1,0,#a9	(LDBI)
JMP Trap+@-0; BYTE 0,5,0,#0a	(TRAP)	...	
FCMP x,y,z; BYTE 0,1,0,#2a	(FCMP)	JMP Cswap+@-0; BYTE 2,2,0,#ba	(CSWAP)
FUN x,y,z; BYTE 0,1,0,#2a	(FUN)	JMP Cswap+@-0; BYTE 2,2,0,#b9	(CSWAPI)
FEQL x,y,z; BYTE 0,1,0,#2a	(FEQL)	LDUNC x,res,0; BYTE 1,1,0,#aa	(LDUNC)
FADD x,y,z; BYTE 0,4,0,#2a	(FADD)	LDUNC x,res,0; BYTE 1,1,0,#a9	(LDUNCI)
FIX x,0,z; BYTE 0,4,0,#26	(FIX)	JMP Error+@-0; BYTE 0,1,0,#2a	(LDVTS)
FSUB x,y,z; BYTE 0,4,0,#2a	(FSUB)	JMP Error+@-0; BYTE 0,1,0,#29	(LDVTSI)
FIXU x,0,z; BYTE 0,4,0,#26	(FIXU)	SWYM 0; BYTE 0,1,0,#0a	(PRELD)
FLOT x,0,z; BYTE 0,4,0,#26	(FLOT)	SWYM 0; BYTE 0,1,0,#09	(PRELDI)
FLOT x,0,z; BYTE 0,4,0,#25	(FLOTI)	SWYM 0; BYTE 0,1,0,#0a	(PREGO)
FLOTU x,0,z; BYTE 0,4,0,#26	(FLOTU)	SWYM 0; BYTE 0,1,0,#09	(PREGOI)
...		JMP Go+@-0; BYTE 0,3,0,#2a	(GO)
FMUL x,y,z; BYTE 0,4,0,#2a	(FMUL)	JMP Go+@-0; BYTE 0,3,0,#29	(GOI)
FCMPE x,y,z; BYTE 0,4,rE,#2a	(FCMPE)	STB x,res,0; BYTE 1,1,0,#9a	(STB)
FUNE x,y,z; BYTE 0,1,rE,#2a	(FUNE)	STB x,res,0; BYTE 1,1,0,#99	(STBI)
FEQLE x,y,z; BYTE 0,4,rE,#2a	(FEQLE)	...	
FDIV x,y,z; BYTE 0,40,0,#2a	(FDIV)	STO xx,res,0; BYTE 1,1,0,#8a	(STCO)
FSQRT x,0,z; BYTE 0,40,0,#26	(FSQRT)	STO xx,res,0; BYTE 1,1,0,#89	(STCOI)
FREM x,y,z; BYTE 0,4,0,#2a	(FREM)	STUNC x,res,0; BYTE 1,1,0,#9a	(STUNC)
FINT x,0,z; BYTE 0,4,0,#26	(FINT)	STUNC x,res,0; BYTE 1,1,0,#99	(STUNCI)
MUL x,y,z; BYTE 0,10,0,#2a	(MUL)	SWYM 0; BYTE 0,1,0,#0a	(SYNCD)
MUL x,y,z; BYTE 0,10,0,#29	(MULI)	SWYM 0; BYTE 0,1,0,#09	(SYNCDI)
JMP Mulu+@-0; BYTE 0,10,0,#2a	(MULU)	SWYM 0; BYTE 0,1,0,#0a	(PREST)
JMP Mulu+@-0; BYTE 0,10,0,#29	(MULUI)	SWYM 0; BYTE 0,1,0,#09	(PRESTI)
JMP Div+@-0; BYTE 0,60,0,#2a	(DIV)	SWYM 0; BYTE 0,1,0,#0a	(SYNCID)
JMP Div+@-0; BYTE 0,60,0,#29	(DIVI)	SWYM 0; BYTE 0,1,0,#09	(SYNCIDI)
JMP DivU+@-0; BYTE 0,60,rD,#2a	(DIVU)	JMP PushGo+@-0; BYTE 0,3,0,#2a	(PUSHGO)
JMP DivU+@-0; BYTE 0,60,rD,#29	(DIVUI)	JMP PushGo+@-0; BYTE 0,3,0,#29	(PUSHGOI)
ADD x,y,z; BYTE 0,1,0,#2a	(ADD)	OR x,y,z; BYTE 0,1,0,#2a	(OR)
ADD x,y,z; BYTE 0,1,0,#29	(ADDI)	OR x,y,z; BYTE 0,1,0,#29	(ORI)
ADDU x,y,z; BYTE 0,1,0,#2a	(ADDU)	...	
...		SET x,z; BYTE 0,1,0,#20	(SETH)
CMPU x,y,z; BYTE 0,1,0,#29	(CMPUI)	SET x,z; BYTE 0,1,0,#20	(SETMH)
NEG x,0,z; BYTE 0,1,0,#26	(NEG)	...	
NEG x,0,z; BYTE 0,1,0,#25	(NEGI)	ANDN x,x,z; BYTE 0,1,0,#30	(ANDNL)
NEGU x,0,z; BYTE 0,1,0,#26	(NEGU)	SET inst_ptr,yz; BYTE 0,1,0,#41	(JMP)
NEGU x,0,z; BYTE 0,1,0,#25	(NEGUI)	SET inst_ptr,yz; BYTE 0,1,0,#41	(JMPB)
SL x,y,z; BYTE 0,1,0,#2a	(SL)	JMP PushJ+@-0; BYTE 0,1,0,#60	(PUSHJ)
...		JMP PushJ+@-0; BYTE 0,1,0,#60	(PUSHJB)
BN x,BTaken+@-0; BYTE 0,1,0,#50	(BN)	SET x,yz; BYTE 0,1,0,#60	(GETA)
BN x,BTaken+@-0; BYTE 0,1,0,#50	(BNB)	SET x,yz; BYTE 0,1,0,#60	(GETAB)
BZ x,BTaken+@-0; BYTE 0,1,0,#50	(BZ)	JMP Put+@-0; BYTE 0,1,0,#02	(PUT)
...		JMP Put+@-0; BYTE 0,1,0,#01	(PUTI)
PBNP x,PBTaken+@-0; BYTE 0,3,0,#50	(PBNPB)	JMP Pop+@-0; BYTE 0,3,rJ,#00	(POP)
PBEV x,PBTaken+@-0; BYTE 0,3,0,#50	(PBEV)	JMP Resume+@-0; BYTE 0,5,0,#00	(RESUME)
PBEV x,PBTaken+@-0; BYTE 0,3,0,#50	(PBEVB)	JMP Save+@-0; BYTE 20,1,0,#20	(SAVE)
CSN x,y,z; BYTE 0,1,0,#3a	(CSN)	JMP Unsave+@-0; BYTE 20,1,0,#02	(UNSAVE)
CSN x,y,z; BYTE 0,1,0,#39	(CSNI)	JMP Sync+@-0; BYTE 0,1,0,#01	(SYNC)
...		SWYM x,y,z; BYTE 0,1,0,#00	(SWYM)
ZSEV x,y,z; BYTE 0,1,0,#2a	(ZSEV)	JMP Get+@-0; BYTE 0,1,0,#20	(GET)
ZSEV x,y,z; BYTE 0,1,0,#29	(ZSEVI)	JMP Trip+@-0; BYTE 0,5,0,#0a	(TRIP)

Entries not shown here explicitly follow a pattern that is easily deduced from the examples shown. (See, for example, exercise 1.)

we can take satisfaction in the fact that an instruction has been simulated successfully and the current cycle is nearly finished. Only a few details still need to be wrapped up: We must store the result x in the appropriate place, if the `X_is_dest_bit` flag is present, and we must check if an arithmetic exception has triggered a trip interrupt:

```

500 Done    AND   t,f,X_is_dest_bit
501      BZ    t,1F          Branch unless $X is a destination.
502 XDone   STOU  x,xptr,0      Store x in simulated $X.
503 1H      GET   t,rA
504      AND   t,t,#ff        t ← new arithmetic exceptions.
505      OR    exc,exc,t     exc ← exc ∨ t.
506      AND   t,exc,U_BIT+X_BIT
507      CMPU  t,t,U_BIT
508      PBNZ  t,1F          Branch unless underflow is exact.
509 OH GREG U_BIT<<8
510      AND   t,aa,0B
511      BNZ   t,1F          Branch if underflow is enabled.
512      ANDNL exc,U_BIT    Ignore U if exact and not enabled.
513 1H      PBZ   exc,Update
514      SRU   t,aa,8
515      AND   t,t,exc
516      PBZ   t,4F          Branch unless trip interrupt needed.
...           (See exercise 13.)
539 4H      OR    aa,aa,exc  Record new exceptions in rA. ■

```

Line number 500 is used here for convenience, although several hundred instructions and the entire Info table actually intervene between line 309 and this part of the program. Incidentally, the label `Done` on line 500 does not conflict with the label `Done` on line 137, because both of them define the same equivalent value for this symbol.

After line 505, register `exc` contains the bit codes for all arithmetic exceptions triggered by the instruction just simulated. At this point we must deal with a curious asymmetry in the rules for IEEE standard floating point arithmetic: An underflow exception (U) is suppressed unless the underflow trip has been enabled in `rA` or unless an inexact exception (X) has also occurred. (We had to enable the underflow trip in line 238 for precisely this reason; the simulator ends with the commands

```
LOC U_Handler;  ORL exc,U_BIT;  JMP Done (5)
```

so that `exc` will properly record underflow exceptions in cases where a floating point computation was exact but produced a denormal result.)

Finally — Hurray! — we are able to close the cycle of operations that began long ago at location `Fetch`. We update the runtime clocks, take a deep breath, and return to `Fetch` again:

```

540 OH GREG #0000000800000004
541 Update   MOR   t,f,0B          232mems + oops

```

542	ADDU	cc,cc,t	Increase the simulated clock, rC.
543	ADDU	uu,uu,1	Increase the usage counter, rU.
544	SUBU	ii,ii,1	Decrease the interval counter, rI.
545	AllDone	PBZ resuming,Fetch	Go to Fetch if resuming = 0.
546		CMPU t,op,#F9	Otherwise set t \leftarrow [op=RESUME].
547		CSNZ resuming,t,0	Clear resuming if not resuming,
548	JMP	Fetch	and go to Fetch. ■

Our simulation program is now complete, except that we still must initialize everything properly. We assume that the simulator will be run with a command line that names a binary file. Exercise 20 explains the simple format of that file, which specifies what should be loaded into the simulated memory before simulation begins. Once the program has been loaded, we launch it as follows: At line 576 below, register loc will contain a location from which a simulated UNSAVE command will get the program off to a good start. (In fact, we simulate an UNSAVE that is being simulated by a simulated RESUME. The code is tricky, perhaps, but it works.)

549	Infile	IS 3	(Handle for binary input file)
550	Main	LDA Mem:head,Chunk0	Initialize MemFind.
551		ADDU Mem:alloc,Mem:head,Mem:nodesize	
552		GET t,rN	
553		INCL t,1	
554		STOU t,g,8*rN	$g[rN] \leftarrow (our\ rN) + 1$.
555		LDOU t,\$1,8	$t \leftarrow$ binary file name ($argv[1]$).
556		STOU t,IOArgs	
557		LDA t,IOArgs	(See line 010)
558		TRAP 0,Fopen,Infile	Open the binary file.
559		BN t,Error	
	...		Now load the file (see exercise 20).
576		STOU loc,g,c255	$g[255] \leftarrow$ place to UNSAVE.
577		SUBU arg,loc,8*13	arg \leftarrow place where \$255 appears.
578		PUSHJ res,MemFind	
579		LDOU inst_ptr,res,0	inst_ptr \leftarrow Main.
580		SET arg,#90	
581		PUSHJ res,MemFind	
582		LDTU x,res,0	$x \leftarrow M_4[#90]$.
583		SET resuming,1	resuming $\leftarrow 1$.
584		CSNZ inst_ptr,x,#90	If $x \neq 0$, set inst_ptr \leftarrow #90.
585	OH	GREG #FB<<24+255	
586		STOU OB,g,8*rX	$g[rX] \leftarrow$ "UNSAVE \$255".
587		SET gg,c255	$G \leftarrow 255$.
588		JMP Fetch	Start the ball rolling.
589	Error	NEG t,22	$t \leftarrow -22$ for error exit.
590	Exit	TRAP 0,Halt,0	End of simulation.
591	LOC Global+8*rK;	OCTA -1	
592	LOC Global+8*rT;	OCTA #8000000500000000	
593	LOC Global+8*rTT;	OCTA #8000000600000000	
594	LOC Global+8*rV;	OCTA #369c200400000000	■

The simulated program's `Main` starting address will be in the simulated register \$255 after the simulated `UNSAVE`. Lines 580–584 of this code implement a feature that wasn't mentioned in Section 1.3.2: If an instruction is loaded into location #90, the program begins there instead of at `Main`. (This feature allows a subroutine library to initialize itself before starting a user program at `Main`.)

Lines 591–594 initialize the simulated `rK`, `rT`, `rTT`, and `rV` to appropriate constant values. Then the program is finished; it ends with the trip-handler instructions of (5).

Whew! Our simulator has turned out to be pretty long—longer, in fact, than any other program that we will encounter in this book. But in spite of its length, the program above is incomplete in several respects because the author did not want to make it even longer:

- a) Several parts of the code have been left as exercises.
- b) The program simply branches to `Error` and quits, when it detects a problem. A decent simulator would distinguish between different types of error, and would have a way to keep going.
- c) The program doesn't gather any statistics, except for the total running time (`cc`) and the total number of instructions simulated (`uu`). A more complete program would, for example, remember how often the user guessed correctly with respect to branches versus probable branches; it would also record the number of times the `StackLoad` and `StackStore` subroutines need to access simulated memory. It might also analyze its own algorithms, studying for example the efficiency of the self-organizing search technique used by `MemFind`.
- d) The program has no diagnostic facilities. A useful simulator would, for example, allow interactive debugging, and would output selected snapshots of the simulated program's execution; such features would not be difficult to add. The ability to monitor a program easily is, in fact, one of the main reasons for the importance of interpretive routines in general.

EXERCISES

1. [20] Table 1 shows the `Info` entries only for selected opcodes. What entries are appropriate for (a) opcode #3F (`SRUI`)? (b) opcode #55 (`PBPB`)? (c) opcode #D9 (`MUXI`)? (d) opcode #E6 (`INCML`)?
- ▶ 2. [26] How much time does it take the simulator to simulate the instructions (a) `ADDU $255,$Y,$Z`; (b) `STHT $X,$Y,0`; (c) `PBNZ $X,0-4`?
3. [23] Explain why $\gamma \neq \alpha$ when `StackRoom` calls `StackStore` on line 097.
- ▶ 4. [20] Criticize the fact that `MemFind` never checks to see if `alloc` has gotten too large. Is this a serious blunder?
- ▶ 5. [20] If the `MemFind` subroutine branches to `Error`, it does not pop the register stack. How many items might be on the register stack at such a time?
6. [20] Complete the simulation of `DIV` and `DIVU` instructions, by filling in the missing code of lines 248–253.
7. [21] Complete the simulation of `CSWAP` instructions, by writing appropriate code.

8. [22] Complete the simulation of **GET** instructions, by writing appropriate code.
 9. [23] Complete the simulation of **PUT** instructions, by writing appropriate code.
 10. [24] Complete the simulation of **POP** instructions, by writing appropriate code.
Note: If the normal action of **POP** as described in Section 1.4.1' would leave $rL > rG$, MMIX will pop entries off the top of the register stack so that $rL = rG$. For example, if the user pushes 250 registers down with **PUSHJ** and then says “**PUT rG,32; POP**”, only 32 of the pushed-down registers will survive.
 11. [25] Complete the simulation of **SAVE** instructions, by writing appropriate code.
Note: **SAVE** pushes all the local registers down and stores the entire register stack in memory, followed by $\$G, \$G+1, \dots, \$255$, followed by $rB, rD, rE, rH, rJ, rM, rR, rP, rW, rX, rY$, and rZ (in that order), followed by the octabyte $2^{56}rG + rA$.
 12. [26] Complete the simulation of **UNSAVE** instructions, by writing appropriate code.
Note: The very first simulated **UNSAVE** is part of the initial loading process (see lines 583–588), so it should not update the simulated clocks.
 13. [27] Complete the simulation of trip interrupts, by filling in the missing code of lines 517–538.
 14. [28] Complete the simulation of **RESUME** instructions, by writing appropriate code.
Note: When rX is nonnegative, its most significant byte is called the “ropcode”; ropcodes 0, 1, 2 are available for user programs. Line 242 of the simulator uses ropcode 0, which simply inserts the lower half of rX into the instruction stream. Ropcode 1 is similar, but the instruction in rX is performed with $y \leftarrow rY$ and $z \leftarrow rZ$ in place of the normal operands; this variant is allowed only when the first hexadecimal digit of the inserted opcode is #0, #1, #2, #3, #6, #7, #C, #D, or #E. Ropcode 2 sets $\$X \leftarrow rZ$ and $exc \leftarrow Q$, where X is the third byte from the right of rX and Q is the third byte from the left; this makes it possible to set the value of a register and simultaneously raise any subset of the arithmetic exceptions DVWIIOUZX. Ropcodes 1 and 2 can be used only when $\$X$ is not marginal. Your solution to this exercise should cause **RESUME** to set **resuming** $\leftarrow 0$ if the simulated rX is negative, otherwise **resuming** $\leftarrow (1, -1, -2)$ for ropcodes (0, 1, 2). You should also supply the code that is missing from lines 166–173.
- ▶ 15. [25] Write the routine **SimFputs**, which simulates the operation of outputting a string to the file corresponding to a given handle.
 - ▶ 16. [25] Write the routine **SimFopen**, which opens a file corresponding to a given handle. (The simulator can use the same handle number as the user program.)
 - ▶ 17. [25] Continuing the previous exercises, write the routine **SimFread**, which reads a given number of bytes from a file corresponding to a given handle.
 - ▶ 18. [21] Would this simulator be of any use if **lring_size** were less than 256, for example if **lring_size** = 32?
 - 19. [14] Study all the uses of the **StackRoom** subroutine (namely in line 218, line 268, and in the answer to exercise 11). Can you suggest a better way to organize the code? (See step 3 in the discussion at the end of Section 1.4.1'.)
 - 20. [20] The binary files input by the simulator consist of one or more groups of octabytes each having the simple form

$$\lambda, x_0, x_1, \dots, x_{t-1}, 0$$

for some $l \geq 0$, where x_0, x_1, \dots , and x_{l-1} are nonzero; the meaning is

$$M_8[\lambda + 8k] \leftarrow x_k, \quad \text{for } 0 \leq k < l.$$

The file ends after the last group. Complete the simulator by writing MMIX code to load such input (lines 560–575 of the program). The final value of register `loc` should be the location of the last octabyte loaded, namely $\lambda + 8(l - 1)$.

- 21. [20] Is the simulation program of this section able to simulate itself? If so, is it able to simulate itself simulating itself? And if so, is it ...?
- 22. [40] Implement an efficient *jump trace* routine for MMIX. This is a program that records all transfers of control in the execution of another given program by recording a sequence of pairs $(x_1, y_1), (x_2, y_2), \dots$, meaning that the given program jumped from location x_1 to y_1 , then (after performing the instructions in locations y_1, y_1+1, \dots, x_2) it jumped from x_2 to y_2 , etc. [From this information it is possible for a subsequent routine to reconstruct the flow of the program and to deduce how frequently each instruction was performed.]

A trace routine differs from a simulator because it allows the traced program to occupy its normal memory locations. A jump trace modifies the instruction stream in memory, but does so only to the extent necessary to retain control. Otherwise it allows the machine to execute arithmetic and memory instructions at full speed. Some restrictions are necessary; for example, the program being traced shouldn't modify itself. But you should try to keep such restrictions to a minimum.

ANSWERS TO EXERCISES

SECTION 1.3.1'

1. $\#7d9$ or $\#7D9$.

2. (a) $\{B, D, F, b, d, f\}$. (b) $\{A, C, E, a, c, e\}$. An odd fact of life.

3. (Solution by Gregor N. Purdy.) 2 bits = 1 nyp; 2 nyps = 1 nybble; 2 nybbles = 1 byte. Incidentally, the word “byte” was coined in 1956 by members of IBM’s Stretch computer project; see W. Buchholz, *BYTE* 2, 2 (February 1977), 144.

4. $1000 \text{ MB} = 1 \text{ gigabyte (GB)}$, $1000 \text{ GB} = 1 \text{ terabyte (TB)}$, $1000 \text{ TB} = 1 \text{ petabyte (PB)}$, $1000 \text{ PB} = 1 \text{ exabyte (EB)}$, $1000 \text{ EB} = 1 \text{ zettabyte (ZB)}$, $1000 \text{ ZB} = 1 \text{ yottabyte (YB)}$, according to the 19th Conférence Générale des Poids et Mesures (1990).

(Some people, however, use 2^{10} instead of 1000 in these formulas, claiming for example that a kilobyte is 1024 bytes. To resolve the ambiguity, such units should preferably be called *large kilobytes*, *large megabytes*, etc., and denoted by KKB, MMB, ... to indicate their binary nature.)

5. If $-2^{n-1} \leq x < 2^{n-1}$, then $-2^n < x - s(\alpha) < 2^n$; hence $x \neq s(\alpha)$ implies that $x \not\equiv s(\alpha) \pmod{2^n}$. But $s(\alpha) = u(\alpha) - 2^n[\alpha \text{ begins with } 1] \equiv u(\alpha) \pmod{2^n}$.

6. Using the notation of the previous exercise, we have $u(\bar{\alpha}) = 2^n - 1 - u(\alpha)$; hence $u(\bar{\alpha}) + 1 \equiv -u(\alpha) \pmod{2^n}$, and it follows that $s(\bar{\alpha}) + 1 = -s(\alpha)$. Overflow might occur, however, when adding 1. In that case $\alpha = 10\dots0$, $s(\alpha) = -2^{n-1}$, and $-s(\alpha)$ is not representable.

7. Yes. (See the discussion of shifting.)

8. The radix point now falls between rH and \$X. (In general, if the binary radix point is m positions from the end of \$Y and n positions from the end of \$Z, it is $m+n$ positions from the end of the product.)

9. Yes, except when $X = Y$, or $X = Z$, or overflow occurs.

10. $\$Y = \#8000000000000000$, $\$Z = \#fffffffffffffff$ is the *only* example!

11. (a) True, because $s($Y) \equiv u($Y)$ and $s($Z) \equiv u($Z)$ (modulo 2^{64}) by exercise 5.
(b) Clearly true if $s($Y) \geq 0$ and $s($Z) \geq 0$, because $s($Y) = u($Y)$ and $s($Z) = u($Z)$ in such a case. Also true if $$Z = 0$ or $$Z = 1$ or $$Z = Y or $$Y = 0$. Otherwise false.

12. If $X \neq Y$, say ‘`ADDU $X,$Y,$Z; CMPU carry,$X,$Y; ZSN carry,carry,1`’. But if $X = Y = Z$, say ‘`ZSN carry,$X,1; ADDU $X,$X,$X`’.

13. Overflow occurs on signed addition if and only if \$Y and \$Z have the same sign but their unsigned sum has the opposite sign. Thus

```
XOR $0,$Y,$Z; ADDU $X,$Y,$Z; XOR $1,$X,$Y; ANDN $1,$1,$0; ZSN ovfl,$1,1
```

determines the presence or absence of overflow when $X \neq Y$.

- 14.** Interchange X and Y in the previous answer. (Overflow occurs when computing $x = y - z$ if and only if it occurs when computing $y = x + z$.)
- 15.** Let \dot{y} and \dot{z} be the sign bits of y and z , so that $s(y) = y - 2^{64}\dot{y}$ and $s(z) = z - 2^{64}\dot{z}$; we want to calculate $s(y)s(z) \bmod 2^{128} = (yz - 2^{64}(\dot{y}\dot{z} + y\dot{z})) \bmod 2^{128}$. Thus the program MULU \$X,\$Y,\$Z; GET \$0,rH; ZSN \$1,\$Y,\$Z; SUBU \$0,\$0,\$1; ZSN \$1,\$Z,\$Y; SUBU \$0,\$0,\$1 puts the desired octabyte in \$0.
- 16.** After the instructions in the previous answer, check that the upper half is the sign extension of the lower half, by saying ‘SR \$1,\$X,63; CMP \$1,\$0,\$1; ZSNZ ovfl,\$1,1’.
- 17.** Let a be the stated constant, which is $(2^{65} + 1)/3$. Then $ay/2^{65} = y/3 + y/(3 \cdot 2^{65})$, so $\lfloor ay/2^{65} \rfloor = \lfloor y/3 \rfloor$ for $0 \leq y < 2^{65}$.
- 18.** By a similar argument, $\lfloor ay/2^{66} \rfloor = \lfloor y/5 \rfloor$ for $0 \leq y < 2^{66}$ when $a = (2^{66} + 1)/5 = \#cccccccccccccccd$.
- 19.** This statement is widely believed, and it has been implemented by compiler writers who did not check the math. But it is *false* when $z = 7, 21, 23, 25, 29, 31, 39, 47, 49, 53, 55, 61, 63, 71, 81, 89, \dots$, and in fact for 189 odd divisors z less than 1000! Let $\epsilon = ay/2^{64+e} - y/z = (z - r)y/(2^{64+e}z)$, where $r = 2^{64+e} \bmod z$. Then $0 < \epsilon < 2/z$, hence trouble can arise only when $y \equiv -1 \pmod{z}$ and $\epsilon \geq 1/z$. It follows that the formula $\lfloor ay/2^{64+e} \rfloor = \lfloor y/z \rfloor$ holds for all unsigned octabytes y , $0 \leq y < 2^{64}$, if and only if it holds for the single value $y = 2^{64} - 1 - (2^{64} \bmod z)$. (The formula is, however, always correct in the restricted range $0 \leq y < 2^{63}$. And Michael Yoder observes that high-multiplication by $\lceil 2^{64+e+1}/z \rceil - 2^{64}$, followed by addition of y and right-shift by $e + 1$, does work in general.)
- 20.** 4ADDU \$X,\$Y,\$Y; 4ADDU \$X,\$X,\$X.
- 21.** SL sets \$X to zero, overflowing if \$Y was nonzero. SLU and SRU set \$X to zero. SR sets \$X to 64 copies of the sign bit of \$Y, namely to $-[\$Y < 0]$. (Notice that shifting left by -1 does *not* shift right.)
- 22.** Dull’s program takes the wrong branch when the SUB instruction causes overflow. For example, it treats *every* nonnegative number as less than -2^{63} ; it treats $2^{63} - 1$ as less than every negative number. Although no error arises when \$1 and \$2 have the same sign, or when the numbers in \$1 and \$2 are both less than 2^{62} in absolute value, the correct formulation ‘CMP \$0,\$1,\$2; BN \$0,Case1’ is much better. (Similar errors have been made by programmers and compiler writers since the 1950s, often causing significant and mysterious failures.)
- 23.** CMP \$0,\$1,\$2; BNP \$0,Case1.
- 24.** ANDN.
- 25.** XOR \$X,\$Y,\$Z; SADD \$X,\$X,0.
- 26.** ANDN \$X,\$Y,\$Z.
- 27.** BDIF \$W,\$Y,\$Z; ADDU \$X,\$Z,\$W; SUBU \$W,\$Y,\$W.
- 28.** BDIF \$0,\$Y,\$Z; BDIF \$X,\$Z,\$Y; OR \$X,\$0,\$X.
- 29.** NOR \$0,\$Y,0; BDIF \$0,\$0,\$Z; NOR \$X,\$0,0. (This sequence computes $2^n - 1 - \max(0, (2^n - 1 - y) - z)$ in each byte position.)
- 30.** XOR \$1,\$0,\$2; BDIF \$1,\$3,\$1; SADD \$1,\$1,0 when \$2 = #20202020202020 and \$3 = #0101010101010101.
- 31.** MXOR \$1,\$4,\$0; SADD \$1,\$1,0 when \$4 = #0101010101010101.
- 32.** $C_{ji}^T = C_{ij} = (A_{1i}^T \bullet B_{j1}^T) \circ \dots \circ (A_{ni}^T \bullet B_{jn}^T) = (B^T \bullet A^T)_{ji}$ if \bullet is commutative.

33. MOR (or MXOR) with the constant #0180402010080402.

34. MOR \$X,\$Z,[#0080004000200010]; MOR \$Y,\$Z,[#0008000400020001]. (Here we use brackets to denote registers that contain auxiliary constants.)

To go back, also checking that an 8-bit code is sufficient:

```
PUT    rM, [#00ff00ff00ff]
MOR    $0,$X, [#4020100804020180]
MUX    $1,$0,$Y
BNZ    $1,BadCase
MUX    $1,$Y,$0
MOR    $Z,$1, [#8020080240100401] ■
```

35. MOR \$X,\$Y,\$Z; MOR \$X,\$Z,\$X; here \$Z is the constant (14).

36. XOR \$0,\$Y,\$Z; MOR \$0,[-1],\$0. *Notes:* Changing XOR to BDIF gives a mask for the bytes where \$Y exceeds \$Z. Given such a mask, AND it with #8040201008040201 and MOR with #ff to get a one-byte encoding of the relevant byte positions.

37. Let the elements of the field be polynomials in the Boolean matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

For example, this matrix is $m(\#402010080402018e)$, and if we square it with MXOR we get the matrix $m(\#2010080402018e47)$. The sum and product of such field elements are then obtained by XOR and MXOR, respectively.

(A field with 2^k elements for $2 \leq k \leq 7$ is obtained in a similar way from polynomials in the matrices #0103, #020105, #04020109, #0804020112, #100804020121, #20100804020141. Matrices of size up to 16×16 can be represented as four octabytes; then multiplication requires eight MXORs and four XORs. We can, however, do multiplication in a field of 2^{16} elements by performing only five MXORs and three XORs, if we represent the large field as a quadratic extension of the field of 2^8 elements.)

38. It sets \$1 to the sum of the eight signed bytes initially in \$0; it also sets \$2 to the rightmost nonzero such byte, or zero; and it sets \$0 to zero. (Changing SR to SRU would treat the bytes as unsigned. Changing SLU to SL would often overflow.)

39. The assumed running times are (a) ($3v$ or $2v$) versus $2v$; (b) ($4v$ or $3v$) versus $2v$; (c) ($4v$ or $3v$) versus $3v$; (d) (v or $4v$) versus $2v$; (e) ($2v$ or $5v$) versus $2v$; (f) ($2v$ or $5v$) versus $3v$. So we should use the conditional instructions in cases (a, d) and (c, f), unless \$0 is negative with probability $> 2/3$; in the latter case we should use the PBN variants, (d) and (f). The conditionals always win in cases (b, e).

If the ADDU commands had been ADD, the instructions would not have been equivalent, because of possible overflows.

40. Suppose you GO to address #101; this sets $@ \leftarrow \#101$. The tetrabyte $M_4[\#101]$ is the same as the tetrabyte $M_4[\#100]$. If the opcode of that instruction is, say, PUSHJ, register rJ will be set to #105. Similarly, if that instruction is GETA \$0,@, register \$0 will be set to #101. In such situations the value for @ in MMIX assembly language is slightly different from the actual value during program execution.

Programmers could use these principles to send some sort of signal to a subroutine, based on the two trailing bits of @. (Tricky, but hey, why not use the bits we've got?)

- 41.** (a) True. (b) True. (c) True. (d) False, but true with SRU in place of SR.
- 42.** (a) NEGU \$1,\$0; CSNN \$1,\$0,\$0. (b) ANDN \$1,\$0,[#8000000000000000].
- 43.** Trailing zeros (solution by J. Dallos): SUBU \$0,\$Z,1; SADD \$0,\$0,\$Z.
- Leading zeros: FLOTU \$0,1,\$Z; SRU \$0,\$0,52; SUB \$0,[1086],\$0. (If \$Z could be zero, add the command CSZ \$0,\$Z,64.) This is the shortest program, but not the fastest; we save $2v$ if we reverse all bits (exercise 35) and count *trailing* zeros.
- 44.** Use “high tetra arithmetic,” in which each 32-bit number appears in the *left* half of a register. LDHT and STHT load and store such quantities (see exercise 7); SETMH loads an immediate constant. To add, subtract, multiply, or divide high tetras \$Y and \$Z, producing a high tetra \$X with correct attention to integer overflow and divide check, the following commands work perfectly: (a) ADD \$X,\$Y,\$Z. (b) SUB \$X,\$Y,\$Z. (c) SR \$X,\$Z,32; MUL \$X,\$Y,\$X (assuming that we have $X \neq Y$). (d) DIV \$X,\$Y,\$Z; SL \$X,\$X,32; now rR is the high tetra remainder.

- 46.** It causes a trip to location 0.

47. #DF is MXORI (“multiple exclusive-or immediate”); #55 is PBPB (“probable branch positive backward”). But in a program we use the names MXOR and PBP; the assembler silently adds the I and B when required.

48. STO and STOU; also the “immediate” variants LDOI and LDOUI, STOI and STOUI; also NEGI and NEGUI, although NEG is not equivalent to NEGU; also any two of the four opcodes FLOTI, FLOTUI, SFLOTI, and SFLOTUI.

(Every MMIX operation on signed numbers has a corresponding operation on unsigned numbers, obtained by adding 2 to the opcode. This consistency makes the machine design easier to learn, the machine easier to build, and the compilers easier to write. But of course it also makes the machine less versatile, because it leaves no room for other operations that might be desired.)

49. Octabyte $M_8[0]$ is set to #0000010000000001; rH is set to #0000012343210000; $M_2[#024442000000122]$ is set to #0121; rA is set to #00041 (because overflow occurs on the STW); rB is set to f(7) = #401c000000000000; and $\$1 \leftarrow \#6ff8fffffffffffff$. (Also $rL \leftarrow 2$, if rL was originally 0 or 1.) We assume that the program is not located in such a place that the STCO, STB, or STW instructions could clobber it.

50. $4\mu + 34v = v + (\mu+v) + v + (\mu+v) + (\mu+v) + v + v + 10v + v + (\mu+v) + v + 4v + v + v + v + 3v + v + v + v$.

51.

35010001	a0010101	2e010101	a5010101	f6000001	c4010101
b5010101	8e010101	1a010101	db010101	c7010101	3d010101
33010101	e4010001	f7150001	08010001	5701ffff	3f010101

52. Opcodes ADDI, ADDUI, SUBI, SUBUI, SLI, SLUI, SRI, SRUI, ORI, XORI, ANDNI, BDIFI, WDIFI, TDIFI, ODIFI: $X = Y = 255$, $Z = 0$. Opcode MULI: $X = Y = 255$, $Z = 1$. Opcodes INCH, INCMH, INCML, INCL, ORH, ORMH, ORML, ORL, ANDNH, ANDNMH, ANDNML, ANDNL: $X = 255$, $Y = Z = 0$. Opcodes OR, AND, MUX: $X = Y = Z = 255$. Opcodes CSM, CSZ, ..., CSEV: $X = Z = 255$, Y arbitrary. Opcodes BN, BZ, ..., PBEV: X arbitrary, $Y = 0$, $Z = 1$. Opcode JMP: $X = Y = 0$, $Z = 1$. Opcodes PRELD, PRELDI, PREGO, PREGOI, SWYM: X , Y , Z arbitrary. (Subtle point: An instruction that sets register \$X is not a no-op when X is marginal, because it causes rL to increase; and all registers except \$255 are marginal when rL = 0 and rG = 255.)

53. MULU, MULUI, PUT, PUTI, UNSAVE.

54. FCMP, FADD, FIX, FSUB, . . . , FCMPE, FEQLE, . . . , FINT, MUL, MULI, DIV, DIVI, ADD, ADDI, SUB, SUBI, NEG, SL, SLI, STB, STBI, STW, STWI, STT, STTI, STSF, STSFI, PUT, PUTI, UNSAVE. (This was not quite a fair question, because the complete rules for floating point operations appear only elsewhere. One fine point is that FCMP might change the I_BIT of rA, if \$Y or \$Z is Not-a-Number, but FEQL and FUN never cause exceptions.)

55. FCMP, FUN, . . . , SRUI, CSN, CSNI, . . . , LDUNCI, GO, GOI, PUSHGO, PUSHGOI, OR, ORI, . . . , ANDNL, PUSHJ, PUSHJB, GETA, GETAB, PUT, PUTI, POP, SAVE, UNSAVE, GET.

56. <i>Minimum space:</i>	LDO \$1,x	MUL \$0,\$0,\$1
	SET \$0,\$1	SUB \$2,\$2,1
	SETL \$2,12	PBP \$2,0-4*2

Space = $6 \times 4 = 24$ bytes, time = $\mu + 149v$. Faster solutions are possible.

Minimum time: The assumption that $|x^{13}| \leq 2^{63}$ implies that $|x| < 2^5$ and $x^8 < 2^{39}$. The following solution, based on an idea of Y. N. Patt, exploits this fact.

LDO	\$0,x	$\$0 = x$
MUL	\$1,\$0,\$0	$\$1 = x^2$
MUL	\$1,\$1,\$1	$\$1 = x^4$
SL	\$2,\$1,25	$\$2 = 2^{25}x^4$
SL	\$3,\$0,39	$\$3 = 2^{39}x$
ADD	\$3,\$3,\$1	$\$3 = 2^{39}x + x^4$
MULU	\$1,\$3,\$2	$u(\$1) = 2^{25}x^8$, rH = $x^5 + 2^{25}x^4$ [$x < 0$]
GET	\$2,rH	$\$2 \equiv x^5$ (modulo 2^{25})
PUT	rM,[#1fffff]	
MUX	\$2,\$2,\$0	$\$2 = x^5$
SRU	\$1,\$1,25	$\$1 = x^8$
MUL	\$0,\$1,\$2	$\$0 = x^{13}$

Space = $12 \times 4 = 48$ bytes, time = $\mu + 48v$. At least five multiplications are “necessary,” according to the theory developed in Section 4.6.3; yet this program uses only four! And in fact there is a way to avoid multiplication altogether.

True minimum time: As R. W. Floyd points out, we have $|x| \leq 28$, so the minimum execution time is achieved by referring to a table (unless $\mu > 45v$):

LDO	\$0,x	$\$0 = x$
8ADDU	\$0,\$0,[Table]	
LDO	\$0,\$0,8*28	$\$0 = x^{13}$
...		
Table OCTA	-28*28	
OCTA	-27*27	
...		
OCTA	28*28	■

Space = $3 \times 4 + 57 \times 8 = 468$ bytes, time = $2\mu + 3v$.

57. (1) An operating system can allocate high-speed memory more efficiently if program blocks are known to be “read-only.” (2) An instruction cache in hardware will be faster and less expensive if instructions cannot change. (3) Same as (2), with “pipeline” in place of “cache.” If an instruction is modified after entering a pipeline, the pipeline needs to be flushed; the circuitry needed to check this condition is complex and time-consuming. (4) Self-modifying code cannot be used by more than one process at once. (5) Self-modifying code can defeat techniques for “profiling” (that is, for computing the number of times each instruction is executed).

SECTION 1.3.2'

1. (a) It refers to the label of line 24. (b) No indeed. Line 23 would refer to line 24 instead of line 38; line 31 would refer to line 24 instead of line 21.
2. The current value of **9B** will be a running count of the number of such lines that have appeared earlier.
3. Read in 100 octabytes from standard input; exchange their maximum with the last of them; exchange the maximum of the remaining 99 with the last of those; etc. Eventually the 100 octabytes will become completely sorted into nondecreasing order. The result is then written to the standard output. (Compare with Algorithm 5.2.3S.)
4. #2233445566778899. (Large values are reduced mod 2^{64} .)
5. BYTE "silly"; but this trick is not recommended.
6. False; TETRA @, @ is not the same as TETRA @; TETRA @.
7. He forgot that relative addresses are to tetrabyte locations; the two trailing bits are ignored.
8. LOC 16*((@+15)/16) or LOC -@/16*-16 or LOC (@+15)&-16, etc.
9. Change 500 to 600 on line 02; change Five to Six on line 35. (Five-digit numbers are not needed unless 1230 or more primes are to be printed. Each of the first 6542 primes will fit in a single wyde.)
10. $M_2[\#2000000000000000] = \#0002$, and the following nonzero data goes into the text segment:

#100: #e3 fe 00 03	#15c: #23 ff f6 00
#104: #c1 fb f7 00	#160: #00 00 07 01
#108: #a6 fe f8 fb	#164: #35 fa 00 02
#10c: #e7 fb 00 02	#168: #20 fa faf7
#110: #42 fb 00 13	#16c: #23 ff f6 1b
#114: #e7 fe 00 02	#170: #00 00 07 01
#118: #c1 fa f7 00	#174: #86 f9 f8 fa
#11c: #86 f9 f8 fa	#178: #af f5 f8 00
#120: #1c fd fe f9	#17c: #23 ff f8 04
#124: #fe fc 00 06	#180: #1d f9 f9 0a
#128: #43 fc fff b	#184: #fe fc 00 06
#12c: #30 ff fdf9	#188: #e7 fc 00 30
#130: #4dff fff6	#18c: #a3 fc ff 00
#134: #e7 fa 00 02	#190: #25 ff ff 01
#138: #f1 ff fff9	#194: #5b f9 fff b
#13c: #46 69 72 73	#198: #23 ff f8 00
#140: #74 20 46 69	#19c: #00 00 07 01
#144: #76 65 20 48	#1a0: #e7 fa 00 64
#148: #75 6e 64 72	#1a4: #51 fa fff4
#14c: #65 64 20 50	#1a8: #23 ff f6 19
#150: #72 69 6d 65	#1ac: #00 00 07 01
#154: #73 0a 00 20	#1b0: #31 ff fa 62
#158: #20 20 00 00	#1b4: #5b ff ffed

(Notice that SET becomes SETL in #100, but ORI in #104. The current location @ is aligned to #15c at line 38, according to rule 7(a).) When the program begins, rG will be #f5, and we will have \$248 = #200000000003e8, \$247 = #fffffffffffffc1a, \$246 = #13c, \$245 = #2030303030000000.

11. (a) If n is not prime, by definition n has a divisor d with $1 < d < n$. If $d > \sqrt{n}$, then n/d is a divisor with $1 < n/d < \sqrt{n}$. (b) If n is not prime, n has a prime divisor d with $1 < d \leq \sqrt{n}$. The algorithm has verified that n has no prime divisors $\leq p = \text{PRIME}[k]$; also $n = pq + r < pq + p \leq p^2 + p < (p + 1)^2$. Any prime divisor of n is therefore greater than $p + 1 > \sqrt{n}$.

We must also prove that there will be a sufficiently large prime less than n when n is prime, namely that the $(k + 1)$ st prime p_{k+1} is less than $p_k^2 + p_k$; otherwise k would exceed j and $\text{PRIME}[k]$ would be zero when we needed it to be large. The necessary proof follows from “Bertrand’s postulate”: If p is prime there is a larger prime less than $2p$.

12. We could move `Title`, `NewLn`, and `Blank` to the data segment following `BUF`, where they could use `p_top` as their base address. Or we could change the `LDA` instructions on lines 38, 42, and 58 to `SETL`, knowing that the string addresses happen to fit in two bytes because this program is short. Or we could change `LDA` to `GETA`; but in that case we would have to align each string modulo 4, for example by saying

```
Title BYTE "First Five Hundred Primes",#a,0
LOC (@+3)&-4
NewLn BYTE #a,0
LOC (@+3)&-4
Blanks BYTE " ",0
```

(See exercises 7 and 8.)

13. Line 35 gets the new title; change `BYTE` to `WYDE` on lines 35–37. Change `Fputs` to `Fputws` in lines 39, 43, 55, 59. Change the constant in line 45 to `#0020066006600660`. Change `BUF+4` to `BUF+2*4` on line 47. And change lines 50–52 to

```
INCL r,'..'; STWU r,t,0; SUB t,t,2.
```

Incidentally, the new title line might look like

```
Title WYDE "أول خمس مئات لأرقام الأولية"
```

when it is printed bidirectionally, but in the computer file the individual characters actually appear in “logical” order without ligatures. Thus a spelled-out sequence like

```
Title WYDE 'ا', 'ل', 'أ', 'ل', 'أ', 'ر', 'ق', 'ا', 'م', 'أ', 'ل', 'أ', 'ل', 'ي', 'ة', ''
```

would give an equivalent result, by the rule for string constants (rule 2).

14. We can, for example, replace lines 26–30 of Program P by

```
fn GREG 0
sqrttn GREG 0
    FLOT fn,n
    FSQRT sqrttn,fn
6H LDWU pk,p_top,kk
    FLOT t,pk
    FREM r,fn,t
    BZ r,4B
7H FCMP t,sqrttn,t
```

The new `FREM` instruction is performed 9597 times, not 9538, because the new test in step P7 is not quite as effective as before. In spite of this, the floating point calculations reduce the running time by $426192\mu - 59\mu$, a notable improvement (unless of course

$\mu/v > 7000$). An additional savings of $38169v$ can be achieved if the primes are stored as short floats instead of as unsigned wydes.

The number of divisibility tests can actually be reduced to 9357 if we replace q by $\sqrt{n} - 1.9999$ in step P7 (see the answer to exercise 11). But the extra subtractions cost more than they save, unless $\mu/v > 15$.

15. It prints a string consisting of a blank space followed by an asterisk followed by two blanks followed by an asterisk ... followed by k blanks followed by an asterisk ... followed by 74 blanks followed by an asterisk; a total of $2+3+\dots+75 = \binom{76}{2} - 1 = 2849$ characters. The total effect is one of OP art.

17. The following subroutine returns zero if and only if the instruction is OK.

a	IS	#ffffffff	Table entry when anything goes
b	IS	#ffff04ff	Table entry when Y \leq ROUND_NEAR
c	IS	#001f00ff	Table entry for PUT and PUTI
d	IS	#ff000000	Table entry for RESUME
e	IS	#ffff0000	Table entry for SAVE
f	IS	#ff0000ff	Table entry for UNSAVE
g	IS	#ff000003	Table entry for SYNC
h	IS	#ffff001f	Table entry for GET
table	GREG	@	
	TETRA	a,a,a,a,a,b,a,b,b,b,b,b,b,b,b,b	0x
	TETRA	a,a,a,a,a,b,a,b,a,a,a,a,a,a,a,a	1x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	2x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	3x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	4x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	5x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	6x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	7x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	8x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	9x
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Ax
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Bx
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Cx
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Dx
	TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Ex
	TETRA	a,a,a,a,a,c,c,a,d,e,f,g,a,h,a	Fx
tetra	IS	\$1	
maxXYZ	IS	\$2	
InstTest	BN	\$0,9F	Invalid if address is negative.
	LDTU	tetra,\$0,0	Fetch the tetrabyte.
	SR	\$0,tetra,22	Extract its opcode (times 4).
	LDT	maxXYZ,table,\$0	Get X _{max} , Y _{max} , Z _{max} .
	BDIF	\$0,tetra,maxXYZ	Check if any max is exceeded.
	PBNP	maxXYZ,9F	If not a PUT, we are done.
	ANDNML	\$0,#ff00	Zero out the OP byte.
	BNZ	\$0,9F	Branch if any max is exceeded.
	MOR	tetra,tetra,#4	Extract the X byte.
	CMP	\$0,tetra,18	
	CSP	tetra,\$0,0	Set X $\leftarrow 0$ if 18 < X < 32.

```

ODIF  $0,tetra,7      Set $0 ← X ÷ 7.
9H     POP   1,0       Return $0 as the answer. ■

```

This solution does not consider a tetrabyte to be invalid if it would jump to a negative address, nor is ‘SAVE \$0,0’ called invalid (although \$0 can never be a global register).

18. The catch to this problem is that there may be several places in a row or column where the minimum or maximum occurs, and each is a potential saddle point.

Solution 1: In this solution we run through each row in turn, making a list of all columns in which the row minimum occurs and then checking each column on the list to see if the row minimum is also a column maximum. Notice that in all cases the terminating condition for a loop is that a register is ≤ 0 .

* Solution 1		
t	IS	\$255
a00	GREG	Data_Segment Address of “ <i>a</i> ₀₀ ”
a10	GREG	Data_Segment+8 Address of “ <i>a</i> ₁₀ ”
ij	IS	\$0 Element index and return register
j	GREG	0 Column index
k	GREG	0 Size of list of minimum indices
x	GREG	0 Current minimum value
y	GREG	0 Current element
Saddle	SET	ij,9*8
RowMin	SET	j,8
	LDB	x,a10,ij Candidate for row minimum
2H	SET	k,0 Set list empty.
4H	INCL	k,1
	STB	j,a00,k Put column index in list.
1H	SUB	ij,ij,1 Go left one.
	SUB	j,j,1
	BZ	j,ColMax Done with row?
3H	LDB	y,a10,ij
	SUB	t,x,y
	PBN	t,1B Is x still minimum?
	SET	x,y
	PBP	t,2B New minimum?
	JMP	4B Remember another minimum.
ColMax	LDB	\$1,a00,k Get column from list.
	ADD	j,\$1,9*8-8
1H	LDB	y,a10,j
	CMP	t,x,y
	PBN	t,No Is row min < column element?
	SUB	j,j,8
	PBP	j,1B Done with column?
Yes	ADD	ij,ij,\$1 Yes; ij ← index of saddle.
	LDA	ij,a10,ij
	POP	1,0
No	SUB	k,k,1 Is list empty?
	BP	k,ColMax If not, try again.
	PBP	ij,RowMin Have all rows been tried?
	POP	1,0 Yes; \$0 = 0, no saddle. ■

Solution 2: An infusion of mathematics gives a different algorithm.

Theorem. Let $R(i) = \min_j a_{ij}$, $C(j) = \max_i a_{ij}$. The element $a_{i_0 j_0}$ is a saddle point if and only if $R(i_0) = \max_i R(i) = C(j_0) = \min_j C(j)$.

Proof. If $a_{i_0 j_0}$ is a saddle point, then for any fixed i , $R(i_0) = C(j_0) \geq a_{i_0 j_0} \geq R(i)$; so $R(i_0) = \max_i R(i)$. Similarly $C(j_0) = \min_j C(j)$. Conversely, we have $R(i) \leq a_{ij} \leq C(j)$ for all i and j ; hence $R(i_0) = C(j_0)$ implies that $a_{i_0 j_0}$ is a saddle point. ■

(This proof shows that we always have $\max_i R(i) \leq \min_j C(j)$. So there is no saddle point if and only if all the R 's are less than all the C 's.)

According to the theorem, it suffices to find the smallest column maximum, then to search for an equal row minimum.

* Solution 2			
t	IS	\$255	
a00	GREG	Data_Segment	Address of “ a_{00} ”
a10	GREG	Data_Segment+8	Address of “ a_{10} ”
a20	GREG	Data_Segment+8*2	Address of “ a_{20} ”
ij	GREG	0	Element index
ii	GREG	0	Row index times 8
j	GREG	0	Column index
x	GREG	0	Current maximum
y	GREG	0	Current element
z	GREG	0	Current min max
ans	IS	\$0	Return register
Phase1	SET	j,8	Start at column 8.
	SET	z,1000	$z \leftarrow \infty$ (more or less).
3H	ADD	ij,j,9*8-2*8	
	LDB	x,a20,ij	
1H	LDB	y,a10,ij	
	CMP	t,x,y	Is $x < y$?
	CSN	x,t,y	If so, update the maximum.
2H	SUB	ij,ij,8	Move up one.
	PBP	ij,1B	
	STB	x,a10,ij	Store column maximum.
	CMP	t,x,z	Is $x < z$?
	CSN	z,t,x	If so, update the min max.
	SUB	j,j,1	Move left a column.
	PBP	j,3B	
Phase2	SET	ii,9*8-8	(At this point $z = \min_j C(j)$.)
3H	ADD	ij,ii,8	Prepare to search a row.
	SET	j,8	
1H	LDB	x,a10,ij	
	SUB	t,z,x	Is $z > a_{ij}$?
	PBP	t,No	There's no saddle in this row.
	PBN	t,2F	
	LDB	x,a00,j	Is $a_{ij} = C(j)$?
	CMP	t,x,z	
	CSZ	ans,t,ij	If so, remember a possible saddle point.

2H	SUB j,j,1	Move left in row.
	SUB ij,ij,1	
	PBP j,1B	
	LDA ans,a10,ans	A saddle point was found here.
	POP 1,0	
No	SUB ii,ii,8	
	PBP ii,3B	Try another row.
	SET ans,0	
	POP 1,0	ans = 0; no saddle. ■

We leave it to the reader to invent a still better solution in which Phase 1 records all possible rows that are candidates for the row search in Phase 2. It is not necessary to search all rows, just those i_0 for which $C(j_0) = \min_j C(j)$ implies $a_{i_0 j_0} = C(j_0)$. Usually there is at most one such row.

In some trial runs with elements selected at random from $\{-2, -1, 0, 1, 2\}$, Solution 1 required approximately $147\mu + 863v$ to run, while Solution 2 took about $95\mu + 510v$. Given a matrix of all zeros, Solution 1 found a saddle point in $26\mu + 188v$, Solution 2 in $96\mu + 517v$.

If an $m \times n$ matrix has *distinct* elements, and $m \geq n$, we can solve the problem by looking at only $O(m+n)$ of them and doing $O(m \log n)$ auxiliary operations. See Bienstock, Chung, Fredman, Schäffer, Shor, and Suri, *AMM* **98** (1991), 418–419.

19. Assume an $m \times n$ matrix. (a) By the theorem in the answer to exercise 18, all saddle points of a matrix have the same value, so (under our assumption of distinct elements) there is at most one saddle point. By symmetry the desired probability is mn times the probability that a_{11} is a saddle point. This latter is $1/(mn)!$ times the number of permutations with $a_{12} > a_{11}, \dots, a_{1n} > a_{11}, a_{11} > a_{21}, \dots, a_{11} > a_{m1}$; and this is $1/(m+n-1)!$ times the number of permutations of $m+n-1$ elements in which the first is greater than the next $(m-1)$ and less than the remaining $(n-1)$, namely $(m-1)!(n-1)!$. The answer is therefore

$$mn(m-1)!(n-1)!/(m+n-1)! = (m+n) \Big/ \binom{m+n}{n}.$$

In our case this is $17/\binom{17}{8}$, only one chance in 1430. (b) Under the second assumption, an entirely different method must be used since there can be multiple saddle points; in fact either a whole row or whole column must consist entirely of saddle points. The probability equals the probability that there is a saddle point with value zero plus the probability that there is a saddle point with value one. The former is the probability that there is at least one column of zeros; the latter is the probability that there is at least one row of ones. The answer is $(1 - (1 - 2^{-m})^n) + (1 - (1 - 2^{-n})^m)$; in our case, 924744796234036231/18446744073709551616, about 1 in 19.9. An approximate answer is $n2^{-m} + m2^{-n}$.

20. M. Hofri and P. Jacquet [*Algorithmica* **22** (1998), 516–528] have analyzed the case when the $m \times n$ matrix entries are distinct and in random order. The running times of the two MMIX programs are then $(mn + mH_n + 2m + 1 + (m+1)/(n-1))\mu + (6mn + 7mH_n + 5m + 11 + 7(m+1)/(n-1))v + O((m+n)^2/\binom{m+n}{m})$ and $(m+1)n\mu + (5mn + 6m + 4n + 7H_n + 8)v + O(1/n) + O((\log n)^2/m)$, respectively, as $m \rightarrow \infty$ and $n \rightarrow \infty$, assuming that $(\log n)/m \rightarrow 0$.

21. Farey SET y,1; ... POP.

 This answer is the first of many in Volumes 1–3 for which MMIXmasters are being asked to contribute elegant solutions. (See the website information on page ii.) The fourth edition of this book will present the best parts of the best programs submitted. Note: Please reveal your full name, including all middle names, if you enter this competition, so that proper credit can be given!

22. (a) Induction. (b) Let $k \geq 0$ and $X = ax_{k+1} - x_k$, $Y = ay_{k+1} - y_k$, where $a = \lfloor (y_k + n)/y_{k+1} \rfloor$. By part (a) and the fact that $0 < Y \leq n$, we have $X \perp Y$ and $X/Y > x_{k+1}/y_{k+1}$. So if $X/Y \neq x_{k+2}/y_{k+2}$ we have, by definition, $X/Y > x_{k+2}/y_{k+2}$. But this implies that

$$\begin{aligned}\frac{1}{Yy_{k+1}} &= \frac{Xy_{k+1} - Yx_{k+1}}{Yy_{k+1}} = \frac{X}{Y} - \frac{x_{k+1}}{y_{k+1}} \\ &= \left(\frac{X}{Y} - \frac{x_{k+2}}{y_{k+2}} \right) + \left(\frac{x_{k+2}}{y_{k+2}} - \frac{x_{k+1}}{y_{k+1}} \right) \\ &\geq \frac{1}{Yy_{k+2}} + \frac{1}{y_{k+1}y_{k+2}} = \frac{y_{k+1} + Y}{Yy_{k+1}y_{k+2}} \\ &> \frac{n}{Yy_{k+1}y_{k+2}} \geq \frac{1}{Yy_{k+1}}.\end{aligned}$$

Historical notes: C. Haros gave a (more complicated) rule for constructing such sequences, in *J. de l'École Polytechnique* 4, 11 (1802), 364–368; his method was correct, but his proof was inadequate. Several years later, the geologist John Farey independently conjectured that x_k/y_k is always equal to $(x_{k-1} + x_{k+1})/(y_{k-1} + y_{k+1})$ [*Philosophical Magazine and Journal* 47 (1816), 385–386]; a proof was supplied shortly afterwards by A. Cauchy [*Bulletin Société Philomathique de Paris* (3) 3 (1816), 133–135], who attached Farey's name to the series. For more of its interesting properties, see G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, Chapter 3.

23. The following routine should do reasonably well on most pipeline and cache configurations.

a	IS	\$0	SUB	n,n,8	STCO	0,a,56
n	IS	\$1	ADD	a,a,8	ADD	a,a,64
z	IS	\$2	3H AND	t,a,63	PBNN	t,4B
t	IS	\$255	PBNZ	t,2B	5H CMP	t,n,8
			CMP	t,n,64	BN	t,7F
1H	STB	z,a,0	BN	t,5F	6H STCO	0,a,0
	SUB	n,n,1	4H PREST	63,a,0	SUB	n,n,8
	ADD	a,a,1	SUB	n,n,64	ADD	a,a,8
Zero	BZ	n,9F	CMP	t,n,64	CMP	t,n,8
	SET	z,0	STCO	0,a,0	PBNN	t,6B
	AND	t,a,7	STCO	0,a,8	7H BZ	n,9F
	BNZ	t,1B	STCO	0,a,16	8H STB	z,a,0
	CMP	t,n,64	STCO	0,a,24	SUB	n,n,1
	PBNN	t,3F	STCO	0,a,32	ADD	a,a,1
	JMP	5F	STCO	0,a,40	PBNZ	n,8B
2H	STCO	0,a,0	STCO	0,a,48	9H POP	■

24. The following routine merits careful study; comments are left to the reader. A faster program would be possible if we treated $\$0 \equiv \1 (modulo 8) as a special case.

in	IS	\$2	SUB	\$1,\$1,8
out	IS	\$3	SRU	out,out,l
r	IS	\$4	MUX	in,in,out
l	IS	\$5	BDIF	t,ones,in
m	IS	\$6	AND	t,t,m
t	IS	\$7	SRU	mm,mm,r
mm	IS	\$8	PUT	rM,mm
tt	IS	\$9	PBZ	t,2F
flip	GREG	#0102040810204080	JMP	5F
ones	GREG	#0101010101010101	3H	MUX
	LOC	#100		out,tt,out
StrCpy	AND	in,\$0,#7	2H	SLU
	SLU	in,in,3		out,in,l
	AND	out,\$1,#7		LDOU
	SLU	out,out,3		in,\$0,8
	SUB	r,out,in	4H	INCL
	LDOU	out,\$1,0		\$0,8
	SUB	\$1,\$1,\$0		MUX
	NEG	m,0,1		out,tt,out
	SRU	m,m,in		BNZ
	LDOU	in,\$0,0		mm,1F
	PUT	rM,m	5H	STOU
	NEG	mm,0,1		out,\$0,\$1
	BN	r,1F		INCL
	NEG	l,64,r	1H	\$0,8
	SLU	tt,out,r		SLU
	MUX	in,in,tt		out,in,l
	BDIF	t,ones,in		SLU
	AND	t,t,m		mm,t,l
	SRU	mm,mm,r	1H	LDOU
	PUT	rM,mm		in,\$0,\$1
	JMP	4F		MOR
1H	NEG	l,0,r		mm,mm,flip
	INCL	r,64		SUBU
				mm,mm,1
				MOR
				mm,mm,flip
				SUBU
				mm,mm,1
				PUT
				rM,mm
				MUX
				in,in,out
				STOU
				in,\$0,\$1
				POP
				0

The running time, approximately $(n/4 + 4)\mu + (n + 40)v$ plus the time to POP, is less than the cost of the trivial code when $n \geq 8$ and $\mu \geq v$.

25. We assume that register p initially contains the address of the first byte, and that this address is a multiple of 8. Other local or global registers a, b, ... have also been declared. The following solution starts by counting the wyde frequencies first, since this requires only half as many operations as it takes to count byte frequencies. Then the byte frequencies are obtained as row and column sums of a 256×256 matrix.

```
* Cryptanalysis Problem (CLASSIFIED)
    LOC    Data_Segment
count  GREG   @          Base address for wyde counts
       LOC   @+8*(1<<16)  Space for the wyde frequencies
freq   GREG   @          Base address for byte counts
       LOC   @+8*(1<<8)   Space for the byte frequencies
p      GREG   @
BYTE   "abracadabraa",0,"abc" Trivial test data
```

ones	GREG	#0101010101010101		
	LOC	#100		
2H	SRU	b,a,45	Isolate next wyde.	main loop, should run as fast as possible
	LDO	c,count,b	Load old count.	
	INCL	c,1		
	STO	c,count,b	Store new count.	
	SLU	a,a,16	Delete one wyde.	
	PBNZ	a,2B	Done with octabyte?	
Phase1	LDOU	a,p,0	Start here: Fetch the next eight bytes.	
	INCL	p,8		
	BDIF	t,ones,a	Test if there's a zero byte.	
	PBZ	t,2B	Do main loop, unless near the end.	
2H	SRU	b,a,45	Isolate next wyde.	
	LDO	c,count,b	Load old count.	
	INCL	c,1		
	STO	c,count,b	Store new count.	
	SRU	b,t,48		
	SLU	a,a,16		
	BDIF	t,ones,a		
	PBZ	b,2B	Continue unless done.	
Phase2	SET	p,8*255	Now get ready to sum rows and columns.	
1H	SL	a,p,8		
	LDA	a,count,a	a ← address of row p.	
	SET	b,8*255		
	LDO	c,a,0		
	SET	t,p		
2H	INCL	t,#800		
	LDO	x,count,t	Element of column p	
	LDO	y,a,b	Element of row p	
	ADD	c,c,x		
	ADD	c,c,y		
	SUB	b,b,8		
	PBP	b,2B		
	STO	c,freq,p		
	SUB	p,p,8		
	PBP	p,1B		
	POP	■		

How long is “long”? This two-phase method is inferior to a simple one-phase approach when the string length n is less than 2^{17} , but it takes only about $10/17$ as much time as the one-phase scheme when $n \approx 10^6$. A slightly faster routine can be obtained by “unrolling” the inner loop, as in the next answer.

Another approach, which uses a jump table and keeps the counts in 128 registers, is worthy of consideration when μ/v is large.

[This problem has a long history. See, for example, Charles P. Bourne and Donald F. Ford, “A study of the statistics of letters in English words,” *Information and Control* 4 (1961), 48–67.]

26. The wyde-counting trick in the previous solution will backfire if the machine’s primary cache holds fewer than 2^{19} bytes, unless comparatively few of the wyde counts

are nonzero. Therefore the following program computes only one-byte frequencies. This code avoids stalls, in a conventional pipeline, by never using the result of a LDO in the immediately following instruction.

```

Start  LDOU a,p,0          INCL c,1
       INCL p,8           SRU bb,bb,53
       BDIF t,ones,a      STO c,freq,b
       BNZ t,3F           LDO c,freq,bb
2H     SRU b,a,53          LDOU a,p,0
       LDO c,freq,b      INCL p,8
       SLU bb,a,8          INCL c,1
       INCL c,1           BDIF t,ones,a
       SRU bb,bb,53        STO c,freq,bb
       STO c,freq,b        PBZ t,2B
       LDO c,freq,bb        3H   SRU b,a,53
       SLU b,a,16          LDO c,freq,b
       INCL c,1           INCL c,1
       SRU b,b,53          STO c,freq,b
       STO c,freq,bb        SRU b,b,3
       LDO c,freq,b        SLU a,a,8
       ...                 PBNZ b,3B
       SLU bb,a,56          POP   ■

```

Another solution works better on a superscalar machine that issues two instructions simultaneously:

```

Start  LDOU a,p,0          SLU bbb,a,48
       INCL p,8           SLU bbbb,a,56
       BDIF t,ones,a      INCL c,1
       SLU bb,a,8          INCL cc,1
       BNZ t,3F           SRU bbb,bbb,53
2H     SRU b,a,53          SRU bbbb,bbbb,53
       SRU bb,bb,53        STO c,freq,b
       LDO c,freq,b        STO cc,freqq,bb
       LDO cc,freqq,bb     LDO c,freq,bbb
       SLU bbb,a,16         LDO cc,freqq,bbbb
       SLU bbbb,a,24        LDOU a,p,0
       INCL c,1           INCL p,8
       INCL cc,1           INCL c,1
       SRU bbb,bbb,53      INCL cc,1
       SRU bbbb,bbbb,53    BDIF t,ones,a
       STO c,freq,b        SLU bb,a,8
       STO cc,freqq,bb     STO c,freq,bbb
       LDO c,freq,bbb      STO cc,freqq,bbbb
       LDO cc,freqq,bbbb    PBZ t,2B
       SLU b,a,32          3H   SRU b,a,53
       SLU bb,a,40          ...

```

In this case we must keep two separate frequency tables (and combine them at the end); otherwise an “aliasing” problem would lead to incorrect results in cases where *b* and *bb* both represent the same character.

27. (a)

t	IS	\$255	t	IS	\$255
n	IS	\$0	n	IS	\$0
new	GREG		new	GREG	
old	GREG		old	GREG	
phi	GREG		phii	GREG	#9e3779b97f4a7c16
rt5	GREG		lo	GREG	
acc	GREG		hi	GREG	
f	GREG		hihi	GREG	
	LOC	#100		LOC	#100
Main	FLOT	t,5	Main	SET	n,2
	FSQRT	rt5,t		SET	old,1
	FLOT	t,1		SET	new,1
	FADD	phi,t,rt5	1H	ADDU	new,new,old
	INCH	phi,#fff0		INCL	n,1
	FDIV	acc,phi,rt5		CMPU	t,new,old
	SET	n,1		BN	t,9F
	SET	new,1		SUBU	old,new,old
1H	ADDU	new,new,old		MULU	lo,old,phii
	INCL	n,1		GET	hi,rH
	CMPU	t,new,old		ADDU	hi,hi,old
	BN	t,9F		ADDU	hihi,hi,1
	SUBU	old,new,old		CSN	hi,lo,hihi
	FMUL	acc,acc,phi		CMP	t,hi,new
	FIXU	f,acc		PBZ	t,1B
	CMP	t,f,new		SET	t,1
	PBZ	t,1B	9H	TRAP	0,Halt,0
	SET	t,1			■
9H	TRAP	0,Halt,0			■

Program (a) halts with $t = 1$ and $n = 71$; the floating point representation of ϕ is slightly high, hence errors ultimately accumulate until $\phi^{71}/\sqrt{5}$ is approximated by $F_{71} + .7$, which rounds to $F_{71} + 1$. Program (b) halts with $t = -1$ and $n = 94$; unsigned overflow occurs before the approximation fails. (Indeed, $F_{93} < 2^{64} < F_{94}$.)

29. The last man is in position 15. The total time before output is ...



MMIXmasters, please help! What is the neatest program that is analogous to the solution to exercise 1.3.2–22 in the third edition? Also, what would D. Ingalls do in the new situation? (Find a trick analogous to his previous scheme, but do not use self-modifying code.)

An asymptotically faster method appears in exercise 5.1.1–5.

30. Work with scaled numbers, $R_n = 10^n r_n$. Then $R_n(1/m) = R$ if and only if $10^n/(R + \frac{1}{2}) \leq m < 10^n/(R - \frac{1}{2})$; thus we find $m_{k+1} = \lfloor (2 \cdot 10^n - 1)/(2R - 1) \rfloor$.

* Sum of Rounded Harmonic Series

MaxN	IS	10	
a	GREG	0	Accumulator
c	GREG	0	$2 \cdot 10^n$
d	GREG	0	Divisor or digit
r	GREG	0	Scaled reciprocal

s	GREG	0	Scaled sum
m	GREG	0	m_k
mm	GREG	0	m_{k+1}
nn	GREG	0	$n - \text{MaxN}$
	LOC	Data_Segment	
dec	GREG	@+3	Decimal point location
	BYTE	" . "	
	LOC	#100	
Main	NEG	nn,MaxN-1	$n \leftarrow 1.$
	SET	c,20	
1H	SET	m,1	
	SR	s,c,1	$S \leftarrow 10^n.$
	JMP	2F	
3H	SUB	a,c,1	
	SL	d,r,1	
	SUB	d,d,1	
	DIV	mm,a,d	
4H	SUB	a,mm,m	
	MUL	a,r,a	
	ADD	s,s,a	
	SET	m,mm	$k \leftarrow k + 1.$
2H	ADD	a,c,m	
	2ADDU	d,m,2	
	DIV	r,a,d	
	PBNZ	r,3B	
5H	ADD	a,nn,MaxN+1	
	SET	d,#a	Newline
	JMP	7F	
6H	DIV	s,s,10	Convert digits.
	GET	d,rR	
	INCL	d,'0'	
7H	STB	d,dec,a	
	SUB	a,a,1	
	BZ	a,@-4	
	PBNZ	s,6B	
8H	SUB	\$255,dec,3	
	TRAP	0,Fputs,StdOut	
9H	INCL	nn,1	$n \leftarrow n + 1.$
	MUL	c,c,10	
	PBNP	nn,1B	
	TRAP	0,Halt,0	■

The outputs are respectively 3.7, 6.13, 8.445, 10.7504, 13.05357, 15.356255, 17.6588268, 19.96140681, 22.263991769, 24.5665766342, in $82\mu + 40659359v$. The calculation would work for n up to 17 without overflow, but the running time is of order $10^{n/2}$. (We could save about half the time by calculating $R_n(1/m)$ directly when $m < 10^{n/2}$, and by using the fact that $R_n(m_{k+1}) = R_n(m_k - 1)$ for larger values of m .)

31. Let $N = \lfloor 2 \cdot 10^n / (2m+1) \rfloor$. Then $S_n = H_N + O(N/10^n) + \sum_{k=1}^m (\lceil 2 \cdot 10^n / (2k-1) \rceil - \lfloor 2 \cdot 10^n / (2k+1) \rfloor)k/10^n = H_N + O(m^{-1}) + O(m/10^n) - 1 + 2H_{2m} - H_m = n \ln 10 + 2\gamma - 1 + 2 \ln 2 + O(10^{-n/2})$ if we sum by parts and set $m \approx 10^{n/2}$.

Our approximation to S_{10} is ≈ 24.5665766209 , which is closer than predicted.

- 32.** To make the problem more challenging, the following ingenious solution due in part to ——— uses a lot of *trickery* in order to reduce execution time. Can the reader squeeze out any more nanoseconds?

 **MMIXmasters:** Please help fill in the blanks! Note, for example, that remainders mod 7, 19, and 30 are most rapidly computed by FREM; division by 100 can be reduced to multiplication by 1//100+1 (see exercise 1.3.1'-19); etc.

[To calculate Easter in years ≤ 1582 , see CACM 5 (1962), 209–210. The first systematic algorithm for calculating the date of Easter was the *canon paschalis* due to Victorius of Aquitania (A.D. 457). There are many indications that the sole nontrivial application of arithmetic in Europe during the Middle Ages was the calculation of Easter date, hence such algorithms are historically significant. See *Puzzles and Paradoxes* by T. H. O’Beirne (London: Oxford University Press, 1965), Chapter 10, for further commentary; and see the book *Calendrical Calculations* by E. M. Reingold and N. Dershowitz (Cambridge Univ. Press, 2001) for date-oriented algorithms of all kinds.]

- 33.** The first such year is A.D. 10317, although the error *almost* leads to failure in A.D. $10108 + 19k$ for $0 \leq k \leq 10$.

Incidentally, T. H. O’Beirne pointed out that the date of Easter repeats with a period of exactly 5,700,000 years. Calculations by Robert Hill show that the most common date is April 19 (220400 times per period), while the earliest and least common is March 22 (27550 times); the latest, and next-to-least common, is April 25 (42000 times). Hill found a nice explanation for the curious fact that the number of times any particular day occurs in the period is always a multiple of 25.

- 34.** The following program follows the protocol to within a dozen or so v ; this is more than sufficiently accurate, since ρ is typically more than 10^8 , and $\rho v = 1$ sec. All computation takes place in registers, except when a byte is input.

```
* Traffic Signal Problem
rho      GREG  250000000          Assume 250 MHz clock rate
t       IS    $255
Sensor_Buf IS   Data_Segment
            GREG Sensor_Buf
            LOC  #100
Lights   IS    3                  Handle for /dev/lights
Sensor   IS    4                  Handle for /dev/sensor
Lights_Name BYTE "/dev/lights",0
Sensor_Name BYTE "/dev/sensor",0
Lights_Args OCTA Lights_Name,BinaryWrite
Sensor_Args OCTA Sensor_Name,BinaryRead
Read_Sensor OCTA Sensor_Buf,1
Boulevard BYTE #77,0           Green/red, WALK/DON'T
                  BYTE #7f,0           Green/red, DON'T/DON'T
                  BYTE #73,0           Green/red, off/DON'T
                  BYTE #bf,0           Amber/red, DON'T/DON'T
Avenue    BYTE #dd,0           Red/green, DON'T/WALK
                  BYTE #df,0           Red/green, DON'T/DON'T
                  BYTE #dc,0           Red/green, DON'T/off
                  BYTE #ef,0           Red/amber, DON'T/DON'T
```

goal	GREG	0	Transition time for lights
Main	GETA	t,Lights_Args	Open the files: Fopen(Lights, "/dev/lights",BinaryWrite)
	TRAP	0,Fopen,Lights	
	GETA	t,Sensor_Args	Fopen(Sensor, "/dev/sensor",BinaryRead)
	TRAP	0,Fopen,Sensor	
	GET	goal,rC	
	JMP	2F	
	GREG	@	
delay_go	GREG		
Delay	GET	t,rC	Subroutine for busy-waiting: (N.B. Not CMPU; see below)
	SUBU	t,t,goal	
	PBN	t,Delay	Repeat until rC passes goal.
	GO	delay_go,delay_go,0	Return to caller.
flash_go	GREG		
n	GREG	0	Iteration counter
green	GREG	0	Boulevard or Avenue
temp	GREG		
Flash	SET	n,8	Subroutine to flash the lights:
1H	ADD	t,green,2*1	
	TRAP	0,Fputs,Lights	DON'T WALK
	ADD	temp,goal,rho	
	SR	t,rho,1	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	
	ADD	t,green,2*2	
	TRAP	0,Fputs,Lights	(off)
	SET	goal,temp	
	GO	delay_go,Delay	
	SUB	n,n,1	
	PBP	n,1B	Repeat eight times.
	ADD	t,green,2*1	
	TRAP	0,Fputs,Lights	DON'T WALK
	MUL	t,rho,4	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	Hold for 4 sec.
	ADD	t,green,2*3	
	TRAP	0,Fputs,Lights	DON'T WALK, amber
	GO	flash_go,flash_go,0	Return to caller.
Wait	GET	goal,rC	Extend the 18 sec green.
1H	GETA	t,Read_Sensor	
	TRAP	0,Fread,Sensor	
	LDB	t,Sensor_Buf	
	BZ	t,Wait	Repeat until sensor is nonzero.
	GETA	green,Boulevard	
	GO	flash_go,Flash	Finish the boulevard cycle.
	MUL	t,rho,8	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	Amber for 8 sec.

	GETA t,Avenue	
	TRAP 0,Fputs,Lights	Green light for Berkeley.
	MUL t,rho,8	
	ADDU goal,goal,t	
	GO delay_go,Delay	
	GETA green,Avenue	
	GO flash_go,Flash	Finish the avenue cycle.
	GETA t,Read_Sensor	
	TRAP 0,Fread,Sensor	Ignore sensor during green time.
	MUL t,rho,5	
	ADDU goal,goal,t	
	GO delay_go,Delay	Amber for 5 sec.
2H	GETA t,Boulevard	
	TRAP 0,Fputs,Lights	Green light for Del Mar.
	MUL t,rho,18	
	ADDU goal,goal,t	
	GO delay_go,Delay	At least 18 sec to WALK.
	JMP 1B	■

The SUBU instruction in the Delay subroutine is an interesting example of a case where the comparison should be done with SUBU, *not* with CMPU, in spite of the comments in exercise 1.3.1'-22. The reason is that the two quantities being compared, rC and goal, "wrap around" modulo 2^{64} .

SECTION 1.4.1'

1. j GREG ;m GREG ;kk GREG ;xk GREG ;rr GREG
 GREG @ Base address
 GoMax SET \$2,1 Special entrance for $r = 1$
 GoMaxR SL rr,\$2,3 Multiply arguments by 8.
 SL kk,\$1,3
 LDO m,x0,kk
 ... (Continue as in (1))
 5H SUB kk,kk,rr $k \leftarrow k - r$.
 PBP kk,3B Repeat if $k > 0$.
 6H GO kk,\$0,0 Return to caller. ■

The calling sequence for the general case is SET \$2,\$r; SET \$1,\$n; GO \$0,GoMaxR.

2. j IS \$0 ;m IS \$1 ;kk IS \$2 ;xk IS \$3 ;rr IS \$4
 Max100 SET \$0,100 Special entrance for $n = 100$ and $r = 1$
 Max SET \$1,1 Special entrance for $r = 1$
 MaxR SL rr,\$1,3 Multiply arguments by 8.
 SL kk,\$0,3
 LDO m,x0,kk
 ... (Continue as in (1))
 5H SUB kk,kk,rr $k \leftarrow k - r$.
 PBP kk,3B Repeat if $k > 0$.
 6H POP 2,0 Return to caller. ■

In this case the general calling sequence is SET \$A1,\$r; SET \$A0,\$n; PUSHJ \$R,MaxR, where $A0 = R + 1$ and $A1 = R + 2$.

3. Just Sub ...; GO \$0,\$0,0. The local variables can be kept entirely in registers.

4. `PUSHJ $X,RA` has a relative address, allowing us to jump to any subroutine within $\pm 2^{18}$ bytes of our current location. `PUSHGO $X,$Y,$Z` or `PUSHGO $X,A` has an absolute address, allowing us to jump to any desired place.

5. True. There are $256 - G$ globals and L locals.

6. $\$5 \leftarrow rD$ and $rR \leftarrow 0$ and $rL \leftarrow 6$. All other newly local registers are also set to zero; for example, if rL was 3, this `DIVU` instruction would set $\$3 \leftarrow 0$ and $\$4 \leftarrow 0$.

7. $\$L \leftarrow 0, \dots, \$4 \leftarrow 0, \$5 \leftarrow \#abcd0000, rL \leftarrow 6$.

8. Usually such an instruction has no essential impact, except that context switching with `SAVE` and `UNSAVE` generally take longer when fewer marginal registers are present. However, an important difference can arise in certain scenarios. For example, a subsequent `PUSHJ $255,Sub` followed by `POP 1,0` would leave a result in $\$16$ instead of $\$10$.

9. `PUSHJ $255,Handler` will make at least 32 marginal registers available (because $G \geq 32$); then `POP 0` will restore the previous local registers, and two additional instructions “`GET $255,rB; RESUME`” will restart the program as if nothing had happened.

10. Basically true. MMIX will start a program with rG set to 255 minus the number of assembled `GREG` operations, and with rL set to 2. Then, in the absence of `PUSHJ`, `PUSHGO`, `POP`, `SAVE`, `UNSAVE`, `GET`, and `PUT`, the value of rG will never change. The value of rL will increase if the program puts anything into $\$2, \$3, \dots$, or $\$(rG - 1)$, but the effect will be the same as if all registers were equivalent. The only register with slightly different behavior is $\$255$, which is affected by trip interrupts and used for communication in I/O traps. We could permute register numbers $\$2, \$3, \dots, \$254$ arbitrarily in any `PUSH/POP/SAVE/UNSAVE/RESUME`-free program that does not `GET rL` or `PUT` anything into rL or rG ; the permuted program would produce identical results.

The distinction between local, global, and marginal is irrelevant also with respect to `SAVE`, `UNSAVE`, and `RESUME`, in the absence of `PUSH` and `POP`, except that the destination register of `SAVE` must be global and the destination register of certain instructions inserted by `RESUME` mustn’t be marginal (see exercise 1.4.3’–14).

11. The machine tries to access virtual address `#5fffffff ffffff8`, which is just below the stack segment. Nothing has been stored there, so a “page fault” occurs and the operating system aborts the program.

(The behavior is, however, much more bizarre if a `POP` is given just after a `SAVE`, because `SAVE` essentially begins a new register stack immediately following the saved context. Anybody who tries such things is asking for trouble.)

12. (a) True. (Similarly, the name of the current “working directory” in a UNIX shell always begins with a slash.) (b) False. But confusion can arise if such prefixes are defined, so their use is discouraged. (c) False. (In this respect MMIXAL’s structured symbols are *not* analogous to UNIX directory names.)

13. Fib	<code>CMP \$1,\$0,2</code>	<code>Fib1</code>	<code>CMP \$1,\$0,2</code>	<code>Fib2</code>	<code>CMP \$1,\$0,1</code>
	<code>PBN \$1,1F</code>		<code>BN \$1,1F</code>		<code>BNP \$1,1F</code>
	<code>GET \$1,rJ</code>		<code>SUB \$2,\$0,1</code>		<code>SUB \$2,\$0,1</code>
	<code>SUB \$3,\$0,1</code>		<code>SET \$0,1</code>		<code>SET \$0,0</code>
	<code>PUSHJ \$2,Fib</code>		<code>SET \$1,0</code>	<code>2H</code>	<code>ADDU \$0,\$0,\$1</code>
	<code>SUB \$4,\$0,2</code>	<code>2H</code>	<code>ADDU \$0,\$0,\$1</code>		<code>ADDU \$1,\$0,\$1</code>
	<code>PUSHJ \$3,Fib</code>		<code>SUBU \$1,\$0,\$1</code>		<code>SUB \$2,\$2,2</code>
	<code>ADDU \$0,\$2,\$3</code>		<code>SUB \$2,\$2,1</code>		<code>PBP \$2,2B</code>
	<code>PUT rJ,\$1</code>		<code>PBNZ \$2,2B</code>		<code>CSZ \$0,\$2,\$1</code>
1H	<code>POP 1,0</code>	■	<code>POP 1,0</code>	■	<code>POP 1,0</code>

Here **Fib2** is a faster alternative to **Fib1**. In each case the calling sequence has the form “SET \$A,n; PUSHJ \$R,Fib...”, where A = R + 1.

14. Mathematical induction shows that the POP instruction in **Fib** is executed exactly $2F_{n+1} - 1$ times and the ADDU instruction is executed $F_{n+1} - 1$ times. The instruction at 2H is performed $n - [n \neq 0]$ times in **Fib1**, $[n/2]$ times in **Fib2**. Thus the total cost, including the two instructions in the calling sequence, comes to $(19F_{n+1} - 12)v$ for **Fib**, $(4n + 8)v$ for **Fib1**, and $(4[n/2] + 12)v$ for **Fib2**, assuming that $n > 1$.

(The recursive subroutine **Fib** is a *terrible* way to compute Fibonacci numbers, because it forgets the values it has already computed. It spends more than $10^{22}v$ units of time just to compute F_{100} .)

15.	n	GREG		GO	\$0,Fib
	fn	IS n		STO	fn,fp,24
		GREG @		LDO	n,fp,16
	Fib	CMP \$1,n,2		SUB	n,n,2
		PBN \$1,1F		GO	\$0,Fib
		STO fp,sp,0		LDO	\$0,fp,24
		SET fp,sp		ADDU	fn,fn,\$0
		INCL sp,8*4		LDO	\$0,fp,8
		STO \$0,fp,8		SET	sp,fp
		STO n,fp,16		LDO	fp,sp,0
		SUB n,n,1	1H	GO	\$0,\$0,0

The calling sequence is SET n,n; GO \$0,Fib; the answer is returned in global register fn. The running time comes to $(8F_{n+1} - 8)\mu + (32F_{n+1} - 23)v$, so the ratio between this version and the register stack subroutine of exercise 13 is approximately $(8\mu/v + 32)/19$. (Although exercise 14 points out that we shouldn't really calculate Fibonacci numbers recursively, this analysis does demonstrate the advantage of a register stack. Even if we are generous and assume that $\mu = v$, the memory stack costs more than twice as much in this example. A similar behavior occurs with respect to other subroutines, but the analysis for **Fib** is particularly simple.)

In the special case of **Fib** we can do without the frame pointer, because fp is always a fixed distance from sp. A memory-stack subroutine based on this observation runs about $(6\mu/v + 29)/19$ slower than the register-stack version; it's better than the version with general frames, but still not very good.

16. This is an ideal setup for a subroutine with two exits. Let's assume for convenience that B and C do not return any value, and that they each save rJ in \$1 (because they are not leaf subroutines). Then we can proceed as follows: A calls B by saying PUSHJ \$R,B as usual. B calls C by saying PUSHJ \$R,C; PUT rJ,\$1; POP 0,0 (with perhaps a different value of R than used by subroutine A). C calls itself by saying PUSHJ \$R,C; PUT rJ,\$1; POP 0,0 (with perhaps a different value of R than used by B). C jumps to A by saying PUT rJ,\$1; POP 0,0. C exits normally by saying PUT rJ,\$1; POP 0,2.

Extensions of this idea, in which values are returned and an arbitrary jump address can be part of the returned information, are clearly possible. Similar schemes apply to the GO-oriented memory stack protocol of (15).

SECTION 1.4.2'

- If one coroutine calls the other only once, it is nothing but a subroutine; so we need an application in which each coroutine calls the other in at least two distinct places. Even then, it is often easy to set some sort of switch or to use some property

of the data, so that upon entry to a fixed place within one coroutine it is possible to branch to one of two desired places; again, nothing more than a subroutine would be required. Coroutines become correspondingly more useful as the number of references between them grows larger.

2. The first character found by `In` would be lost.

3. This is an MMIXAL trick to make `OutBuf` contain fifteen tetrabytes `TETRA ' '`, followed by `TETRA #a`, followed by zero; and `TETRA ' '` is equivalent to `BYTE 0,0,0,' '`. The output buffer is therefore set up to receive a line of 16 three-character groups separated by blank spaces.

4. If we include the code

```
rR_A GREG
rR_B GREG
    GREG @
A GET rR_B,rR
    PUT rR,rR_A
    GO t,a,0
B GET rR_A,rR
    PUT rR,rR_B
    GO t,b,0
```

then A can invoke B by “`GO a,B`” and B can invoke A by “`GO b,A`”.

5. If we include the code

```
a GREG
b GREG
    GREG @
A GET b,rJ
    PUT rJ,a
    POP 0
B GET a,rJ
    PUT rJ,b
    POP 0
```

then A can invoke B by “`PUSHJ $255,B`” and B can invoke A by “`PUSHJ $255,A`”. Notice the similarity between this answer and the previous one. The coroutines should not use the register stack for other purposes except as permitted by the following exercise.

6. Suppose coroutine A has something in the register stack when invoking B. Then B is obliged to return the stack to the same state before returning to A, although B might push and pop any number of items in the meantime.

Coroutines might, of course, be sufficiently complicated that they each do require a register stack of their own. In such cases MMIX’s `SAVE` and `UNSAVE` operations can be used, with care, to save and restore the context needed by each coroutine.

SECTION 1.4.3’

1. (a) `SRU x,y,z; BYTE 0,1,0,#29.` (b) `PBP x,PBTaken+@-0; BYTE 0,3,0,#50.`
 (c) `MUX x,y,z; BYTE 0,1,rM,#29.` (d) `ADDU x,x,z; BYTE 0,1,0,#30.`

2. The running time of `MemFind` is $9v + (2\mu + 8v)C + (3\mu + 6v)U + (2\mu + 11v)A$, where C is the number of key comparisons on line 042, $U = [\text{key} \neq \text{curkey}]$, and $A = [\text{new node needed}]$. The running time of `GetReg` is $\mu + 6v + 6vL$, where $L = [\$k \text{ is local}]$.

If we assume that $C = U = A = L = 0$ on each call, the time for simulation can be broken down as follows:

	(a)	(b)	(c)
fetching (lines 105–115)	$\mu + 17v$	$\mu + 17v$	$\mu + 17v$
unpacking (lines 141–153)	$\mu + 12v$	$\mu + 12v$	$\mu + 12v$
relating (lines 154–164)	$2v$	$2v$	$9v$
installing X (lines 174–182)	$7v$	$\mu + 17v$	$\mu + 17v$
installing Z (lines 183–197)	$\mu + 13v$	$6v$	$6v$
installing Y (lines 198–207)	$\mu + 13v$	$\mu + 13v$	$6v$
destining (lines 208–231)	$8v$	$23v$	$6v$
resuming (lines 232–242)	$14v$	$\mu + 14v$	$16v - \pi$
postprocessing (lines 243–539)	$\mu + 10v$	$11v$	$11v - 4\pi$
updating (lines 540–548)	$5v$	$5v$	$5v$
total	$5\mu + 101v$	$5\mu + 120v$	$3\mu + 105v - 5\pi$

To these times we must add $6v$ for each occurrence of a local register as a source, plus penalties for the times when `MemFind` doesn't immediately have the correct chunk. In case (b), `MemFind` *must* miss on line 231, and again on line 111 when fetching the following instruction. (We would be better off with *two* `MemFind` routines, one for data and one for instructions.) The most optimistic net cost of (b) is therefore obtained by taking $C = A = 2$, for a total running time of $13\mu + 158v$. (On long runs of the simulator simulating itself, the empirical average values per call of `MemFind` were $C \approx .29$, $U \approx .00001$, $A \approx .16$.)

3. We have $\beta = \gamma$ and $L > 0$ on line 097. Thus $\alpha = \gamma$ *can* arise, but only in extreme circumstances when $L = 256$ (see line 268 and exercise 11). Luckily L will soon become 0 in that case.

4. No problem can occur until a node invades the pool segment, which begins at address #4000000000000000; then remnants of the command line might interfere with the program's assumption that a newly allocated node is initially zero. But the data segment is able to accommodate $\lfloor (2^{61} - 2^{12} - 2^4)/(2^{12} + 24) \rfloor = 559,670,633,304,293$ nodes, so we will not live long enough to experience any problem from this "bug."

5. Line 218 calls `StackRoom` calls `StackStore` calls `MemFind`; this is as deep as it gets. Line 218 has pushed 3 registers down; `StackRoom` has pushed only 2 (since $rL = 1$ on line 097); `StackStore` has pushed 3. The value of rL on line 032 is 2 (although rL increases to 5 on line 034). Hence the register stack contains $3 + 2 + 3 + 2 = 10$ unpopped items in the worst case.

The program halts shortly after branching to `Error`; and even if it were to continue, the extra garbage at the bottom of the stack won't hurt anything—we could simply ignore it. However, we could clear the stack by providing second exits as in exercise 1.4.1–16. A simpler way to flush an entire stack is to pop repeatedly until rO equals its initial value, `Stack_Segment`.

6. 247 Div DIV x,y,z Divide y by z, signed.
 248 JMP 1F
 249 DivU PUT rD,x Put simulated rD into real rD.
 250 DIVU x,y,z Divide y by z, unsigned.
 251 1H GET t,rR
 252 STO t,g,8*rR g[rR] \leftarrow remainder.
 253 JMP XDone Finish by storing x. ■

7. (The following instructions should be inserted between line 309 of the text and the Info table, together with the answers to the next several exercises.)

```
Cswap LDOU z,g,8*rP
       LDOU y,res,0
       CMPU t,y,z
       BNZ t,1F      Branch if M8[A] ≠ g[rP].
       STOU x,res,0  Otherwise set M8[A] ← $X.
       JMP 2F
1H    STOU y,g,8*rP Set g[rP] ← M8[A].
2H    ZSZ  x,t,1   x ← result of equality test.
       JMP  XDone   Finish by storing x. ■
```

8. Here we store the simulated registers that we're keeping in actual registers. (This approach is better than a 32-way branch to see which register is being gotten; it's also better than the alternative of storing the registers every time we change them.)

```
Get  CMPU t,yz,32
      BNN  t,Error   Make sure that YZ < 32.
      STOU ii,g,8*rI Put the correct value into g[rI].
      STOU cc,g,8*rC Put the correct value into g[rC].
      STOU oo,g,8*r0 Put the correct value into g[r0].
      STOU ss,g,8*rS Put the correct value into g[rS].
      STOU uu,g,8*rU Put the correct value into g[rU].
      STOU aa,g,8*rA Put the correct value into g[rA].
      SR   t,11,3
      STOU t,g,8*rL Put the correct value into g[rL].
      SR   t,gg,3
      STOU t,g,8*rG Put the correct value into g[rG].
      SLU  t,zz,3
      LDOU x,g,t   Set x ← g[Z].
      JMP  XDone   Finish by storing x. ■

9. Put  BNZ  yy,Error  Make sure that Y = 0.
      CMPU t,xx,32
      BNN  t,Error   Make sure that X < 32.
      CMPU t,xx,rC
      BN   t,PutOK   Branch if X < 8.
      CMPU t,xx,rF
      BN   t,1F      Branch if X < 22.
      PutOK STOU z,g,xxx Set g[X] ← z.
      JMP  Update   Finish the command.
1H    CMPU t,xx,rG
      BN   t,Error   Branch if X < 19.
      SUB  t,xx,rL
      PBP  t,PutA   Branch if X = rA.
      BN   t,PutG   Branch if X = rG.
      PutL  SLU  z,z,3  Otherwise X = rL.
      CMPU t,z,ll
      CSN  ll,t,z  Set rL ← min(z, rL).
      JMP  Update   Finish the command.

OH GREG #40000
```

	PutA	CMPU t,z,OB	
	BNN	t>Error	Make sure $z \leq \#3fff$.
	SET	aa,z	Set rA $\leftarrow z$.
	JMP	Update	Finish the command.
	PutG	SRU t,z,8	
	BNZ	t>Error	Make sure $z < 256$.
	CMPU	t,z,32	
	BN	t>Error	Make sure $z \geq 32$.
	SLU	z,z,3	
	CMPU	t,z,11	
	BN	t>Error	Make sure $z \geq rL$.
	JMP	2F	
1H	SUBU	gg,gg,8	$G \leftarrow G - 1$. (\$G becomes global.)
	STCO	0,g,gg	$g[G] \leftarrow 0$. (Compare with line 216.)
2H	CMPU	t,z,gg	
	PBN	t,1B	Branch if $G < z$.
	SET	gg,z	Set rG $\leftarrow z$.
	JMP	Update	Finish the command. ■

In this case the nine commands that branch to either PutOK, PutA, PutG, PutL, or Error are tedious, yet still preferable to a 32-way switching table.

10. Pop	SUBU	oo,oo,8	
	BZ	xx,1F	Branch if X = 0.
	CMPU	t,11,xxx	
	BN	t,1F	Branch if X > L.
	ADDU	t,xxx,oo	
	AND	t,t,lring_mask	
	LDOU	y,l,t	y \leftarrow result to return.
1H	CMPU	t,oo,ss	
	PBNN	t,1F	Branch unless $\alpha = \gamma$.
	PUSHJ	0,StackLoad	
1H	AND	t,oo,lring_mask	
	LDOU	z,l,t	$z \leftarrow$ number of additional registers to pop.
	AND	z,z,#ff	Make sure $z \leq 255$ (in case of weird error).
	SLU	z,z,3	
1H	SUBU	t,oo,ss	
	CMPU	t,t,z	
	PBNN	t,1F	Branch unless z registers not all in the ring. (See note below.)
	PUSHJ	0,StackLoad	
	JMP	1B	Repeat until all necessary registers are loaded.
1H	ADDU	11,11,8	
	CMPU	t,xxx,11	
	CSN	11,t,xxx	Set $L \leftarrow \min(X, L + 1)$.
	ADDU	11,11,z	Then increase L by z.
	CMPU	t,gg,11	
	CSN	11,t,gg	Set $L \leftarrow \min(L, G)$.
	CMPU	t,z,11	
	BNN	t,1F	Branch if returned result should be discarded.
	AND	t,oo,lring_mask	
	STOU	y,l,t	Otherwise set $l[(\alpha - 1) \bmod \rho] \leftarrow y$.

1H	LDOU y,g,8*rJ	
	SUBU oo,oo,z	Decrease α by $1 + z$.
	4ADDU inst_ptr,yz,y	Set $inst_ptr \leftarrow g[rJ] + 4YZ$.
	JMP Update	Finish the command. ■

Here it is convenient to decrease oo in two steps, first by 8 and then by 8 times z . The program is complicated in general, but in most cases comparatively little computation actually needs to be done. If $\beta = \gamma$ when the second `StackLoad` call is given, we implicitly decrease β by 1 (thereby discarding the topmost item of the register stack). That item will not be needed unless it is the value being returned, but the latter value has already been placed in y .

11.	Save BNZ yz,Error	Make sure $YZ = 0$.
	CMPU t,xxx,gg	
	BN t,Error	Make sure \$X is global.
	ADDU t,oo,ll	
	AND t,t,lring_mask	
	SRU y,ll,3	
	STOU y,1,t	Set $\$L \leftarrow L$, considering $\$L$ to be local.
	INCL ll,8	
	PUSHJ 0,StackRoom	Make sure $\beta \neq \gamma$.
	ADDU oo,oo,ll	
	SET ll,0	Push down all local registers and set $rL \leftarrow 0$.
1H	PUSHJ 0,StackStore	
	CMPU t,ss,oo	
	PBNZ t,1B	Store all pushed down registers in memory.
	SUBU y,gg,8	Set $k \leftarrow G - 1$. (Here $y \equiv 8k$.)
4H	ADDU y,y,8	Increase k by 1.
1H	SET arg,ss	
	PUSHJ res,MemFind	
	CMPU t,y,8*(rZ+1)	
	LDOU z,g,y	Set $z \leftarrow g[k]$.
	PBNZ t,2F	
	SLU z,gg,56-3	
	ADDU z,z,aa	If $k = rZ + 1$, set $z \leftarrow 2^{56}rG + rA$.
2H	STOU z,res,0	Store z in $M_8[rS]$.
	INCL ss,8	Increase rS by 8.
	BNZ t,1F	Branch if we just stored rG and rA .
	CMPU t,y,c255	
	BZ t,2F	Branch if we just stored \$255.
	CMPU t,y,8*rR	
	PBNZ t,4B	Branch unless we just stored rR .
	SET y,8*rP	Set $k \leftarrow rP$.
	JMP 1B	
2H	SET y,8*rB	Set $k \leftarrow rB$.
	JMP 1B	
1H	SET oo,ss	$rO \leftarrow rS$.
	SUBU x,oo,8	$x \leftarrow rO - 8$.
	JMP XDone	Finish by storing x . ■

(The special registers saved are those with codes 0–6 and 23–27, plus (rG, rA) .)

	12. Unsave	BNZ xx,Error	Make sure X = 0.
		BNZ yy,Error	Make sure Y = 0.
		ANDNL z,#7	Make sure z is a multiple of 8.
		ADDU ss,z,8	Set rS $\leftarrow z + 8$.
		SET y,8*(rZ+2)	Set $k \leftarrow rZ + 2$. ($y \equiv 8k$)
1H		SUBU y,y,8	Decrease k by 1.
4H		SUBU ss,ss,8	Decrease rS by 8.
		SET arg,ss	
		PUSHJ res,MemFind	
		LDOU x,res,0	Set $x \leftarrow M_8[rS]$.
		CMPU t,y,8*(rZ+1)	
		PBNZ t,2F	
		SRU gg,x,56-3	If $k = rZ + 1$, initialize rG and rA.
		SLU aa,x,64-18	
		SRU aa,aa,64-18	
		JMP 1B	
2H		STOU x,g,y	Otherwise set $g[k] \leftarrow x$.
3H		CMPU t,y,8*rP	If $k = rP$, set $k \leftarrow rR + 1$.
		CSZ y,t,8*(rR+1)	If $k = rB$, set $k \leftarrow 256$.
		CSZ y,y,c256	
		CMPU t,y,gg	
		PBNZ t,1B	Repeat the loop unless $k = G$.
		PUSHJ 0,StackLoad	
		AND t,ss,lring_mask	
		LDOU x,l,t	$x \leftarrow$ the number of local registers.
		AND x,x,#ff	Make sure $x \leq 255$ (in case of weird error).
		BZ x,1F	
		SET y,x	Now load x local registers into the ring.
2H		PUSHJ 0,StackLoad	
		SUBU y,y,1	
		PBNZ y,2B	
		SLU x,x,3	
1H		SET ll,x	
		CMPU t,gg,x	
		CSN ll,t,gg	Set $rL \leftarrow \min(x, rG)$.
		SET oo,ss	Set $rO \leftarrow rS$.
		PBNZ uu,Update	Branch, if not the first time.
		BZ resuming,Update	Branch, if first command is UNSAVE.
		JMP AllDone	Otherwise clear resuming and finish. ■

A straightforward answer
is as good as a kiss of friendship.

— Proverbs 24:26

13.	517	SET xx,0	
	518	SLU t,t,55	Loop to find highest trip bit.
	519 2H	INCL xx,1	
	520	SLU t,t,1	
	521	PBNN t,2B	
	522	SET t,#100	Now xx = index of trip bit.
	523	SRU t,t,xx	t ← corresponding event bit.
	524	ANDN exc,exc,t	Remove t from exc.
	525 TakeTrip	STOU inst_ptr,g,8*rW	g[rW] ← inst_ptr.
	526	SLU inst_ptr,xx,4	inst_ptr ← xx \ll 4.
	527	INCH inst,#8000	
	528	STOU inst,g,8*rX	g[rX] ← inst + 2^{63} .
	529	AND t,f,Mem_bit	
	530	PBZ t,1F	Branch if op doesn't access memory.
	531	ADDU y,y,z	Otherwise set y ← (y + z) mod 2^{64} ,
	532	SET z,x	z ← x.
	533 1H	STOU y,g,8*rY	g[rY] ← y.
	534	STOU z,g,8*rZ	g[rZ] ← z.
	535	LDOU t,g,c255	
	536	STOU t,g,8*rB	g[rB] ← g[255].
	537	LDOU t,g,8*rJ	
	538	STOU t,g,c255	g[255] ← g[rJ]. ■
14.	Resume	SLU t,inst,40	
		BNZ t>Error	Make sure XYZ = 0.
		LDOU inst_ptr,g,8*rW	inst_ptr ← g[rW].
		LDOU x,g,8*rX	
		BN x,Update	Finish the command if rX is negative.
		SRU xx,x,56	Otherwise let xx be the ropcode.
		SUBU t,xx,2	
		BNN t,1F	Branch if the ropcode is ≥ 2 .
		PBZ xx,2F	Branch if the ropcode is 0.
		SRU y,x,28	Otherwise the ropcode is 1:
		AND y,y,#f	y ← k, the leading nybble of the opcode.
		SET z,1	
		SLU z,z,y	z ← 2^k .
		ANDNL z,#70cf	Zero out the acceptable values of z.
		BNZ z>Error	Make sure the opcode is "normal."
	1H	BP t>Error	Make sure the ropcode is ≤ 2 .
		SRU t,x,13	
		AND t,t,c255	
		CMPU y,t,ll	Branch if \$X is local.
		BN y,2F	Otherwise make sure \$X is global.
		CMPU y,t,gg	
		BN y>Error	
	2H	MOR t,x,#8	Make sure the opcode isn't RESUME.
		CMPU t,t,#F9	
		BZ t>Error	
		NEG resuming,xx	

```

        CSNN  resuming,resuming,1 Set resuming as specified.
        JMP   Update      Finish the command. ■

166  LDOU  y,g,8*rY          y ← g[rY].
167  LDOU  z,g,8*rZ          z ← g[rZ].
168  BOD   resuming,Install_Y Branch if opcode was 1.
169  OH GREG #C1<<56+(x-$0)<<48+(z-$0)<<40+1<<16+X_is_dest_bit
170  SET   f,OB              Otherwise change f to an ORI instruction.
171  LDOU  exc,g,8*rX
172  MOR   exc,exc,#20      exc ← third-from-left byte of rX.
173  JMP   XDest            Continue as for ORI. ■

```

15. We need to deal with the fact that the string to be output might be split across two or more chunks of the simulated memory. One solution is to output eight bytes at a time with `Fwrite` until reaching the last octabyte of the string; but that approach is complicated by the fact that the string might start in the middle of an octabyte. Alternatively, we could simply `Fwrite` only one byte at a time; but that would be almost obscenely slow. The following method is much better:

```

SimFputs SET  xx,0          (xx will be the number of bytes written)
           SET  z,t          Set z ← virtual address of string.
1H       SET  arg,z
           PUSHJ res,MemFind
           SET  t,res          Set t ← actual address of string.
           GO   $0,DoInst      (See below.)
           BN   t,TrapDone    If error occurred, pass the error to user.
           BZ   t,1F            Branch if the string was empty.
           ADD  xx,xx,t        Otherwise accumulate the number of bytes.
           ADDU z,z,t          Find the address following the string output.
           AND  t,z,Mem:mask
           BZ   t,1B            Continue if string ended at chunk boundary.
1H       SET  t,xx          t ← number of bytes successfully put.
           JMP  TrapDone      Finish the operation. ■

```

Here `DoInst` is a little subroutine that inserts `inst` into the instruction stream. We provide it with additional entrances that will be useful in the next answers:

```

GREG  @          Base address
:SimInst LDA   t,IOArgs  DoInst to IOArgs and return.
           JMP   DoInst
SimFinish LDA   t,IOArgs  DoInst to IOArgs and finish.
SimFclos GETA $0,TrapDone DoInst and finish.
:DoInst  PUT   rW,$0      Put return address into rW.
           PUT   rX,inst    Put inst into rX.
           RESUME 0         And do it. ■

```

16. Again we need to worry about chunk boundaries (see the previous answer), but a byte-at-a-time method is tolerable since file names tend to be fairly short.

```

SimFopen PUSHJ 0,GetArgs  (See below.)
           ADDU xx,Mem:alloc,Mem:nodesize
           STOU xx,IOArgs
           SET   x,xx          (We'll copy the file name into this open space.)
1H       SET   arg,z
           PUSHJ res,MemFind

```

```

LDBU  t,res,0
STBU  t,x,0      Copy byte M[z].
INCL  x,1
INCL  z,1
PBNZ  t,1B      Repeat until the string has ended.
GO    $0,SimInst
3H    STCO 0,x,0      Now open the file.
      CMPU  z,xx,x
      SUB   x,x,8
      PBN   z,3B      Repeat until it is surely obliterated.
      JMP   TrapDone   Pass the result t to the user. ■

```

Here `GetArgs` is a subroutine that will be useful also in the implementation of other I/O commands. It sets up `IOArgs` and computes several other useful results in global registers.

```

:GetArgs GET   $0,rJ      Save the return address.
SET   y,t        y ← g[255].
SET   arg,t
PUSHJ res,MemFind
LDOU  z,res,0      z ← virtual address of first argument.
SET   arg,z
PUSHJ res,MemFind
SET   x,res        x ← internal address of first argument.
STO   x,IOArgs
SET   xx,Mem:Chunk
AND   zz,x,Mem:mask
SUB   xx,xx,zz      xx ← bytes from x to chunk end.
ADDU  arg,y,8
PUSHJ res,MemFind
LDOU  zz,res,0      zz ← second argument.
STOU  zz,IOArgs+8   Convert IOArgs to internal form.
PUT   rJ,$0        Restore the return address.
POP   0           ■

```

17. This solution, which uses the subroutines above, works also for `SimFwrite(1)`.

```

SimFread PUSHJ 0,GetArgs   Massage the input arguments.
SET   y,zz        y ← number of bytes to read.
1H    CMP   t,xx,y
      PBNN t,SimFinish Branch if we can stay in one chunk.
      STO   xx,IOArgs+8  Oops, we have to work piecewise.
      SUB   y,y,xx
      GO    $0,SimInst
      BN   t,1F        Branch if an error occurs.
      ADD   z,z,xx
      SET   arg,z
      PUSHJ res,MemFind
      STOU  res,IOArgs  Reduce to the previous problem.
      STO   y,IOArgs+8
      ADD   xx,Mem:mask,1
      JMP   1B

```

```
1H      SUB    t,t,y      Compute the correct number of missing bytes.
        JMP    TrapDone
```

```
SimFwrite IS SimFread ;SimFseek IS SimFclos e ;SimFtell IS SimFclos e ■
```

(The program assumes that no file-reading error will occur if the first `Fread` was successful.) Analogous routines for `SimGets`, `SimGetws`, and `SimPutws` can be found in the file `sim.mms`, which is one of many demonstration files included with the author's MMIXware programs.

18. The stated algorithms will work with any MMIX program for which the number of local registers, L , never exceeds $\rho - 1$, where ρ is the `lring_size`.
19. In all three cases the preceding instruction is `INCL 11,8`, and a value is stored in location $1 + ((\text{oo} + 11) \wedge \text{lring_mask})$. So we could shorten the program slightly.

```
20. 560 1H GETA  t,OctaArgs
      561  TRAP  0,Fread,Infile  Input  $\lambda$  into  $g[255]$ .
      562  BN    t,9F          Branch if end of file.
      563  LDOU  loc,g,c255    $\text{loc} \leftarrow \lambda$ .
      564 2H GETA  t,OctaArgs
      565  TRAP  0,Fread,Infile  Input an octabyte  $x$  into  $g[255]$ .
      566  LDOU  x,g,c255
      567  BN    t,Error       Branch on unexpected end of file.
      568  SET   arg,loc
      569  BZ    x,1B          Start a new sequence if  $x = 0$ .
      570  PUSHJ res,MemFind
      571  STOU  x,res,0       Otherwise store  $x$  in  $M_8[\text{loc}]$ .
      572  INCL  loc,8         Increase  $\text{loc}$  by 8.
      573  JMP   2B          Repeat until encountering a zero.
      574 9H TRAP  0,Fclose,Infile Close the input file.
      575  SUBU  loc,loc,8     Decrease  $\text{loc}$  by 8. ■
```

Also put “OctaArgs OCTA Global+8*255,8” in some convenient place.

21. Yes it is, up to a point; but the question is interesting and nontrivial.

To analyze it quantitatively, let `sim.mms` be the simulator in MMIXAL, and let `sim.mmo` be the corresponding object file produced by the assembler. Let `Hello.mmo` be the object file corresponding to Program 1.3.2'H. Then the command line ‘Hello’ presented to MMIX’s operating system will output ‘Hello, world’ and stop after $\mu + 17v$, not counting the time taken by the operating system to load it and to take care of input/output operations.

Let `Hello0.mmb` be the binary file that corresponds to the command line ‘Hello’, in the format of exercise 20. (This file is 176 bytes long.) Then the command line ‘sim Hello0.mmb’ will output ‘Hello, world’ and stop after $168\mu + 1699v$.

Let `Hello1.mmb` be the binary file that corresponds to the command line ‘sim Hello0.mmb’. (This file is 5768 bytes long.) Then the command line ‘sim Hello1.mmb’ will output ‘Hello, world’ and stop after $10549\mu + 169505v$.

Let `Hello2.mmb` be the binary file that corresponds to the command line ‘sim Hello1.mmb’. (This file also turns out to be 5768 bytes long.) Then the command line ‘sim Hello2.mmb’ will output ‘Hello, world’ and stop after $789739\mu + 15117686v$.

Let `Hello3.mmb` be the binary file that corresponds to the command line ‘sim Hello2.mmb’. (Again, 5768 bytes.) Then the command line ‘sim Hello3.mmb’ will output ‘Hello, world’ if we wait sufficiently long.

Now let `recurse.mmb` be the binary file that corresponds to the command line ‘`sim recurse.mmb`’. Then the command line ‘`sim recurse.mmb`’ runs the simulator simulating itself simulating itself simulating itself … ad infinitum. The file handle `Infile` is first opened at time $3\mu + 13v$, when `recurse.mmb` begins to be read by the simulator at level 1. That handle is closed at time $1464\mu + 16438v$ when loading is complete; but the simulated simulator at level 2 opens it at time $1800\mu + 19689v$, and begins to load `recurse.mmb` into simulated simulated memory. The handle is closed again at time $99650\mu + 1484347v$, then reopened by the simulated simulated simulator at time $116999\mu + 1794455v$. The third level finishes loading at time $6827574\mu + 131658624v$ and the fourth level starts at time $8216888\mu + 159327275v$.

But the recursion cannot go on forever; indeed, the simulator running itself is a finite-state system, and a finite-state system cannot produce `Fopen`-`Fclose` events at exponentially longer and longer intervals. Eventually the memory will fill up (see exercise 4) and the simulation will go awry. When will this happen? The exact answer is not easy to determine, but we can estimate it as follows: If the k th level simulator needs n_k chunks of memory to load the $(k+1)$ st level simulator, the value of n_{k+1} is at most $4 + \lceil (2^{12} + 16 + (2^{12} + 24)n_k)/2^{12} \rceil$, with $n_0 = 0$. We have $n_k = 6k$ for $k < 30$, but this sequence eventually grows exponentially; it first surpasses 2^{61} when $k = 6066$. Thus we can simulate at least 100^{6065} instructions before any problem arises, if we assume that each level of simulation introduces a factor of at least 100 (see exercise 2).

22. The pairs (x_k, y_k) can be stored in memory following the trace program itself, which should appear after all other instructions in the text segment of the program being traced. (The operating system will give the trace routine permission to modify the text segment.) The main idea is to scan ahead from the current location in the traced program to the next branch or `G0` or `PUSH` or `POP` or `JMP` or `RESUME` or `TRIP` instruction, then to replace that instruction temporarily in memory with a `TRIP` command. The tetrabytes in locations #0, #10, #20, …, #80 of the traced program are changed so that they jump to appropriate locations within the trace routine; then all control transfers will be traced, including transfers due to arithmetic interrupts. The original instructions in those locations can be traced via `RESUME`, as long as they are not themselves `RESUME` commands.

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

- : (colon), 61–62, 65, 80.
- " (double-quote), 31, 37, 44, 72, 100.
- _ (underscore), 37.
- @ (at sign), 15, 35, 38, 81.
- \$0, 31, 58.
- \$1, 31, 58.
- 2ADDU (times 2 and add unsigned), 9.
- 4ADDU (times 4 and add unsigned), 9.
- 8ADDU (times 8 and add unsigned), 9.
- 16ADDU (times 16 and add unsigned), 9.
- \$255, 34, 40–43, 56, 68, 114.
- μ (average memory access time), 22.
- ϕ (golden ratio), 8, 47.
- v (instruction cycle time), 22.
- Absolute address, 15.
- Absolute difference, 26.
- Absolute value, 26, 27.
- ACE computer, 65.
- ADD, 8.
- Addition, 8, 12, 14, 25.
- Addition chains, 98.
- ADDU (add unsigned), 8.
- Adobe Systems, 74.
- Ahrens, Wilhelm Ernst Martin Georg, 48.
- ALGOL language, 74.
- Algol W language, iv.
- Alhazen, *see* Ibn al-Haytham.
- Aliasing, 108.
- Alignment, 39, 44.
- Alpha 21164 computer, 2.
- AMD 29000 computer, 2.
- AND (bitwise and), 10.
- ANDN (bitwise and-not), 10.
- ANDNH (bitwise and-not high wyde), 14.
- ANDNL (bitwise and-not low wyde), 14.
- ANDNMH (bitwise and-not medium high wyde), 14.
- ANDNML (bitwise and-not medium low wyde), 14.
- ANSI: The American National Standards Institute, 12.
- Arabic numerals, 44.
- Arabic script, 44, 100.
- Arguments, 54.
- Arithmetic exceptions, 18, 89.
- Arithmetic operators of MMIX, 8–9.
- Arithmetic overflow, 6, 7, 18, 25, 27, 65, 84, 95, 109.
- Arithmetic status register, 18.
- ASCII: American Standard Code for Information Interchange, iv, 3, 26, 32, 34, 37, 44, 67.
- Assembly language for MMIX, 28–44.
- Assembly program, 29, 30, 40.
- Associative law: $(a \circ b) \circ c = a \circ (b \circ c)$, 11.
- At sign (@), 15, 35, 38, 81.
- Atomic instruction, 17.
- b(x), 11.
- Ball, Walter William Rouse, 48.
- Base address, 35, 39.
- BDIF (byte difference), 11, 26, 101.
- Bertrand, Joseph Louis François, postulate, 100.
- BEV (branch if even), 15.
- Bidirectional typesetting, 44.
- Bienstock, Daniel, 104.
- Big-endian convention: Most significant byte first, 4–7, 116.
- Binary file, 41.
 - for programs, 90, 92–93, 125.
- Binary number system, 4.
- Binary operators in MMIXAL, 38.
- Binary radix point, 8, 24.
- Binary-to-decimal conversion, 37.
- BinaryRead mode, 43.
- BinaryReadWrite mode, 43.
- BinaryWrite mode, 43.
- Bit: “Binary digit”, either zero or unity, 2.
- Bit difference, 26.
- Bit reversal, 26, 97.
- Bit vectors, 10.
- Bitwise difference, 14.
- Bitwise operators of MMIX, 10, 14, 25.
- Blank space, 26, 40, 67.
- BN (branch if negative), 15.
- BNN (branch if nonnegative), 15.
- BNP (branch if nonpositive), 15.
- BNZ (branch if nonzero), 15.
- BOD (branch if odd), 15.
- Boolean matrix, 11, 96.
- Bootstrap register, 18.
- Bourne, Charles Percy, 107.
- BP (branch if positive), 15.
- Branch operators of MMIX, 15, 85.
- BSPEC (begin special data), 62.
- Buchholz, Werner, 94.
- Byte: An 8-bit quantity, 3, 24, 94.
- Byte difference, 11, 26.
- BYTE operator, 31, 39.
- Byte reversal, 12.
- BZ (branch if zero), 15.

- C language, iv, 45.
 C++ language, iv.
 Cache memory, 17, 22–23, 72, 98, 105, 107.
 Calendar, 49.
 Calling sequence, 54–56, 60, 68–70.
 Carry, 25.
 Cauchy, Augustin Louis, 105.
 Ceiling, 13.
 Character constant, 37.
 Chess, 66.
 Chung, Fan Rong King (鍾金芳蓉), 104.
 Chunks, 77, 123.
 Clavius, Christopher, 49.
 Clipper C300 computer, 2.
 Clock register, 19, 76, 112.
CMP (compare), 9.
CMPU (compare unsigned), 9, 113.
 Colon (:), 61, 65, 80.
 Command line arguments, 31, 90, 125.
 Comments, 29.
 Commutative law: $a \circ b = b \circ a$, 95.
 Comparison operators of MMIX, 9,
 13, 25, 113.
 Compiler algorithms, 62, 74.
 Complement, 10, 24.
 Complete MMIX program, 30, 45.
 Conditional operators of MMIX, 10, 26.
 Conversion operators of MMIX, 13.
 Conway, Melvin Edward, 35.
 Copying a string, 47.
 Coroutines, 66–73.
 linkage, 66, 72–73.
 Counting bits, 11.
 Coxeter, Harold Scott Macdonald, 48.
 CRAY I computer, 2.
 Crossword puzzle, 50–51.
 Cryptanalysis, 47.
CSEV (conditional set if even), 10.
CSN (conditional set if negative), 10.
CSNN (conditional set if nonnegative), 10.
CSNP (conditional set if nonpositive), 10.
CSNZ (conditional set if nonzero), 10.
CSOD (conditional set if odd), 10.
CSP (conditional set if positive), 10.
CSWAP (compare and swap), 17, 91.
CSZ (conditional set if zero), 10.
 Current prefix, 61, 65.
 Cycle counter, 19.
 Cyclic shift, 26.
- D_BIT** (integer divide check bit), 18.
 Dallos, József, 97.
 Data segment of memory, 36, 57,
 76–77, 81, 117.
 Debugging, 64–65, 73, 91.
 Decimal constant, 37.
 Defined symbol, 37.
 Denormal floating point number, 12, 89.
 Dershowitz, Nachum (נחום דרשוי), 111.
- Dickens, Charles John Huffam, iii.
 Dictionaries, iii.
 Dijkstra, Edsger Wijbe, 63.
 Discrete system simulators, 76.
DIV (divide), 8, 24–25.
Divide check, 8, 18.
 Dividend register, 9.
 Division, 9, 13, 24–25, 49, 91.
 by small constants, 25.
 by zero, 18.
 converted to multiplication, 25, 111.
DIVU (divide unsigned), 8.
 Double-quote ("), 31, 37, 44, 72, 100.
 Dull, Brutus Cyclops, 25.
 DVWILOZX, 18, 27, 89, 92.
 Dynamic traps, 19.
- Easter date, 49.
 Emulator, 75.
 Enable bits, 18, 85.
 Ending a program, 19, 31.
 Entrances to subroutines, 52–57, 123.
 Epsilon register, 13.
 Equivalent of MMIXAL symbol, 38.
 Error recovery, 91.
ESPEC (end special data), 62.
 Evaluation of powers, 28, 98.
 Evans, Arthur, Jr., 74.
 Event bits, 18, 85.
 Exabyte, 94.
 Exceptions, 18, 89.
 Execution register, 18.
 Exiting from a program, 19, 31.
 Exits from subroutines, 52–57, 115.
 Exponent of a floating point number, 12.
 Exponentiation, 28.
EXPR field of MMIXAL line, 29, 38.
 Expression, in MMIXAL, 38.
 Extending the sign bit, 7, 9, 95.
- $f(x)$, 12.
FADD (floating add), 12.
 Fallacies, 95.
 Farey, John, 105.
 series, 47.
 Fascicles, iii.
Fclose operation, 41, 43.
FCMP (floating compare), 13, 98.
FCMPE (floating compare with respect
 to epsilon), 13.
FDIV (floating divide), 12.
FEQL (floating equal to), 13, 98.
FEQLE (floating equivalent with respect
 to epsilon), 13.
Fgets operation, 42, 43.
Fgetws operation, 42, 43.
 Fibonacci, Leonardo, of Pisa.
 numbers, 47, 66.
 Filters, 71.
 Finite fields, 26.

- FINT** (floating integer), 13, 23.
FIX (convert floating to fixed), 13.
Fixed point arithmetic, 45.
FIXU (convert floating to fixed unsigned), 13.
Flag bits, 82, 87.
Floating binary number, 12.
Floating point arithmetic, 12–13, 44, 45, 89.
Floating point operators of MMIX, 12–13.
FLOT (convert fixed to floating), 13.
FLOTU (convert fixed to floating unsigned), 13, 97.
Floyd, Robert W, 98.
FMUL (floating multiply), 12.
Fopen operation, 41, 43, 92.
Ford, Donald Floyd, 107.
Forward reference, *see* Future reference.
Fput`s` operation, 42, 43, 92.
Fput`ws` operation, 42, 43.
Fraction of a floating point number, 12.
Frame pointer, 58, 115.
Fread operation, 42, 43, 92.
Fredman, Michael Lawrence, 104.
FREM (floating remainder), 13, 23, 44, 111.
Fseek operation, 42, 43.
FSQRT (floating square root), 13.
FSUB (floating subtract), 12.
Ftell operation, 43.
Fuchs, David Raymond, 27, 74.
FUN (floating unordered), 13, 98.
FUNE (floating unordered with respect to epsilon), 13.
Future reference, 37, 39.
Fwrite operation, 42, 43, 124.
- Generalized matrix product, 11, 26.
GET (get from special register), 19, 92.
GETA (get address), 20, 100.
Gigabyte, 94.
Global registers, 16, 34, 58, 65, 79, 80, 84, 92.
Global threshold register, 16.
G0, 15, 26, 53–58.
Gove, Philip Babcock, iii.
Graphical display, 50–51.
Graphics, 11, 26.
GREG (allocate global register), 34–35, 39, 62.
- Half-bytes, 24.
Halt operation, 31, 43.
Handles, 41.
Handlers, 18, 65, 89.
Hardy, Godfrey Harold, 105.
Harmonic convergence, 48.
Harmonic series, 48–49.
Haros, C., 105.
Heller, Joseph, 3.
Hello, world, 30–32, 125.
Hennessy, John LeRoy, v.
Hexadecimal constants, 37.
Hexadecimal digits, 3, 24.
Hexadecimal notation, 3, 19.
High tetra arithmetic, 97.
Hill, Robert, 111.
Himult register, 8.
Hints to MMIX, 16–17.
Hitachi SuperH4 computer, 2.
Hofri, Micha (מיכה חפר), 104.
- I_BIT** (invalid floating operation bit), 18, 98.
IBM 601 computer, 2.
IBM 801 computer, 2.
Ibn al-Haytham, Abū ‘Alī al-Hasan (= Alhazen, أبو علي الحسن بن الهيثم), 48.
IEC: The International Electrotechnical Commission, 3.
IEEE: The Institute of Electrical and Electronics Engineers.
floating point standard, 12, 89.
Immediate constants, 13–14, 19.
INCH (increase by high wyde), 14.
INCL (increase by low wyde), 14.
INCMH (increase by medium high wyde), 14.
INCML (increase by medium low wyde), 14.
Inexact exception, 18, 89.
Ingalls, Daniel Henry Holmes, 109.
Initialization, 31, 91.
of routines, 70.
Infinite floating point number, 12.
int *x*, 13.
Input-output operations, 19, 31, 40–43, 92.
Instruction, machine language: A code that, when interpreted by the circuitry of a computer, causes the computer to perform some action.
in MMIX, 5–28.
numeric form, 27–29, 44.
symbolic form, 28–40.
Integer overflow, 6, 7, 18, 25, 27, 65, 84, 95, 109.
Intel i960 computer, 2.
Internet, ii, v.
Interpreter, 73–75.
Interrupt mask register, 19.
Interrupt request register, 19.
Interrupts, 18–19, 86, 89, 92.
Interval counter, 19.
Invalid floating operation, 18.
IS, 30, 34, 39.
ISO: The International Organization for Standardization, 3.
Ivanović, Vladimir Gresham, v.
Iverson, Kenneth Eugene, 11.

- Jacquet, Philippe Pierre, 104.
 Java language, iv, 45.
JMP (jump), 15.
Joke, 72.
 Josephus, Flavius, son of Matthias
 $(\text{יְהוָה בָּנָי} = \text{Φλάβιος Ἰωσήπος Ματθέου})$, problem, 48.
 Jump operators of MMIX, 15.
 Jump table, 86–87.
 Jump trace, 93.
- Kernel space, 36.
 Kernighan, Brian Wilson, 23.
 Kilobyte, 24, 94.
 KKB (large kilobyte), 94.
 Knuth, Donald Ervin (高德纳), i, v, 45, 65, 74, 89.
- LABEL** field of MMIXAL line, 29, 38.
 Large kilobyte, 94.
 Large programs, 63–65.
LDA (load address), 7, 9, 100.
LDB (load byte), 6.
LDBU (load byte unsigned), 7.
LDHT (load high tetra), 7, 24, 97.
LDO (load octa), 6.
LDOU (load octa unsigned), 7.
LDSF (load short float), 13.
LDT (load tetra), 6.
LDTU (load tetra unsigned), 7.
LDUNC (load octa uncached), 17.
LDVTS (load virtual translation status), 17.
LDW (load wyde), 6.
LDWU (load wyde unsigned), 7.
 Leaf subroutine, 57, 65, 80.
 Library of subroutines, 52, 61, 62, 91.
 Lilius, Aloysius, 49.
 Linked allocation, 77–78.
 Literate programming, 45, 65.
 Little-endian convention: Least significant
 byte first, *see* Bidirectional typesetting.
 Byte reversal.
 Loader, 36.
 Loading operators of MMIX, 6–7.
LOC (change location), 30, 39.
LOCAL (guarantee locality), 62.
 Local registers, 16, 58, 65, 80, 84, 92.
 ring of, 76, 79–81, 92.
 Local symbols, 35–37, 43.
 Local threshold register, 16.
 Loop optimization, 115.
- $m(x)$, 11.
 Machine language, 2.
 Magic squares, 47–48.
Main location, 31, 91.
 Marginal registers, 16, 58, 65, 80, 84, 97.
 Matrix: A two-dimensional array, 46, 106.
 Matrix multiplication, generalized, 11, 26.
- Maximum, 26.
 subroutine, 28–29, 52–56.
 Megabyte, 24, 94.
MemFind subroutine, 77–78, 91, 116–117.
 Memory: Part of a computer system
 used to store data, 4–6.
 address, 6.
 hierarchy, 17, 22–23, 72, 98, 105, 107.
 Memory stack, 57–58, 115.
 Mems: Memory accesses, 22.
 Meta-simulator, 22–23, 47, 76.
METAPOST language, 51.
 Minimum, 26.
 Minus zero, 13.
 MIPS 4000 computer, 2.
MIX computer, iv.
 .mmb (MMIX binary file), 125.
 MMB (Large megabyte), 94.
 MMIX computer, iv, 2–28.
 MMIX simulator, 22–23, 30.
 in MMIX, 75–93.
MMIXAL: MMIX Assembly Language,
 28–44, 61–62.
 MMIXmasters, v, 51, 105, 111.
 MMIXware document, 2.
 .mmo (MMIX object file), 30, 125.
 .mms (MMIX symbolic file), 30, 125.
MOR (multiple or), 12, 23, 26.
 Motorola 88000 computer, 2.
 Move-to-front heuristic, 77–78.
Mu (μ), 22.
MUL (multiply), 8.
 Multipass algorithms, 70–72, 74.
 Multiple entrances, 56, 123.
 Multiple exits, 56–57, 60, 115.
 Multiplex mask register, 11.
 Multiplication, 8, 12, 25, 85.
 by small constants, 9, 25.
 Multiway decisions, 45, 46, 82, 86–88, 119.
MULU (multiply unsigned), 8, 25.
 Murray, James Augustus Henry, iii.
MUX (multiplex), 11.
MXOR (multiple exclusive-or), 12, 23, 26.
- NaN (Not-a-Number), 12, 98.
NAND (bitwise not-and), 10.
NEG (negate), 9.
 Negation, 9, 24.
NEGU (negate unsigned), 9.
 Newline, 32, 42.
NNIX operating system, 28, 31.
 No-op, 21, 28.
 Nonlocal **goto** statements, 66, 91, 117.
NOR (bitwise not-or), 10.
 Normal floating point number, 12.
 Not-a-Number, 12, 98.

Notational conventions:

$b(x)$, 11.
 $f(x)$, 12.
 $\text{int } x$, 13.
 $m(x)$, 11.
 $s(x)$, 6, 24.
 $t(x)$, 11.
 $u(x)$, 6, 24.
 $v(x)$, 10.
 $\bar{v}(x)$, 10.
 $w(x)$, 11.
 $x \dot{-} y$, 11.
 $x \gg y$, 9.
 $x \ll y$, 9.
 $x \wedge y$, 10.
 $x \vee y$, 10.
 $x \oplus y$, 10.
 $x \text{ rem } y$, 13.
XYZ, 6.
YZ, 5–6.
NXOR (bitwise not-exclusive-or), 10.
Nybble: A 4-bit quantity, 24.
Nyp: A 2-bit quantity, 94.

0_BIT (floating overflow bit), 18.
O’Beirne, Thomas Hay, 111.
Object file, 30–31, 125.
Octa: Short form of “octabyte”, 4.
0CTA operator, 39.
Octabyte: A 64-bit quantity, 4.
0DIF (octa difference), 11, 102.
Oops, 22.
0P field of MMIXAL line, 29, 38.
Opcode: Operation code, 5, 19.
chart, 20.
Operands, 5, 83–84.
Operating system, 28, 36, 40–43.
Optimization of loops, 47.
OR (bitwise or), 10.
ORH (bitwise or with high wyde), 14.
ORL (bitwise or with low wyde), 14.
ORMH (bitwise or with medium high wyde), 14.
ORML (bitwise or with medium low wyde), 14.
ORN (bitwise or-not), 10.
Overflow, 6, 7, 18, 25, 27, 65, 84, 95, 109.
Oxford English Dictionary, iii.

Packed data, 82, 87–88.
Page fault, 114.
Parameters, 54.
Parity, 26.
Pascal language, iv.
Pass, in a program, 70–72.
Patt, Yale Nance, 98.
PBEV (probable branch if even), 16.
PBN (probable branch if negative), 15.
PBNN (probable branch if nonnegative), 15.
PBNP (probable branch if nonpositive), 16.

PBNZ (probable branch if nonzero), 16.
PBOD (probable branch if odd), 15.
PBP (probable branch if positive), 15.
PBZ (probable branch if zero), 15.
Petabyte, 94.
Phi (ϕ), 8, 47.
Pipe, 71.
Pipeline, 22, 47, 76, 98.
Pixel values, 11, 26.
PL/360 language, 45.
PL/MMIX language, 45, 63.
Pool segment of memory, 36, 117.
POP (pop registers and return), 16,
53, 59, 73, 92.
Population counting, 11.
PostScript language, 74.
POWER 2 computer, 2.
Power of number, evaluation, 28.
Predefined symbols, 36–38, 43.
Prediction register, 17.
PREFIX specification, 61–62, 65, 77–78, 80.
Prefetching, 17, 22.
Prefixes for units of measure, 94.
PREGO (prefetch to go), 17.
PRELD (preload data), 17.
PREST (prestore data), 17.
Primary, in MMIXAL, 38.
Prime numbers, program to compute,
32–34, 37.
Privileged instructions, 46, 76.
Probable branch, 15–16, 22, 26, 85.
Profile of a program: The number of
times each instruction is performed,
29, 31, 93, 98.
Program construction, 63–65.
Programming languages, iv, 63.
Pseudo-operations, 30–31.
Purdy, Gregor Neal, 94.
PUSHGO (push registers and go), 16,
65, 73, 85–86.
PUSHJ (push registers and jump), 16,
53, 59, 73, 85–86.
PUT (put into special register), 19, 92.
Quick, Jonathan Horatio, 44.

rA (arithmetic status register), 18, 28.
RA (relative address), 15.
Radix point, 8, 24.
Randell, Brian, 74.
Randolph, Vance, 28.
Rational numbers, 47.
rB (bootstrap register for trips), 18.
rBB (bootstrap register for traps), 18.
rC (cycle counter), 19, 112.
rD (dividend register), 9.
rE (epsilon register), 13.
Reachability, 51.
Read-only access, 36.

Recursive use of subroutines, 57, 66, 125–126.
 Register \$0, 31, 58.
 Register \$1, 31, 58.
 Register \$255, 34, 40–43, 56, 68, 114.
 Register number, 34, 58.
 Register stack, 16, 58–61, 65–66, 70, 73, 78–81, 84–86, 115.
 Register stack offset, 17.
 Register stack pointer, 17.
 Registers: Portions of a computer's internal circuitry in which data is most accessible.
 of MMIX, 4–5, 21, 23, 76, 79.
 saving and restoring, 55; *see also SAVE, UNSAVE.*
 Reingold, Edward Martin (ר'יינולד צחק משה בן חיים), 111.
 Relative addresses, 15–16, 20, 30, 83, 87, 99.
 Remainder, 8, 13, 49.
 Remainder register, 8.
 Replicated routines, 72.
 Reprogramming, 75.
 RESUME (resume after interrupt), 19, 84, 92, 114, 126.
 Return-jump register, 16.
 Reversal of bits and bytes, 12, 26, 97.
 Rewinding a file, 42.
 Rewrites, v, 64.
 rG (global threshold register), 16, 58, 92.
 rH (himult register), 8, 28, 85, 94.
 rI (interval counter), 19.
 Ring of local registers, 76, 79–81, 92.
 RISC: Reduced Instruction Set Computer, 24.
 RISC II computer, 2.
 rJ (return-jump register), 16, 60, 80, 81.
 rK (interrupt mask register), 19, 90–91.
 rL (local threshold register), 16, 28, 58, 79, 92, 97, 117.
 rM (multiplex mask register), 11.
 rN (serial number), 19.
 rO (register stack offset), 17, 79.
 Rokicki, Tomas Gerhard, 74.
 Roman numerals, 2, 3.
 Ropcodes, 19, 92.
 ROUND_DOWN mode, 13.
 ROUND_NEAR mode, 13, 37.
 ROUND_OFF mode, 13.
 ROUND_UP mode, 13.
 Rounding, 13, 18, 47, 48.
 Row major order, 46.
 rP (prediction register), 17.
 rQ (interrupt request register), 19.
 rR (remainder register), 8.
 rS (register stack pointer), 17, 79.
 rT (trap address register), 18, 90–91.
 rTT (dynamic trap address register), 19, 90–91.

rU (usage counter), 19.
 Running time, 20–23.
 Russell, Lawford John, 74.
 rV (virtual translation register), 20, 90–91.
 rW (where-interrupted register for traps), 18.
 rWW (where-interrupted register for traps), 18.
 rX (execution register for traps), 18.
 rXX (execution register for traps), 18.
 rY (Y operand register for traps), 18.
 rYY (Y operand register for traps), 18.
 rZ (Z operand register for traps), 18.
 rZZ (Z operand register for traps), 18.
 s(x), 6, 24.
 SADD (sideways add), 11.
 Saddle point, 46.
 Saturating addition, 26.
 Saturating subtraction, 11.
 SAVE (save process state), 16, 61, 92, 114, 116.
 Saving and restoring registers, 55; *see also SAVE, UNSAVE.*
 Scalar variables, 61.
 Schäffer, Alejandro Alberto, 104.
 Segments of user space, 36.
 Self-modifying code, iv, 28, 93.
 Self-organizing list search, 77–78.
 Self-reference, 126, 132.
 Sequential array allocation, 46.
 Serial number register, 19.
 SET, 14, 99.
 Set difference, 25.
 Set intersection, 25.
 Set union, 25.
 SETH (set high wyde), 14.
 SETL (set low wyde), 14, 100.
 SETMH (set medium high wyde), 14, 97.
 SETML (set medium low wyde), 14.
 SFLOT (convert fixed to short float), 13.
 SFLOTU (convert fixed to short float unsigned), 13.
 Shift operators of MMIX, 9.
 Shor, Peter Williston, 104.
 Short float format, 12–13.
 Sideways addition, 11.
 Sign extension, 7, 9, 95.
 Sign of floating point number, 12.
 Signed integers, 4, 6–7, 25.
 Sikes, William, iii.
 Simon, Marvin Neil, v.
 Simulation of computers, 75–76.
 Sites, Richard Lee, v.
 SL (shift left), 9, 25.
 SLU (shift left unsigned), 9, 25.
 Small constant numbers, 9, 13.
 division by, 25.
 multiplication by, 9, 25.
 Sparc 64 computer, 2.
 Special registers of MMIX, 5, 19, 21, 76, 118.

Square root, 13.
SR (shift right), 9, 25.
SRU (shift right unsigned), 9, 25.
 Stack offset register, 79.
 Stack operators of MMIX, 16–17.
 Stack pointer register, 57–58, 79.
 Stack segment of memory, 36, 61, 114, 117.
 Stacks, *see* Memory stack, Register stack.
 Stallng a pipeline, 108.
 Standard error file, 41.
 Standard input file, 41.
 Standard output file, 31, 41.
 Starting a program, 31, 70, 91.
STB (store byte), 7.
STBU (store byte unsigned), 8.
STCO (store constant octabyte), 8.
StdErr (standard error file), 41.
StdIn (standard input file), 41.
StdOut (standard output file), 30–31, 41.
STHT (store high tetra), 8, 24, 97.
STO (store octa), 7.
 Storing operators of MMIX, 7–8.
STOU (store octa unsigned), 8.
 Stretch computer, 94.
 String constant in MMIXAL, 31, 37, 100.
 String manipulation, 26, 47.
 Strong binary operators, 38.
 StrongArm 110 computer, 2.
 Structured symbols, 61–62, 65, 77–78, 80.
STSF (store short float), 13.
STT (store tetra), 7.
STTU (store tetra unsigned), 8.
STUNC (store octa uncached), 17.
STW (store wyde), 7.
STWU (store wyde unsigned), 8.
SUB (subtract), 8.
 Subroutines, 30, 45, 52–70, 75, 77–81, 92.
 linkage of, 52–61.
 Subsets, representation of, 25.
 Subtraction, 8, 12, 25.
SUBU (subtract unsigned), 8.
 Superscalar machine, 108.
 Suri, Subhash (सुभाष सूरी), 104.
 Switching tables, 45, 46, 82, 86–88, 119.
SWYM (sympathize with your machinery), 21.
SYNC (synchronize), 17, 86.
SYNCD (synchronize data), 17.
SYNCID (synchronize instructions and
 data), 17, 28.
 System operators of MMIX, 17.
 System/360 computer, 45.

$t(x)$, 11.
 Table-driven computation, 45, 46, 82,
 86–88, 119.
TDIF (tetra difference), 11.
 Terabyte, 94.
 Term, in MMIXAL, 38.
 Terminating a program, 19, 31.

Tetra: Short form of “tetrabyte”, 4.
 Tetra difference, 11.
TETRA operator, 39, 72.
 Tetrabyte: A 32-bit quantity, 4.
 Tetrabyte arithmetic, 27.
TEX, 65, 74–75.
 Text file, 41.
 Text segment of memory, 36, 77, 81.
TextRead mode, 43.
TextWrite mode, 43.
 Threads, 72.
 Trace routine, 64, 93.
 Traffic signals, 50.
TRAP (force trap interrupt), 18–19, 40, 86–87.
 Trap address register, 18.
 Trap handlers, 18–19.
TRIP (force trip interrupt), 18, 86.
 Trip handlers, 18, 89.
 Trip interrupts, 65, 92.
 Turing, Alan Mathison, 65.
 Twist, Oliver, iii.
 Two’s complement notation, 4, 24.

$u(x)$, 6, 24.
U_BIT (floating underflow bit), 18, 85, 89.
U_Handler: Address of an underflow trip, 89.
 UCS: Universal Multiple-Octet Coded
 Character Set, 3.
 Underflow, 18, 89.
 Underscore (_), 37.
 Unicode, 3, 26, 37, 44.
 Units of measure, 94.
 UNIVAC I computer, 35.
 UNIX operating system, 71, 114.
 Unpacking, 82.
 Unrolling a loop, 107.
UNSAVE (restore process state), 16, 61,
 90, 92, 116.
 Unsigned integers, 4, 6–8.
 Upsilon (ν), 22.
 Usage counter, 19.
 User space, 36.

$v(x)$, $\bar{v}(x)$, 10.
V_BIT (integer overflow bit), 18.
 Valid MMIX instruction, 46.
 Van Wyk, Christopher John, 23.
 Vector, 10.
 Victorius of Aquitania, 111.
 Virtual address translation, 17.
 Virtual machine, 73.
 Virtual translation register, 20.

$w(x)$, 11.
W_BIT (float-to-fix overflow bit), 18.
W_Handler: Address of a float-to-fix overflow trip, 37.
WDIF (wyde difference), 11.
 Weak binary operators, 38.
 Webster, Noah, iii.
 Where-interrupted register, 18.
 Whitespace character, 67.
 Wide strings, 42.
 Wilson, George Pickett, 28.
 Wirth, Niklaus Emil, 45, 63.
 Wordsworth, William, 24.
 Wright, Edward Maitland, 105.
 Wyde: A 16-bit quantity, 4.
 Wyde difference, 11.
 Wyde immediate, 14.
 WYDE operator, 39.
 X field of MMIX instruction, 5.
X_BIT (floating inexact bit), 18, 89.
XOR (bitwise exclusive-or), 10.
 XYZ field of MMIX instruction, 6.

Y field of MMIX instruction, 5.
 Y operand register, 18.
 Yoder, Michael Franz, 95.
 Yossarian, John, 3.
 Zettabyte, 94.
 YZ field of MMIX instruction, 5–6.
 Z field of MMIX instruction, 5.
 as immediate constant, 14.
 Z operand register, 18.
Z_BIT (floating division by zero bit), 18.
 Zero or set instructions of MMIX, 10.
 Zettabyte, 94.
ZSEV (zero or set if even), 10.
ZSN (zero or set if negative), 10.
ZSNN (zero or set if nonnegative), 10.
ZSNP (zero or set if nonpositive), 10.
ZSNZ (zero or set if nonzero), 10.
ZSOD (zero or set if odd), 10.
ZSP (zero or set if positive), 10.
ZSZ (zero or set if zero), 10.

ASCII CHARACTERS

	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	
#2x		!	"	#	\$	%	&	,	()	*	+	,	-	.	/	#2x
#3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	#3x
#4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0	#4x
#5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-	#5x
#6x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	#6x
#7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	█	#7x
	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	

MMIX OPERATION CODES

	#0	#1	#2	#3	#4	#5	#6	#7	
#0x	TRAP 5v	FCMP v	FUN v	FEQL v	FADD 4v	FIX 4v	FSUB 4v	FIXU 4v	#0x
	FLOT [I] 4v		FLOTU [I] 4v		SFLOT [I] 4v		SFLOTU [I] 4v		
#1x	FMUL 4v	FCMPF 4v	FUNE v	FEQLE 4v	FDIV 40v	FSQRT 40v	FREM 4v	FINT 4v	#1x
	MUL [I] 10v		MULU [I] 10v		DIV [I] 60v		DIVU [I] 60v		
#2x	ADD [I] v		ADDU [I] v		SUB [I] v		SUBU [I] v		#2x
	2ADDU [I] v		4ADDU [I] v		8ADDU [I] v		16ADDU [I] v		
#3x	CMP [I] v		CMPU [I] v		NEG [I] v		NEGU [I] v		#3x
	SL [I] v		SLU [I] v		SR [I] v		SRU [I] v		
#4x	BN [B] v+π		BZ [B] v+π		BP [B] v+π		BOD [B] v+π		#4x
	BNN [B] v+π		BNZ [B] v+π		BNP [B] v+π		BEV [B] v+π		
#5x	PBN [B] 3v-π		PBZ [B] 3v-π		PBP [B] 3v-π		PBOD [B] 3v-π		#5x
	PBNN [B] 3v-π		PBNZ [B] 3v-π		PBNP [B] 3v-π		PBEV [B] 3v-π		
#6x	CSN [I] v		CSZ [I] v		CSP [I] v		CSOD [I] v		#6x
	CSNN [I] v		CSNZ [I] v		CSNP [I] v		CSEV [I] v		
#7x	ZSN [I] v		ZSZ [I] v		ZSP [I] v		ZSOD [I] v		#7x
	ZSNN [I] v		ZSNZ [I] v		ZSNP [I] v		ZSEV [I] v		
#8x	LDB [I] μ+v		LDBU [I] μ+v		LDW [I] μ+v		LDWU [I] μ+v		#8x
	LDT [I] μ+v		LDTU [I] μ+v		LDO [I] μ+v		LDOU [I] μ+v		
#9x	LDSF [I] μ+v		LDHT [I] μ+v		CSWAP [I] 2μ+2v		LDUNC [I] μ+v		#9x
	LDVTS [I] v		PRELD [I] v		PREGO [I] v		GO [I] 3v		
#Ax	STB [I] μ+v		STBU [I] μ+v		STW [I] μ+v		STWU [I] μ+v		#Ax
	STT [I] μ+v		STTU [I] μ+v		STO [I] μ+v		STOU [I] μ+v		
#Bx	STSF [I] μ+v		STHT [I] μ+v		STCO [I] μ+v		STUNC [I] μ+v		#Bx
	SYNC [I] v		PREST [I] v		SYNCID [I] v		PUSHGO [I] 3v		
#Cx	OR [I] v		ORN [I] v		NOR [I] v		XOR [I] v		#Cx
	AND [I] v		ANDN [I] v		NAND [I] v		NXOR [I] v		
#Dx	BDIF [I] v		WDIF [I] v		TDIF [I] v		ODIF [I] v		#Dx
	MUX [I] v		SADD [I] v		MOR [I] v		MXOR [I] v		
#Ex	SETH v	SETMH v	SETML v	SETL v	INCH v	INCMH v	INCML v	INCL v	#Ex
	ORH v	ORMH v	ORML v	ORL v	ANDNH v	ANDNMH v	ANDNML v	ANDNL v	
#Fx	JMP [B] v		PUSHJ [B] v		GETA [B] v		PUT [I] v		#Fx
	POP 3v	RESUME 5v	[UN]SAVE 20μ+v		SYNC v	SWYM v	GET v	TRIP 5v	
	#8	#9	#A	#B	#C	#D	#E	#F	

π = 2v if the branch is taken, π = 0 if the branch is not taken

Simon
Platt
1985

**THE ART OF
COMPUTER PROGRAMMING**

SECOND EDITION

DONALD E. KNUTH *Stanford University*

 **ADDISON-WESLEY PUBLISHING COMPANY**

Volume 2 / Seminumerical Algorithms

**THE ART OF
COMPUTER PROGRAMMING**

SECOND EDITION

Reading, Massachusetts
Menlo Park, California · London · Amsterdam · Don Mills, Ontario · Sydney

This book is in the

**ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING**

MICHAEL A. HARRISON, Consulting Editor

Library of Congress Cataloging in Publication Data

Knuth, Donald Ervin (1938-
The Art of Computer Programming. 2d ed.

(Addison-Wesley series in computer science and information processing)
Includes index.
Contents: v. 1. Fundamental algorithms.—v. 2. Seminumerical algorithms.
1. Electronic digital computers—Programming. I. Title.
QA76.6.K64 001.6'42 73-1830
ISBN 0-201-03822-6 (v. 2)

COPYRIGHT © 1981, 1969 BY ADDISON-WESLEY PUBLISHING COMPANY, INC.
PHILIPPINES COPYRIGHT 1981 BY ADDISON-WESLEY PUBLISHING COMPANY,
INC. ALL RIGHTS RESERVED. NO PART OF THIS PUBLICATION MAY BE REPRO-
DUCED, STORED IN A RETRIEVAL SYSTEM, OR TRANSMITTED, IN ANY FORM OR
BY ANY MEANS, ELECTRONIC, MECHANICAL, PHOTOCOPYING, RECORDING, OR
OTHERWISE, WITHOUT THE PRIOR WRITTEN PERMISSION OF THE PUBLISHER.
PRINTED IN THE UNITED STATES OF AMERICA. PUBLISHED SIMULTANEOUSLY
IN CANADA.

The quotation on page 60 is reprinted by permission of Grove Press, Inc.

ISBN 0-201-03822-6
BCDEFGHIJ-MA-898765432

PREFACE

*O dear Ophelia!
I am ill at these numbers:
I have not art to reckon my groans.*

—Hamlet (Act II, Sc. 2, Line 120)

THE ALGORITHMS discussed in this book deal directly with numbers; yet I believe they are properly called *seminumerical*, because they lie on the borderline between numeric and symbolic calculation. Each algorithm not only computes the desired answers to a problem, it also is intended to blend well with the internal operations of a digital computer. In many cases a person will not be able to appreciate the beauty of such an algorithm unless he or she also has some knowledge of a computer's machine language; the efficiency of the corresponding machine program is a vital factor that cannot be divorced from the algorithm itself. The problem is to find the best ways to make computers deal with numbers, and this involves tactical as well as numerical considerations. Therefore the subject matter of this book is unmistakably a part of computer science, as well as of numerical mathematics.

Some people working in "higher levels" of numerical analysis will regard the topics treated here as the domain of system programmers. Other people working in "higher levels" of system programming will regard the topics treated here as the domain of numerical analysts. But I hope that there are a few people left who will want to look carefully at these basic methods; although the methods reside perhaps on a low level, they underlie all of the more grandiose applications of computers to numerical problems, so it is important to know them well. We are concerned here with the interface between numerical mathematics and computer programming, and it is the mating of both types of skills that makes the subject so interesting.

There is a noticeably higher percentage of mathematical material in this book than in other volumes of this series, because of the nature of the subjects treated. In most cases the necessary mathematical topics are developed here starting almost from scratch (or from results proved in Volume 1), but in some easily recognizable sections a knowledge of calculus has been assumed.

This volume comprises Chapters 3 and 4 of the complete series. Chapter 3 is concerned with "random numbers": it is not only a study of various methods for generating random sequences, it also investigates statistical tests for randomness, as well as the transformation of uniform random numbers into other types of random quantities; the latter subject illustrates how random numbers are used in practice. I have also included a section about the nature of randomness itself. Chapter 4 is my attempt to tell the fascinating story of what mankind has been able to learn about the processes of arithmetic, after centuries of progress. It discusses various systems for representing numbers, and how to convert between them; and it treats arithmetic on floating point numbers, high-precision integers, rational fractions, polynomials, and power series, including the questions of factoring and finding greatest common divisors.

Each of Chapters 3 and 4 can be used as the basis of a one-semester college course at the junior to graduate level. Although courses on "Random Numbers" and on "Arithmetic" are not presently a part of many college curricula, I believe the reader will find that the subject matter of these chapters lends itself nicely to a unified treatment of material that has real educational value. My own experience has been that these courses are a good means of introducing elementary probability theory and number theory to college students; nearly all of the topics usually treated in such introductory courses arise naturally in connection with applications, and the presence of these applications can be an important motivation that helps the student to learn and to appreciate the theory. Furthermore, each chapter gives a few hints of more advanced topics that will whet the appetite of many students for further mathematical study.

For the most part this book is self-contained, except for occasional discussions relating to the MIX computer explained in Volume 1. Appendix B contains a summary of the mathematical notations used, some of which are a little different from those found in traditional mathematics books.

In addition to the acknowledgments made in the preface to Volume 1, I would like to express deep appreciation to Elwyn R. Berlekamp, John Brillhart, George E. Collins, Stephen A. Cook, D. H. Lehmer, M. Donald MacLaren, Mervin E. Muller, Kenneth B. Stolarsky, and H. Zassenhaus, who have generously devoted considerable time to reading portions of the preliminary manuscript, and who have suggested many valuable improvements.

Princeton, New Jersey
October 1968

D. E. K.

Preface to the Second Edition

My first plan, when beginning to prepare this new edition, was to make it like the second edition of Volume 1: I went through the entire book and tried to improve every page without greatly perturbing the page numbering. But the number of improvements turned out to be so great that the entire book needed to be typeset again. As a result, I decided to make this book the first test case

for a new computer typesetting system I have been developing. I hope that most readers will like the slight changes in format, since my aim has been to produce a book whose typography is of the highest possible quality—superior even to the fine appearance of the previous editions, in spite of the fact that a computer is now involved. If all goes well, the third edition of Volume 1 and the second edition of Volume 3, and all editions of Volumes 4 through 7, will be published in the present style.

The decision to reset this entire book has freed me from the shackles of the previous page numbering, so I have been able to make major refinements and to insert a lot of new material. I estimate that about 45 percent of the book has changed. I did try, however, to keep the exercise numbers from being substantially altered; although many of the old exercises have been replaced by new and better ones, the new exercises tend to relate to the same idea as before. The explosive growth of seminumerical research in recent years has of course made it impossible for me to insert all of the beautiful ideas in this field that have been discovered since 1968; but I think that this edition does contain an up-to-date survey of all the major paradigms and basic theory of the subject, and it seems reasonable to believe that very few of the topics discussed here will ever become obsolete.

The National Science Foundation and the Office of Naval Research have been particularly generous in their support of my research as I work on these books. I am also deeply grateful for the advice and unselfish assistance of many readers, too numerous to mention. In this regard I want to acknowledge especially the help of several people whose contributions have been really major: B. I. Aspvall, R. P. Brent, U. Dieter, M. J. Fischer, R. W. Gosper, D. C. Hoaglin, W. M. Kahan, F. M. Liang, J. F. Reiser, A. G. Waterman, S. Winograd, and M. C. Wunderlich. Furthermore Marion Howe and other people in the Addison-Wesley production department have been enormously helpful in untangling literally thousands of hand-written inserts so that a very chaotic manuscript has come out looking reasonably well-organized. I suppose some mistakes still remain, or have crept in, and I would like to fix them; therefore I will cheerfully pay \$2.00 reward to the first finder of each technical, typographical, or historical error.

Stanford, California

July 1980

D. E. K.

*'Defendit numerus,' [there is safety in numbers]
is the maxim of the foolish;
'Deperdit numerus,' [there is ruin in numbers]
of the wise.*

—C. C. COLTON (1820)



NOTES ON THE EXERCISES

THE EXERCISES in this set of books have been designed for self-study as well as classroom study. It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems and thereby being encouraged to think about what has been read. Furthermore, we all learn best the things that we have discovered for ourselves. Therefore the exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible and to select problems that are enjoyable to solve.

In many books, easy exercises are found mixed randomly among extremely difficult ones. This is sometimes unfortunate because readers like to know in advance how long a problem ought to take—otherwise they may just skip over all the problems. A classic example of such a situation is the book *Dynamic Programming* by Richard Bellman; this is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading “Exercises and Research Problems,” with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied, “If you can solve it, it is an exercise; otherwise it’s a research problem.”

Good arguments can be made for including both research problems and very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance:

Rating Interpretation

- 00 An extremely easy exercise that can be answered immediately if the material of the text has been understood; such an exercise can almost always be worked “in your head.”
- 10 A simple problem that makes you think over the material just read, but it is by no means difficult. It should be possible to do this in one minute at most; pencil and paper may be useful in obtaining the solution.
- 20 An average problem that tests basic understanding of the text material, but you may need about fifteen or twenty minutes to answer it completely.

- 30 A problem of moderate difficulty and/or complexity; this one may involve over two hours' work to solve satisfactorily.
- 40 Quite a difficult or lengthy problem that would be suitable for a term project in classroom situations. It is expected that a student will be able to solve the problem in a reasonable amount of time, but the solution is not trivial.
- 50 A research problem that has not yet been solved satisfactorily, as far as the author knew at the time of writing, although many people have tried. If you have found an answer to such a problem, you ought to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided that it is correct).

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. For example, a rating of 17 would indicate an exercise that is a bit simpler than average. Problems with a rating of 50 that are subsequently solved by some reader may appear with a 45 rating in later editions of the book.

The author has earnestly tried to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else to find a solution; and everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess as to the level of difficulty, but they should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training and sophistication; as a result, some of the exercises are intended only for the use of more mathematically inclined readers. The rating is preceded by an *M* if the exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested only in programming the algorithms themselves. An exercise is marked with the letters "*HM*" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "*HM*" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, "►"; this designates problems that are especially instructive and that are especially recommended. Of course, no reader/student is expected to work *all* of the exercises, so those that seem to be the most valuable have been singled out. (This is not meant to detract from the other exercises!) Each reader should at least make an attempt to solve all of the problems whose rating is 10 or less; and the arrows may help to indicate which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to solve the problem by yourself, or unless you do not have time to work this particular problem. After getting your own solution or giving the problem a decent try, you may find the answer instructive and helpful. The solution given will often be quite short, and it will sketch the details under the assumption that you have earnestly tried to solve it by your own means first. Sometimes the solution gives less information than was asked; often it gives more. It is quite

possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details. Later editions of this book will give the improved solutions together with the solver's name where appropriate.

When working an exercise you may generally use the answers to previous exercises, unless specifically forbidden from doing so. The rating numbers have been assigned with this in mind; thus it is possible for exercise $n + 1$ to have a lower rating than exercise n , even though it includes the result of exercise n as a special case.

Summary of codes:

- Recommended
- M Mathematically oriented
- HM Requiring "higher math"

00	Immediate
10	Simple (one minute)
20	Medium (quarter hour)
30	Moderately hard
40	Term project
50	Research problem

EXERCISES

- 1. [00] What does the rating "M20" mean?
- 2. [10] Of what value can the exercises in a textbook be to the reader?
- 3. [M50] Prove that when n is an integer, $n > 2$, the equation $x^n + y^n = z^n$ has no solution in positive integers x, y, z .

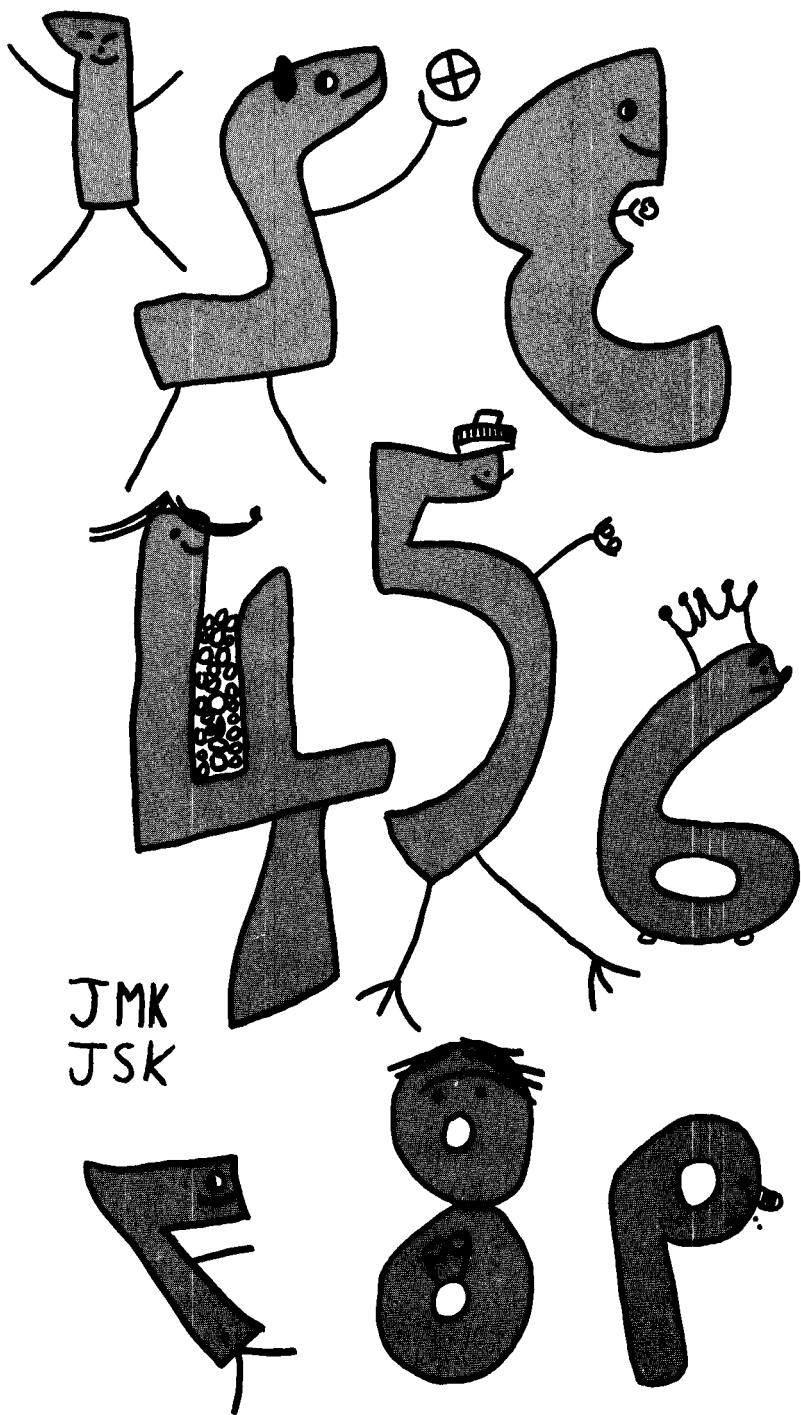
Exercise is the beste instrument in learnynge.

—ROBERT RECORDE (*The Whetstone of Witte*, 1557)

CONTENTS

Chapter 3—Random Numbers	1
3.1. Introduction	1
3.2. Generating Uniform Random Numbers	9
3.2.1. The Linear Congruential Method	9
3.2.1.1. Choice of modulus	11
3.2.1.2. Choice of multiplier	15
3.2.1.3. Potency	22
3.2.2. Other Methods	25
3.3. Statistical Tests	38
3.3.1. General Test Procedures for Studying Random Data	38
3.3.2. Empirical Tests	59
*3.3.3. Theoretical Tests	75
3.3.4. The Spectral Test	89
3.4. Other Types of Random Quantities	114
3.4.1. Numerical Distributions	114
3.4.2. Random Sampling and Shuffling	136
*3.5. What is a Random Sequence?	142
3.6. Summary	170
Chapter 4—Arithmetic	178
4.1. Positional Number Systems	179
4.2. Floating-Point Arithmetic	198
4.2.1. Single-Precision Calculations	198
4.2.2. Accuracy of Floating-Point Arithmetic	213
*4.2.3. Double-Precision Calculations	230
4.2.4. Distribution of Floating-Point Numbers	238
4.3. Multiple-Precision Arithmetic	250
4.3.1. The Classical Algorithms	250
*4.3.2. Modular Arithmetic	268
*4.3.3. How Fast Can We Multiply?	278
4.4. Radix Conversion	300
4.5. Rational Arithmetic	313
4.5.1. Fractions	313
4.5.2. The Greatest Common Divisor	316
*4.5.3. Analysis of Euclid's Algorithm	339
4.5.4. Factoring into Primes	364

4.6. Polynomial Arithmetic	399
4.6.1. Division of Polynomials	401
*4.6.2. Factorization of Polynomials	420
4.6.3. Evaluation of Powers	441
4.6.4. Evaluation of Polynomials	466
*4.7. Manipulation of Power Series	506
Answers to Exercises	516
Appendix A—Tables of Numerical Quantities	659
1. Fundamental Constants (decimal)	659
2. Fundamental Constants (octal)	660
3. Harmonic Numbers, Bernoulli Numbers, Fibonacci Numbers	661
Appendix B—Index to Notations	663
Index and Glossary	668



CHAPTER THREE

RANDOM NUMBERS

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

—JOHN VON NEUMANN (1951)

*Lest men suspect your tale untrue,
Keep probability in view.*

—JOHN GAY (1727)

*There wanted not some beams of light
to guide men in the exercise of their Stocastick faculty.*

—JOHN OWEN (1662)

3.1. INTRODUCTION

NUMBERS that are “chosen at random” are useful in many different kinds of applications. For example:

- a) *Simulation.* When a computer is being used to simulate natural phenomena, random numbers are required to make things realistic. Simulation covers many fields, from the study of nuclear physics (where particles are subject to random collisions) to operations research (where people come into, say, an airport at random intervals).
- b) *Sampling.* It is often impractical to examine all possible cases, but a random sample will provide insight into what constitutes “typical” behavior.
- c) *Numerical analysis.* Ingenious techniques for solving complicated numerical problems have been devised using random numbers. Several books have been written on this subject.
- d) *Computer programming.* Random values make a good source of data for testing the effectiveness of computer algorithms. This is the primary application of interest to us in this series of books; it accounts for the fact that random numbers are already being considered here in Chapter 3, before most of the other computer algorithms have appeared.

Several people experimented with the middle-square method in the early 1950s. Working with numbers that have four digits instead of ten, G. E. Forsythe tried 16 different starting values and found that 12 of them led to sequences ending with the cycle 6100, 2100, 4100, 8100, 6100, ..., while two of them degenerated to zero. N. Metropolis also conducted extensive tests on the middle-square method, mostly in the binary number system. He showed that when 20-bit numbers are being used, there are 13 different cycles into which the sequence might degenerate, the longest of which has a period of length 142.

It is fairly easy to restart the middle-square method on a new value when zero has been detected, but long cycles are somewhat harder to avoid. Exercise 7 discusses some interesting ways to determine the cycles of periodic sequences, using very little memory space.

A theoretical disadvantage of the middle-square method is given in exercises 9 and 10. On the other hand, working with 38-bit numbers, Metropolis obtained a sequence of about 750,000 numbers before degeneracy occurred, and the resulting $750,000 \times 38$ bits satisfactorily passed statistical tests for randomness. This shows that the middle-square method can give usable results, but it is rather dangerous to put much faith in it until after elaborate computations have been performed.

Many random number generators in use today are not very good. There is a tendency for people to avoid learning anything about such subroutines; quite often we find that some old method that is comparatively unsatisfactory has blindly been passed down from one programmer to another, and today's users have no understanding of its limitations. We shall see in this chapter that it is not difficult to learn the most important facts about random number generators and their proper use.

It is not easy to invent a foolproof source of random numbers. This fact was convincingly impressed upon the author several years ago, when he attempted to create a fantastically good generator using the following peculiar approach:

Algorithm K ("Super-random" number generator). Given a 10-digit decimal number X , this algorithm may be used to change X to the number that should come next in a supposedly random sequence. Although the algorithm might be expected to yield quite a random sequence, reasons given below show that it is, in fact, not very good at all. (The reader need not study this algorithm in great detail except to observe how complicated it is; note, in particular, steps K1 and K2.)

- K1. [Choose number of iterations.] Set $Y \leftarrow \lfloor X/10^9 \rfloor$, the most significant digit of X . (We will execute steps K2 through K13 exactly $Y + 1$ times; that is, we will apply randomizing transformations a random number of times.)
- K2. [Choose random step.] Set $Z \leftarrow \lfloor X/10^8 \rfloor \bmod 10$, the second most significant digit of X . Go to step K($3 + Z$). (That is, we now jump to a random step in the program.)
- K3. [Ensure $\geq 5 \times 10^9$.] If $X < 5000000000$, set $X \leftarrow X + 5000000000$.

device. A famous random-number machine called ERNIE has been used to pick the winning numbers in the British Premium Savings Bonds lottery. [See the articles by Kendall and Babington-Smith in *J. Royal Stat. Soc.*, Series A, **101** (1938), 147–166, and Series B, **6** (1939), 51–61; see also the review of the RAND table in *Math. Comp.* **10** (1956), 39–43, and the discussion of ERNIE by W. E. Thomson et al., *J. Royal Stat. Soc.*, Series A, **122** (1959), 301–333.]

Shortly after computers were introduced, people began to search for efficient ways to obtain random numbers within computer programs. A table can be used, but this method is of limited utility because of the memory space and input time requirement, because the table may be too short, and because it is a bit of a nuisance to prepare and maintain the table. Machines such as ERNIE might be attached to the computer, but this would be unsatisfactory since it would be impractical to reproduce calculations exactly a second time when checking out a program; and such machines have tended to suffer from malfunctions that are difficult to detect.

The inadequacy of these methods led to an interest in the production of random numbers using the arithmetic operations of a computer. John von Neumann first suggested this approach in about 1946, using the “middle-square” method. His idea was to take the square of the previous random number and to extract the middle digits; for example, if we are generating 10-digit numbers and the previous value was 5772156649, we square it to get

$$33317792380594909201,$$

and the next number is therefore 7923805949.

There is a fairly obvious objection to this technique: how can a sequence generated in such a way be random, since each number is completely determined by its predecessor? The answer is that this sequence *isn't* random, but it *appears* to be. In typical applications the actual relationship between one number and its successor has no physical significance; hence the nonrandom character is not really undesirable. Intuitively, the middle square seems to be a fairly good scrambling of the previous number.

Sequences generated in a deterministic way such as this are usually called *pseudo-random* or *quasi-random* sequences in the highbrow technical literature, but in this book we shall simply call them random sequences, with the understanding that they only *appear* to be random. Being “apparently random” is perhaps all that can be said about any random sequence anyway. Random numbers generated deterministically on computers have worked quite well in nearly every application, provided that a suitable method has been carefully selected. Of course, deterministic sequences aren't always the answer; they certainly shouldn't replace ERNIE for the lotteries.

Von Neumann's original “middle-square method” has actually proved to be a comparatively poor source of random numbers. The danger is that the sequence tends to get into a rut, a short cycle of repeating elements. For example, if zero ever appears as a number of the sequence, it will continually perpetuate itself.

e) *Decision making.* There are reports that many executives make their decisions by flipping a coin or by throwing darts, etc. It is also rumored that some college professors prepare their grades on such a basis. Sometimes it is important to make a completely "unbiased" decision; this ability is occasionally useful in computer algorithms, for example in situations where a fixed decision made each time would cause the algorithm to run more slowly. Randomness is also an essential part of optimal strategies in the theory of games.

f) *Recreation.* Rolling dice, shuffling decks of cards, spinning roulette wheels, etc., are fascinating pastimes for just about everybody. These traditional uses of random numbers have suggested the name "Monte Carlo method," a general term used to describe any algorithm that employs random numbers.

People who think about this topic almost invariably get into philosophical discussions about what the word "random" means. In a sense, there is no such thing as a random number; for example, is 2 a random number? Rather, we speak of a sequence of independent random numbers with a specified distribution, and this means loosely that each number was obtained merely by chance, having nothing to do with other numbers of the sequence, and that each number has a specified probability of falling in any given range of values.

A uniform distribution on a finite set of numbers is one in which each possible number is equally probable. A distribution is generally understood to be uniform unless some other distribution is specifically mentioned.

Each of the ten digits 0 through 9 will occur about $\frac{1}{10}$ of the time in a (uniform) sequence of random digits. Each pair of two successive digits should occur about $\frac{1}{100}$ of the time, etc. Yet if we take a truly random sequence of a million digits, it will not always have exactly 100,000 zeros, 100,000 ones, etc. In fact, chances of this are quite slim; a sequence of such sequences will have this character on the average.

Any specified sequence of a million digits is equally as probable as the sequence consisting of a million zeros. Thus, if we are choosing a million digits at random and if the first 999,999 of them happen to come out to be zero, the chance that the final digit is zero is still exactly $\frac{1}{10}$, in a truly random situation. These statements seem paradoxical to many people, but there is really no contradiction involved.

There are several ways to formulate decent abstract definitions of randomness, and we will return to this interesting subject in Section 3.5; but for the moment, let us content ourselves with an intuitive understanding of the concept.

At first, people who needed random numbers in their scientific work would draw balls out of a "well-stirred urn" or would roll dice or deal out cards. A table of over 40,000 random digits, "taken at random from census reports," was published in 1927 by L. H. C. Tippett. Since then, a number of devices have been built to generate random numbers mechanically; the first such machine was used in 1939 by M. G. Kendall and B. Babington-Smith to produce a table of 100,000 random digits, and in 1955 the RAND Corporation published a widely used table of a million random digits obtained with the help of another special

- K4.** [Middle square.] Replace X by $\lfloor X^2/10^5 \rfloor \bmod 10^{10}$, i.e., by the middle of the square of X .
- K5.** [Multiply.] Replace X by $(1001001001X) \bmod 10^{10}$.
- K6.** [Pseudo-complement.] If $X < 100000000$, then set $X \leftarrow X + 9814055677$; otherwise set $X \leftarrow 10^{10} - X$.
- K7.** [Interchange halves.] Interchange the low-order five digits of X with the high-order five digits, i.e., $X \leftarrow 10^5(X \bmod 10^5) + \lfloor X/10^5 \rfloor$, the middle 10 digits of $(10^{10} + 1)X$.
- K8.** [Multiply.] Same as step K5.
- K9.** [Decrease digits.] Decrease each nonzero digit of the decimal representation of X by one.
- K10.** [99999 modify.] If $X < 10^5$, set $X \leftarrow X^2 + 99999$; otherwise set $X \leftarrow X - 99999$.
- K11.** [Normalize.] (At this point X cannot be zero.) If $X < 10^9$, set $X \leftarrow 10X$ and repeat this step.
- K12.** [Modified middle square.] Replace X by $\lfloor X(X - 1)/10^5 \rfloor \bmod 10^{10}$, i.e., by the middle 10 digits of $X(X - 1)$.
- K13.** [Repeat?] If $Y > 0$, decrease Y by 1 and return to step K2. If $Y = 0$, the algorithm terminates with X as the desired “random” value. ■

(The machine-language program corresponding to the above algorithm was intended to be so complicated that a person reading a listing of it without explanatory comments wouldn’t know what the program was doing.)

Considering all the contortions of Algorithm K, doesn’t it seem plausible that it should produce almost an infinite supply of unbelievably random numbers? No! In fact, when this algorithm was first put onto a computer, it almost immediately converged to the 10-digit value 6065038420, which—by extraordinary coincidence—is transformed into itself by the algorithm (see Table 1). With another starting number, the sequence began to repeat after 7401 values, in a cyclic period of length 3178.

The moral of this story is that *random numbers should not be generated with a method chosen at random*. Some theory should be used.

In this chapter, we shall consider random number generators that are superior to the middle-square method and to Algorithm K; the corresponding sequences are guaranteed to have certain desirable random properties, and no degeneracy will occur. We shall explore the reasons for this random behavior in some detail, and we shall also consider techniques for manipulating random numbers. For example, one of our investigations will be the shuffling of a simulated deck of cards within a computer program.

Section 3.6 summarizes this chapter and lists several bibliographic sources.

Table 1

A COLOSSAL COINCIDENCE: THE NUMBER 6065038420
IS TRANSFORMED INTO ITSELF BY ALGORITHM K.

Step	X (after)	Step	X (after)
K1	6065038420	K9	1107855700
K3	6065038420	K10	1107755701
K4	6910360760	K11	1107755701
K5	8031120760	K12	1226919902 $Y = 3$
K6	1968879240	K5	0048821902
K7	7924019688	K6	9862877579
K8	9631707688	K7	7757998628
K9	8520606577	K8	2384626628
K10	8520506578	K9	1273515517
K11	8520506578	K10	1273415518
K12	0323372207 $Y = 6$	K11	1273415518
K6	9676627793	K12	5870802097 $Y = 2$
K7	2779396766	K11	5870802097
K8	4942162766	K12	3172562687 $Y = 1$
K9	3831051655	K4	1540029446
K10	3830951656	K5	7015475446
K11	3830951656	K6	2984524554
K12	1905867781 $Y = 5$	K7	2455429845
K12	3319967479 $Y = 4$	K8	2730274845
K6	6680032521	K9	1620163734
K7	3252166800	K10	1620063735
K8	2218966800	K11	1620063735
		K12	6065038420 $Y = 0$

EXERCISES

- 1. [20] Suppose that you wish to obtain a decimal digit at random, not using a computer. Which of the following methods would be suitable?
 - Open a telephone directory to a random place (i.e., stick your finger in it somewhere) and use the units digit of the first number found on the selected page.
 - Same as (a), but use the units digit of the page number.
 - Roll a die that is in the shape of a regular icosahedron, whose twenty faces have been labeled with the digits 0, 0, 1, 1, ..., 9, 9. Use the digit that appears on top, when the die comes to rest. (A felt table with a hard surface is recommended for rolling dice.)
 - Expose a geiger counter to a source of radioactivity for one minute (shielding yourself) and use the units digit of the resulting count. Assume that the geiger counter displays the number of counts in decimal notation, and that the count is initially zero.
 - Glance at your wristwatch; and if the position of the second-hand is between $6n$ and $6(n + 1)$ seconds, choose the digit n .
 - Ask a friend to think of a random digit, and use the digit he names.
 - Ask an enemy to think of a random digit, and use the digit he names.

- h) Assume that 10 horses are entered in a race and that you know nothing whatever about their qualifications. Assign to these horses the digits 0 to 9, in arbitrary fashion, and after the race use the winner's digit.
2. [M22] In a random sequence of a million decimal digits, what is the probability that there are exactly 100,000 of each possible digit?
3. [10] What number follows 1010101010 in the middle-square method?
4. [10] Why can't the value of X be zero when step K11 of Algorithm K is performed? What would be wrong with the algorithm if X could be zero?
5. [15] Explain why, in any case, Algorithm K should not be expected to provide "infinitely many" random numbers, in the sense that (even if the coincidence given in Table 1 had not occurred) one knows in advance that any sequence generated by Algorithm K will eventually be periodic.
- 6. [M21] Suppose that we want to generate a sequence of integers X_0, X_1, X_2, \dots , in the range $0 \leq X_n < m$. Let $f(x)$ be any function such that $0 \leq x < m$ implies $0 \leq f(x) < m$. Consider a sequence formed by the rule $X_{n+1} = f(X_n)$. (Examples are the middle-square method and Algorithm K.)
- Show that the sequence is ultimately periodic, in the sense that there exist numbers λ and μ for which the values $X_0, X_1, \dots, X_\mu, \dots, X_{\mu+\lambda-1}$ are distinct, but $X_{n+\lambda} = X_n$ when $n \geq \mu$. Find the maximum and minimum possible values of μ and λ .
 - (R. W. Floyd.) Show that there exists an $n > 0$ such that $X_n = X_{2n}$; and the smallest such value of n lies in the range $\mu \leq n \leq \mu + \lambda$. Furthermore the value of X_n is unique in the sense that if $X_n = X_{2n}$ and $X_r = X_{2r}$, then $X_r = X_n$.
 - Use the idea of part (b) to design an algorithm that calculates μ and λ for any given function f and any given X_0 , using only $O(\mu + \lambda)$ steps and only a bounded number of memory locations.
- 7. [M21] (R. P. Brent, 1977.) Let $\ell(n)$ be the least power of 2 that is less than or equal to n ; thus, for example, $\ell(15) = 8$ and $\ell(\ell(n)) = \ell(n)$.
- Show that, in terms of the notation in exercise 6, there exists an $n > 0$ such that $X_n = X_{\ell(n)-1}$. Find a formula that expresses the least such n in terms of μ and λ .
 - Apply this result to design an algorithm that can be used in conjunction with any random number generator of the type $X_{n+1} = f(X_n)$, to prevent it from cycling indefinitely. Your algorithm should calculate the period length λ , and it should use only a small amount of memory space—you must not simply store all of the computed sequence values!
 - [28] Make a complete examination of the middle-square method in the case of two-digit decimal numbers. (a) We might start the process out with any of the 100 possible values 00, 01, ..., 99. How many of these values lead ultimately to the repeating cycle 00, 00, ...? [Example: Starting with 43, we obtain the sequence 43, 84, 05, 02, 00, 00, 00, ...] (b) How many possible final cycles are there? How long is the longest cycle? (c) What starting value or values will give the largest number of distinct elements before the sequence repeats?
 - [M14] Prove that the middle-square method using $2n$ -digit numbers to the base b has the following disadvantage: If the sequence includes any number whose most significant n digits are zero, the succeeding numbers will get smaller and smaller until zero occurs repeatedly.

10. [M16] Under the assumptions of the preceding exercise, what can you say about the sequence of numbers following X if the least significant n digits of X are zero? What if the least significant $n + 1$ digits are zero?

► **11.** [M26] Consider sequences of random number generators having the form described in exercise 6. If we choose $f(x)$ and X_0 at random, i.e., if we assume that each of the m^m possible functions $f(x)$ is equally probable and that each of the m possible values of X_0 is equally probable, what is the probability that the sequence will eventually degenerate into a cycle of length $\lambda = 1$? (Note: The assumptions of this problem give a natural way to think of a “random” random number generator of this type. A method such as Algorithm K may be expected to behave somewhat like the generator considered here; the answer to this problem gives a measure of how “colossal” the coincidence of Table 1 really is.)

► **12.** [M31] Under the assumptions of the preceding exercise, what is the average length of the final cycle? What is the average length of the sequence before it begins to cycle? (In the notation of exercise 6, we wish to examine the average values of λ and of $\mu + \lambda$.)

13. [M42] If $f(x)$ is chosen at random in the sense of exercise 11, what is the average length of the longest cycle obtainable by varying the starting value X_0 ? (Note: We have already considered the analogous problem in the case that $f(x)$ is a random permutation; see exercise 1.3.3–23.)

14. [M38] If $f(x)$ is chosen at random in the sense of exercise 11, what is the average number of distinct final cycles obtainable by varying the starting value? [Cf. exercise 8(b).]

15. [M15] If $f(x)$ is chosen at random in the sense of exercise 11, what is the probability that none of the final cycles has length 1, regardless of the choice of X_0 ?

16. [15] A sequence generated as in exercise 6 must begin to repeat after at most m values have been generated. Suppose we generalize the method so that X_{n+1} depends on X_{n-1} as well as on X_n ; formally, let $f(x, y)$ be a function such that $0 \leq x, y < m$ implies $0 \leq f(x, y) < m$. The sequence is constructed by selecting X_0 and X_1 arbitrarily, and then letting

$$X_{n+1} = f(X_n, X_{n-1}), \quad \text{for } n > 0.$$

What is the maximum period conceivably attainable in this case?

17. [10] Generalize the situation in the previous exercise so that X_{n+1} depends on the preceding k values of the sequence.

18. [M20] Invent a method analogous to that of exercise 7 for finding cycles in the general form of random number generator discussed in exercise 17.

19. [M48] Solve the problems of exercises 11 through 15 for the more general case that X_{n+1} depends on the preceding k values of the sequence; each of the m^{mk} functions $f(x_1, \dots, x_k)$ is to be considered equally probable. (Note: The number of functions that yield the maximum period is analyzed in exercise 2.3.4.2–23.)

3.2. GENERATING UNIFORM RANDOM NUMBERS

IN THIS SECTION we shall consider methods for generating a sequence of random fractions, i.e., random *real numbers* U_n , *uniformly distributed between zero and one*. Since a computer can represent a real number with only finite accuracy, we shall actually be generating integers X_n between zero and some number m ; the fraction

$$U_n = X_n/m$$

will then lie between zero and one. Usually m is the word size of the computer, so X_n may be regarded (conservatively) as the integer contents of a computer word with the radix point assumed at the extreme right, and U_n may be regarded (liberally) as the contents of the same word with the radix point assumed at the extreme left.

3.2.1. The Linear Congruential Method

By far the most popular random number generators in use today are special cases of the following scheme, introduced by D. H. Lehmer in 1949. [See *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery* (Cambridge: Harvard University Press, 1951), 141–146.] We choose four “magic numbers”:

$$\begin{aligned} m, & \text{ the modulus; } & m > 0. \\ a, & \text{ the multiplier; } & 0 \leq a < m. \\ c, & \text{ the increment; } & 0 \leq c < m. \\ X_0, & \text{ the starting value; } & 0 \leq X_0 < m. \end{aligned} \tag{1}$$

The desired sequence of random numbers $\langle X_n \rangle$ is then obtained by setting

$$X_{n+1} = (aX_n + c) \bmod m, \quad n \geq 0. \tag{2}$$

This is called a *linear congruential sequence*. Taking the remainder mod m is somewhat like determining where a ball will land in a spinning roulette wheel.

For example, the sequence obtained when $m = 10$ and $X_0 = a = c = 7$ is

$$7, 6, 9, 0, 7, 6, 9, 0, \dots \tag{3}$$

As this example shows, the sequence is not always “random” for all choices of m , a , c , and X_0 ; the principles of choosing the magic numbers appropriately will be investigated carefully in later parts of this chapter.

Example (3) illustrates the fact that the congruential sequences always “get into a loop”; i.e., there is ultimately a cycle of numbers that is repeated endlessly. This property is common to all sequences having the general form $X_{n+1} = f(X_n)$; see exercise 3.1–6. The repeating cycle is called the *period*; sequence (3) has a period of length 4. A useful sequence will of course have a relatively long period.

The special case $c = 0$ deserves explicit mention, since the number generation process is a little faster when $c = 0$ than it is when $c \neq 0$. We shall see later that the restriction $c = 0$ cuts down the length of the period of the sequence, but it is still possible to make the period reasonably long. Lehmer's original generation method had $c = 0$, although he mentioned $c \neq 0$ as a possibility; the idea of taking $c \neq 0$ to obtain longer periods is due to Thomson [Comp. J. 1 (1958), 83, 86] and, independently, to Rotenberg [JACM 7 (1960), 75–77]. The terms *multiplicative congruential method* and *mixed congruential method* are used by many authors to denote linear congruential sequences with $c = 0$ and $c \neq 0$, respectively.

The letters m , a , c , and X_0 will be used throughout this chapter in the sense described above. Furthermore, we will find it useful to define

$$b = a - 1, \quad (4)$$

in order to simplify many of our formulas.

We can immediately reject the case $a = 1$, for this would mean that $X_n = (X_0 + nc) \bmod m$, and the sequence would certainly not behave as a random sequence. The case $a = 0$ is even worse. Hence for practical purposes we may assume that

$$a \geq 2, \quad b \geq 1. \quad (5)$$

Now we can prove a generalization of Eq. (2),

$$X_{n+k} = (a^k X_n + (a^k - 1)c/b) \bmod m, \quad k \geq 0, \quad n \geq 0, \quad (6)$$

which expresses the $(n+k)$ th term directly in terms of the n th term. (The special case $n = 0$ in this equation is worthy of note.) It follows that the subsequence consisting of every k th term of $\langle X_n \rangle$ is another linear congruential sequence, having the multiplier $a^k \bmod m$ and the increment $((a^k - 1)c/b) \bmod m$.

An important corollary of (6) is that the general sequence defined by m , a , c , and X_0 can be expressed very simply in terms of the special case where $c = 1$ and $X_0 = 0$. Let

$$Y_0 = 0, \quad Y_{n+1} = (aY_n + 1) \bmod m. \quad (7)$$

According to Eq. (6) we will have $Y_k \equiv (a^k - 1)/b$ (modulo m), hence the general sequence defined in (2) satisfies

$$X_n = (AY_n + X_0) \bmod m, \quad \text{where } A = (X_0 b + c) \bmod m. \quad (8)$$

EXERCISES

- [10] Example (3) shows a situation in which $X_4 = X_0$, so the sequence begins again from the beginning. Give an example of a linear congruential sequence with $m = 10$ for which X_0 never appears again in the sequence.

► 2. [M20] Show that if a and m are relatively prime, the number X_0 will always appear in the period.

3. [M10] If a and m are not relatively prime, explain why the sequence will be somewhat handicapped and probably not very random; hence we will generally want the multiplier a to be relatively prime to the modulus m .

4. [11] Prove Eq. (6).

5. [M20] Equation (6) holds for $k \geq 0$. If possible, give a formula that expresses X_{n+k} in terms of X_n for negative values of k .

3.2.1.1. Choice of modulus. Our current goal is to find good values for the parameters that define a linear congruential sequence. Let us first consider the proper choice of the number m . We want m to be rather large, since the period cannot have more than m elements. (Even if a person wants to generate only random zeros and ones, he should *not* take $m = 2$, for then the sequence would at best have the form $\dots, 0, 1, 0, 1, 0, 1, \dots$! Methods for modifying random numbers to get random zeros and ones are discussed in Section 3.4.)

Another factor that influences our choice of m is speed of generation: We want to pick a value so that the computation of $(aX_n + c) \bmod m$ is fast.

Consider MIX as an example. We can compute $y \bmod m$ by putting y in registers A and X and dividing by m ; assuming that y and m are positive, we see that $y \bmod m$ will then appear in register X. But division is a comparatively slow operation, and it can be avoided if we take m to be a value that is especially convenient, such as the *word size* of our computer.

Let w be the computer's word size, namely, 2^e on an e -bit binary computer or 10^e on an e -digit decimal machine. (In this book we shall often use the letter e to denote an arbitrary integer exponent, instead of the base of natural logarithms, hoping that the context will make our notation unambiguous. Physicists have a similar problem when they use e for the charge on an electron.) The result of an addition operation is usually given modulo w , except on ones'-complement machines; and multiplication mod w is also quite simple, since the desired result is the lower half of the product. Thus, the following program computes the quantity $(aX + c) \bmod w$ efficiently:

LDA	A	rA $\leftarrow a$.	
MUL	X	rAX $\leftarrow (rA) \cdot X$.	(1)
SLAX	5	rA $\leftarrow rAX \bmod w$.	
ADD	C	rA $\leftarrow (rA + c) \bmod w$. ■	

The result appears in register A. The overflow toggle might be on at the conclusion of the above sequence of instructions, and if this is undesirable, the code should be followed by, e.g., “`JOV *+1`” to turn it off.

A clever technique that is less commonly known can be used to perform computations modulo $(w+1)$. For reasons to be explained later, we will generally want $c = 0$ when $m = w + 1$, so we merely need to compute $(aX) \bmod (w + 1)$.

The following program does this:

01	LDAN X	$rA \leftarrow -X.$
02	MUL A	$rAX \leftarrow (rA) \cdot a.$
03	STX TEMP	
04	SUB TEMP	$rA \leftarrow rA - rX.$
05	JANN *+3	Exit if $rA \geq 0.$
06	INCA 2	$rA \leftarrow rA + 2.$
07	ADD =w - 1=	$rA \leftarrow rA + w - 1.$ (Cf. exercise 3.) ■

(2)

Register A now contains the value $(aX) \bmod (w + 1)$. Of course, this value might lie anywhere between 0 and w , inclusive, so the reader may legitimately wonder how we can represent so many values in the A-register! (The register obviously cannot hold a number larger than $w - 1$.) The answer is that overflow will be on after the above program if and only if the result equals w , assuming that overflow was initially off. It is convenient simply to reject the value w if it appears in the congruential sequence modulo $w + 1$; this will happen if lines 05 and 06 of (2) are replaced by “JANN *+4; INCA 2; JAP *-5”.

To prove that code (2) actually does determine $(aX) \bmod (w + 1)$, note that in line 04 we are subtracting the lower half of the product from the upper half. No overflow can occur at this step; and if $aX = qw + r$, with $0 \leq r < w$, we will have the quantity $r - q$ in register A after line 04. Now

$$aX = q(w + 1) + (r - q),$$

and since $q < w$, we have $-w < r - q < w$; hence $(aX) \bmod (w + 1)$ equals either $r - q$ or $r - q + (w + 1)$, depending on whether $r - q \geq 0$ or $r - q < 0$.

A similar technique can be used to get the product of two numbers modulo $(w - 1)$; see exercise 8.

In later sections we shall require a knowledge of the prime factors of m in order to choose the multiplier a correctly. Table 1 lists the complete factorization of $w \pm 1$ into primes for nearly every known computer word size; the methods of Section 4.5.4 can be used to extend this table if desired.

The reader may well ask why we bother to consider using $m = w \pm 1$, when the choice $m = w$ is so manifestly convenient. The reason is that when $m = w$, the right-hand digits of X_n are much less random than the left-hand digits. If d is a divisor of m , and if

$$Y_n = X_n \bmod d, \quad (3)$$

we can easily show that

$$Y_{n+1} = (aY_n + c) \bmod d. \quad (4)$$

(For, $X_{n+1} = aX_n + c - qm$ for some integer q , and taking both sides mod d causes the quantity qm to drop out when d is a factor of m .)

To illustrate the significance of Eq. (4), let us suppose, for example, that we have a binary computer. If $m = w = 2^e$, the low-order four bits of X_n are

Table 1
PRIME FACTORIZATIONS OF $w \pm 1$

$2^e - 1$	e	$2^e + 1$
7 · 31 · 151	15	$3^2 \cdot 11 \cdot 331$
3 · 5 · 17 · 257	16	65537
131071	17	3 · 43691
$3^3 \cdot 7 \cdot 19 \cdot 73$	18	$5 \cdot 13 \cdot 37 \cdot 109$
524287	19	3 · 174763
$3 \cdot 5^2 \cdot 11 \cdot 31 \cdot 41$	20	17 · 61681
$7^2 \cdot 127 \cdot 337$	21	$3^2 \cdot 43 \cdot 5419$
$3 \cdot 23 \cdot 89 \cdot 683$	22	$5 \cdot 397 \cdot 2113$
47 · 178481	23	3 · 2796203
$3^2 \cdot 5 \cdot 7 \cdot 13 \cdot 17 \cdot 241$	24	97 · 257 · 673
31 · 601 · 1801	25	3 · 11 · 251 · 4051
3 · 2731 · 8191	26	$5 \cdot 53 \cdot 157 \cdot 1613$
7 · 73 · 262657	27	$3^4 \cdot 19 \cdot 87211$
$3 \cdot 5 \cdot 29 \cdot 43 \cdot 113 \cdot 127$	28	17 · 15790321
233 · 1103 · 2089	29	3 · 59 · 3033169
$3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$	30	$5^2 \cdot 13 \cdot 41 \cdot 61 \cdot 1321$
2147483647	31	3 · 715827883
$3 \cdot 5 \cdot 17 \cdot 257 \cdot 65537$	32	641 · 6700417
7 · 23 · 89 · 599479	33	$3^2 \cdot 67 \cdot 683 \cdot 20857$
3 · 43691 · 131071	34	$5 \cdot 137 \cdot 953 \cdot 26317$
31 · 71 · 127 · 122921	35	3 · 11 · 43 · 281 · 86171
$3^3 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 37 \cdot 73 \cdot 109$	36	17 · 241 · 433 · 38737
223 · 616318177	37	3 · 1777 · 25781083
3 · 174763 · 524287	38	$5 \cdot 229 \cdot 457 \cdot 525313$
7 · 79 · 8191 · 121369	39	$3^2 \cdot 2731 \cdot 22366891$
$3 \cdot 5^2 \cdot 11 \cdot 17 \cdot 31 \cdot 41 \cdot 61681$	40	257 · 4278255361
13367 · 164511353	41	3 · 83 · 8831418697
$3^2 \cdot 7^2 \cdot 43 \cdot 127 \cdot 337 \cdot 5419$	42	$5 \cdot 13 \cdot 29 \cdot 113 \cdot 1429 \cdot 14449$
431 · 9719 · 2099863	43	3 · 2932031007403
$3 \cdot 5 \cdot 23 \cdot 89 \cdot 397 \cdot 683 \cdot 2113$	44	17 · 353 · 2931542417
7 · 31 · 73 · 151 · 631 · 23311	45	$3^3 \cdot 11 \cdot 19 \cdot 331 \cdot 18837001$
3 · 47 · 178481 · 2796203	46	$5 \cdot 277 \cdot 1013 \cdot 1657 \cdot 30269$
2351 · 4513 · 13264529	47	3 · 283 · 165768537521
$3^2 \cdot 5 \cdot 7 \cdot 13 \cdot 17 \cdot 97 \cdot 241 \cdot 257 \cdot 673$	48	193 · 65537 · 22253377
179951 · 3203431780337	59	3 · 2833 · 37171 · 1824726041
$3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321$	60	17 · 241 · 61681 · 4562284561
$7^2 \cdot 73 \cdot 127 \cdot 337 \cdot 92737 \cdot 649657$	63	$3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77158673929$
$3 \cdot 5 \cdot 17 \cdot 257 \cdot 641 \cdot 65537 \cdot 6700417$	64	274177 · 67280421310721
$10^e - 1$	e	$10^e + 1$
$3^3 \cdot 7 \cdot 11 \cdot 13 \cdot 37$	6	101 · 9901
$3^2 \cdot 239 \cdot 4649$	7	11 · 909091
$3^2 \cdot 11 \cdot 73 \cdot 101 \cdot 137$	8	17 · 5882353
$3^4 \cdot 37 \cdot 333667$	9	7 · 11 · 13 · 19 · 52579
$3^2 \cdot 11 \cdot 41 \cdot 271 \cdot 9091$	10	101 · 3541 · 27961
$3^2 \cdot 21649 \cdot 513239$	11	$11^2 \cdot 23 \cdot 4093 \cdot 8779$
$3^3 \cdot 7 \cdot 11 \cdot 13 \cdot 37 \cdot 101 \cdot 9901$	12	73 · 137 · 99990001
$3^2 \cdot 11 \cdot 17 \cdot 73 \cdot 101 \cdot 137 \cdot 5882353$	16	353 · 449 · 641 · 1409 · 69857

the numbers $Y_n = X_n \bmod 2^4$. The gist of Eq. (4) is that the low-order four bits of $\langle X_n \rangle$ form a congruential sequence that has a period of length 16 or less. Similarly, the low-order five bits are periodic with a period of at most 32; and the least significant bit of X_n is either constant or strictly alternating.

This situation does not occur when $m = w \pm 1$; in such a case, the low-order bits of X_n will behave just as randomly as the high-order bits do. If, for example, $w = 2^{35}$ and $m = 2^{35} - 1$, the numbers of the sequence will not be very random if we consider only their remainders mod 31, 71, 127, or 122921 (cf. Table 1); but the low-order bit, which represents the numbers of the sequence taken mod 2, would be satisfactorily random.

Another alternative is to let m be the largest prime number less than w . This prime may be found by using the techniques of Section 4.5.4, and a table of suitably large primes appears in that section.

In most applications, the low-order bits are insignificant, and the choice $m = w$ is quite satisfactory—provided that the programmer using the random numbers does so wisely.

Our discussion so far has been based on a “signed magnitude” computer like MIX. Similar ideas apply to machines that use complement notations, although there are some instructive variations. For example, a DEC 20 computer has 36 bits with two’s complement arithmetic; when it computes the product of two nonnegative integers, the lower half contains the least significant 35 bits with a plus sign. On this machine we should therefore take $w = 2^{35}$, not 2^{36} . The 32-bit two’s complement arithmetic on IBM System/370 computers is different: the lower half of a product contains a full 32 bits. Some programmers have felt that this is a disadvantage, since the lower half can be negative when the operands are positive, and it is a nuisance to correct this; but actually it is a distinct advantage from the standpoint of random number generation, since we can take $m = 2^{32}$ instead of 2^{31} (see exercise 4).

EXERCISES

1. [M12] In exercise 3.2.1–3 we concluded that the best congruential generators will have the multiplier a relatively prime to m . Show that when $m = w$ in this case it is possible to compute $(aX + c) \bmod w$ in just three MIX instructions, rather than the four in (1), with the result appearing in register X.

2. [16] Write a MIX subroutine having the following characteristics:

Calling sequence: JMP RANDM

Entry conditions: Location XRAND contains an integer X .

Exit conditions: $X \leftarrow rA \leftarrow (aX + c) \bmod w$, $rX \leftarrow 0$, overflow off.

(Thus a call on this subroutine will produce the next random number of a linear congruential sequence.)

3. [20] How can the constant $(w - 1)$ be specified in general in the MIX assembly language, regardless of the value of the byte size?

► 4. [21] Discuss the calculation of linear congruential sequences with $m = 2^{32}$ on two's-complement machines such as the System/370 series.

5. [20] Given that m is less than the word size, and that x, y are nonnegative integers less than m , show that the difference $(x - y) \bmod m$ may be computed in just four MIX instructions, without requiring any division. What is the best code for the sum $(x + y) \bmod m$?

► 6. [20] The previous exercise suggests that subtraction mod m is easier to perform than addition mod m . Discuss sequences generated by the rule

$$X_{n+1} = (aX_n - c) \bmod m.$$

Are these sequences essentially different from linear congruential sequences as defined in the text? Are they more suited to efficient computer calculation?

7. [M24] What patterns can you spot in Table 1?

► 8. [20] Write a MIX program analogous to (2) that computes $(aX) \bmod (w - 1)$. The values 0 and $w - 1$ are to be treated as equivalent in the input and output of your program.

9. [29] Write a MIX program analogous to the one in exercise 8, but it should compute $(aX) \bmod (w - 2)$.

3.2.1.2. Choice of multiplier. In this section we shall show how to choose the multiplier a so as to give the *period of maximum length*. A long period is essential for any sequence that is to be used as a source of random numbers; indeed, we would hope that the period contains considerably more numbers than will ever be used in a single application. Therefore we shall concern ourselves in this section with the question of period length. The reader should keep in mind, however, that a long period is only one desirable criterion for the randomness of our sequence. For example, when $a = c = 1$, the sequence is simply $X_{n+1} = (X_n + 1) \bmod m$, and this obviously has a period of length m , yet it is anything but random. Other considerations affecting the choice of a multiplier will be given later in this chapter.

Since only m different values are possible, the period surely cannot be longer than m . Can we achieve the maximum length, m ? The example above shows that it is always possible, although the choice $a = c = 1$ does not yield a desirable sequence. Let us investigate *all* possible choices of a , c , and X_0 that give a period of length m . It turns out that all such values of the parameters can be characterized very simply; when m is the product of distinct primes, only $a = 1$ will produce the full period, but when m is divisible by a high power of some prime there is considerable latitude in the choice of a . The following theorem makes it easy to tell if the maximum period is achieved.

Theorem A. *The linear congruential sequence defined by m , a , c , and X_0 has period length m if and only if*

- i) c is relatively prime to m ;
- ii) $b = a - 1$ is a multiple of p , for every prime p dividing m ;
- iii) b is a multiple of 4, if m is a multiple of 4.

The ideas used in the proof of this theorem go back at least a hundred years. The first proof of the theorem in this particular form was given by M. Greenberger in the special case $m = 2^e$ [see JACM 8 (1961), 383–389], and the sufficiency of conditions (i), (ii), and (iii) in the general case was shown by Hull and Dobell [see SLAM Review 4 (1962), 230–254]. To prove the theorem we will first consider some auxiliary number-theoretic results that are of interest in themselves.

Lemma P. *Let p be a prime number, and let e be a positive integer, where $p^e > 2$. If*

$$x \equiv 1 \pmod{p^e}, \quad x \not\equiv 1 \pmod{p^{e+1}}, \quad (1)$$

then

$$x^p \equiv 1 \pmod{p^{e+1}}, \quad x^p \not\equiv 1 \pmod{p^{e+2}}. \quad (2)$$

Proof. We have $x = 1 + qp^e$ for some integer q that is not a multiple of p . By the binomial formula

$$\begin{aligned} x^p &= 1 + \binom{p}{1}qp^e + \cdots + \binom{p}{p-1}q^{p-1}p^{(p-1)e} + q^p p^{pe} \\ &= 1 + qp^{e+1} \left(1 + \frac{1}{p} \binom{p}{2}qp^e + \frac{1}{p} \binom{p}{3}q^2 p^{2e} + \cdots + \frac{1}{p} \binom{p}{p}q^{p-1}p^{(p-1)e} \right). \end{aligned}$$

The quantity in parentheses is an integer, and, in fact, every term inside the parentheses is a multiple of p except the first term. For if $1 < k < p$, the binomial coefficient $\binom{p}{k}$ is divisible by p (cf. exercise 1.2.6–10), hence

$$\frac{1}{p} \binom{p}{k} q^{k-1} p^{(k-1)e}$$

is divisible by $p^{(k-1)e}$; and the last term is $q^{p-1}p^{(p-1)e-1}$, which is divisible by p since $(p-1)e > 1$ when $p^e > 2$. So $x^p \equiv 1 + qp^{e+1} \pmod{p^{e+2}}$, and this completes the proof. (Note: A generalization of this result appears in exercise 3.2.2–11(a).) ■

Lemma Q. *Let the decomposition of m into prime factors be*

$$m = p_1^{e_1} \cdots p_t^{e_t}. \quad (3)$$

The length λ of the period of the linear congruential sequence determined by (X_0, a, c, m) is the least common multiple of the lengths λ_j of the periods of the linear congruential sequences $(X_0 \bmod p_j^{e_j}, a \bmod p_j^{e_j}, c \bmod p_j^{e_j}, p_j^{e_j})$, $1 \leq j \leq t$.

Proof. By induction on t , it suffices to prove that if m_1 and m_2 are relatively prime, the length λ of the linear congruential sequence determined by the parameters $(X_0, a, c, m_1 m_2)$ is the least common multiple of the lengths λ_1 and λ_2 of the periods of the sequences determined by $(X_0 \bmod m_1, a \bmod m_1, c \bmod m_1, m_1)$ and $(X_0 \bmod m_2, a \bmod m_2, c \bmod m_2, m_2)$. We observed in the previous section, Eq. (4), that if the elements of these three sequences are respectively denoted by X_n , Y_n , and Z_n , we will have

$$Y_n = X_n \bmod m_1 \quad \text{and} \quad Z_n = X_n \bmod m_2, \quad \text{for all } n \geq 0.$$

Therefore, by Law D of Section 1.2.4, we find that

$$X_n = X_k \quad \text{if and only if} \quad Y_n = Y_k \quad \text{and} \quad Z_n = Z_k. \quad (4)$$

Let λ' be the least common multiple of λ_1 and λ_2 ; we wish to prove that $\lambda' = \lambda$. Since $X_n = X_{n+\lambda}$ for all suitably large n , we have $Y_n = Y_{n+\lambda}$ (hence λ is a multiple of λ_1) and $Z_n = Z_{n+\lambda}$ (hence λ is a multiple of λ_2), so we must have $\lambda \geq \lambda'$. Furthermore, we know that $Y_n = Y_{n+\lambda'}$ and $Z_n = Z_{n+\lambda'}$ for all suitably large n ; therefore, by (4), $X_n = X_{n+\lambda'}$. This proves $\lambda \leq \lambda'$. ■

Now we are ready to prove Theorem A. Because of Lemma Q, it suffices to prove the theorem when m is a power of a prime number. For

$$p_1^{e_1} \cdots p_t^{e_t} = \lambda = \text{lcm}(\lambda_1, \dots, \lambda_t) \leq \lambda_1 \cdots \lambda_t \leq p_1^{e_1} \cdots p_t^{e_t}$$

can be true if and only if $\lambda_j = p_j^{e_j}$ for $1 \leq j \leq t$.

Therefore, assume that $m = p^e$, where p is prime and e is a positive integer. The theorem is obviously true when $a = 1$, so we may take $a > 1$. The period can be of length m if and only if each possible integer $0 \leq x < m$ occurs in the period, since no value occurs in the period more than once. Therefore the period is of length m if and only if the period of the sequence with $X_0 = 0$ is of length m , and we are justified in supposing that $X_0 = 0$. By formula 3.2.1-6 we have

$$X_n = \left(\frac{a^n - 1}{a - 1} \right) c \bmod m. \quad (5)$$

If c is not relatively prime to m , this value X_n could never be equal to 1, so condition (i) of the theorem is necessary. The period has length m if and only if the smallest positive value of n for which $X_n = X_0 = 0$ is $n = m$. By (5) and condition (i), our theorem now reduces to proving the following fact:

Lemma R. Assume that $1 < a < p^e$, where p is prime. If λ is the smallest positive integer for which $(a^\lambda - 1)/(a - 1) \equiv 0$ (modulo p^e), then

$$\lambda = p^e \quad \text{if and only if} \quad \begin{cases} a \equiv 1 \pmod{p} & \text{when } p > 2, \\ a \equiv 1 \pmod{4} & \text{when } p = 2. \end{cases}$$

Proof. Assume that $\lambda = p^e$. If $a \not\equiv 1 \pmod{p}$, then $(a^n - 1)/(a - 1) \equiv 0 \pmod{p^e}$ if and only if $a^n - 1 \equiv 0 \pmod{p^e}$. The condition $a^{p^e} - 1 \equiv 0 \pmod{p^e}$ then implies that $a^{p^e} \equiv 1 \pmod{p}$; but by Theorem 1.2.4F we have $a^{p^e} \equiv a \pmod{p}$, hence $a \not\equiv 1 \pmod{p}$ leads to a contradiction. And if $p = 2$ and $a \equiv 3 \pmod{4}$, we have $(a^{2^{e-1}} - 1)/(a - 1) \equiv 0 \pmod{2^e}$ by exercise 8. These arguments show that it is necessary in general to have $a = 1 + qp^f$, where $p^f > 2$ and q is not a multiple of p , whenever $\lambda = p^e$.

It remains to be shown that this condition is sufficient to make $\lambda = p^e$. By repeated application of Lemma P, we find that

$$a^{p^g} \equiv 1 \pmod{p^{f+g}}, \quad a^{p^g} \not\equiv 1 \pmod{p^{f+g+1}},$$

for all $g \geq 0$, and therefore

$$\begin{aligned} (a^{p^g} - 1)/(a - 1) &\equiv 0 \pmod{p^g}, \\ (a^{p^g} - 1)/(a - 1) &\not\equiv 0 \pmod{p^{g+1}}. \end{aligned} \tag{6}$$

In particular, $(a^{p^e} - 1)/(a - 1) \equiv 0 \pmod{p^e}$. Now the congruential sequence $(0, a, 1, p^e)$ has $X_n = (a^n - 1)/(a - 1) \pmod{p^e}$; therefore it has a period of length λ , that is, $X_n = 0$ if and only if n is a multiple of λ . Hence p^e is a multiple of λ . This can happen only if $\lambda = p^g$ for some g , and the relations in (6) imply that $\lambda = p^e$, completing the proof. ■

The proof of Theorem A is now complete. ■

We will conclude this section by considering the special case of pure multiplicative generators, when $c = 0$. Although the random number generation process is slightly faster in this case, Theorem A shows us that the maximum period length cannot be achieved. In fact, this is quite obvious, since the sequence now satisfies the relation

$$X_{n+1} = aX_n \pmod{m}, \tag{7}$$

and the value $X_n = 0$ should never appear, lest the sequence degenerate to zero. In general, if d is any divisor of m and if X_n is a multiple of d , all succeeding elements X_{n+1}, X_{n+2}, \dots of the multiplicative sequence will be multiples of d . So when $c = 0$, we will want X_n to be relatively prime to m for all n , and this limits the length of the period to at most $\varphi(m)$, the number of integers between 0 and m that are relatively prime to m .

It may be possible to achieve an acceptably long period even if we stipulate that $c = 0$. Let us now try to find conditions on the multiplier so that the period is as long as possible in this special case.

According to Lemma Q, the period of the sequence depends entirely on the periods of the sequences when $m = p^e$, so let us consider that situation. We have $X_n = a^n X_0 \pmod{p^e}$, and it is clear that the period will be of length 1 if a is a multiple of p , so we take a to be relatively prime to p . Then the period is the smallest integer λ such that $X_0 = a^\lambda X_0 \pmod{p^e}$. If the greatest common divisor of X_0 and p^e is p^f , this condition is equivalent to

$$a^\lambda \equiv 1 \pmod{p^{e-f}}. \quad (8)$$

By Euler's theorem (exercise 1.2.4–28), $a^{\varphi(p^{e-f})} \equiv 1 \pmod{p^{e-f}}$; hence λ is a divisor of

$$\varphi(p^{e-f}) = p^{e-f-1}(p-1).$$

When a is relatively prime to m , the smallest integer λ for which $a^\lambda \equiv 1 \pmod{m}$ is conventionally called the *order of a modulo m* . Any such value of a that has the *maximum* possible order modulo m is called a *primitive element* modulo m .

Let $\lambda(m)$ denote the order of a primitive element, i.e., the maximum possible order, modulo m . The remarks above show that $\lambda(p^e)$ is a divisor of $p^{e-1}(p-1)$; with a little care (see exercises 11 through 16 below) we can give the precise value of $\lambda(m)$ in all cases as follows:

$$\begin{aligned} \lambda(2) &= 1, & \lambda(4) &= 2, & \lambda(2^e) &= 2^{e-2} \quad \text{if } e \geq 3. \\ \lambda(p^e) &= p^{e-1}(p-1), & \text{if } p > 2. \\ \lambda(p_1^{e_1} \dots p_t^{e_t}) &= \text{lcm}(\lambda(p_1^{e_1}), \dots, \lambda(p_t^{e_t})). \end{aligned} \quad (9)$$

Our remarks may be summarized in the following theorem:

Theorem B. [R. D. Carmichael, *Bull. Amer. Math. Soc.* **16** (1910), 232–238.] *The maximum period possible when $c = 0$ is $\lambda(m)$, where $\lambda(m)$ is defined in (9). This period is achieved if*

- i) X_0 is relatively prime to m ;
- ii) a is a primitive element modulo m . ■

Note that we can obtain a period of length $m - 1$ if m is prime; this is just one less than the maximum length, so for all practical purposes such a period is as long as we want.

The question now is, how can we find primitive elements modulo m ? The exercises at the close of this section tell us that there is a fairly simple answer when m is prime or a power of a prime, namely the results stated in our next theorem.

Theorem C. *The number a is a primitive element modulo p^e if and only if*

- i) $p^e = 2$, a is odd; or $p^e = 4$, $a \bmod 4 = 3$;
 or $p^e = 8$, $a \bmod 8 = 3, 5, 7$; or $p = 2$, $e \geq 4$, $a \bmod 8 = 3, 5$;

or

- ii) p is odd, $e = 1$, $a \not\equiv 0$ (modulo p), and $a^{(p-1)/q} \not\equiv 1$ (modulo p)
 for any prime divisor q of $p - 1$;

or

- iii) p is odd, $e > 1$, a satisfies (ii), and $a^{p-1} \not\equiv 1$ (modulo p^2). ■

Conditions (ii) and (iii) of this theorem are readily tested on a computer for large values of p , by using the efficient methods for evaluating powers discussed in Section 4.6.3. Theorem C applies to powers of primes only; if we are given values a_j that are primitive modulo $p_j^{e_j}$, it is possible to find a single value a such that $a \equiv a_j$ (modulo $p_j^{e_j}$), for $1 \leq j \leq t$, using the “Chinese remainder algorithm” discussed in Section 4.3.2, and this number a will be a primitive element modulo $p_1^{e_1} \dots p_t^{e_t}$. Hence there is a reasonably efficient way to construct multipliers satisfying the condition of Theorem B, for any desired value of m , although the calculations can be somewhat lengthy in the general case.

In the common case $m = 2^e$, with $e \geq 4$, the conditions above simplify to the single requirement that $a \equiv 3$ or 5 (modulo 8). In this case, one-fourth of all possible multipliers give the maximum period.

The second most common case is when $m = 10^e$. Using Lemmas P and Q, it is not difficult to obtain necessary and sufficient conditions for the achievement of the maximum period in the case of a decimal computer (cf. exercise 18):

Theorem D. *If $m = 10^e$, $e \geq 5$, $c = 0$, and X_0 is not a multiple of 2 or 5, the period of the linear congruential sequence is $5 \times 10^{e-2}$ if and only if $a \bmod 200$ equals one of the following 32 values:*

$$\begin{aligned} 3, 11, 13, 19, 21, 27, 29, 37, 53, 59, 61, 67, 69, 77, 83, 91, 109, 117, \\ 123, 131, 133, 139, 141, 147, 163, 171, 173, 179, 181, 187, 189, 197. \quad \blacksquare \end{aligned} \quad (10)$$

EXERCISES

- [10] What is the length of the period of the linear congruential sequence with $X_0 = 5772156648$, $a = 3141592621$, $c = 2718281829$, and $m = 10000000000$?
- [10] Are the following two conditions sufficient to guarantee the maximum length period, when m is a power of 2? (i) c is odd; (ii) $a \bmod 4 = 1$.
- [13] Suppose that $m = 10^e$, where $e \geq 2$, and suppose further that c is odd and not a multiple of 5. Show that the linear congruential sequence will have the maximum length period if and only if $a \bmod 20 = 1$.

4. [M20] When a and c satisfy the conditions of Theorem A, and when $m = 2^e$, $X_0 = 0$, what is the value of X_{2^e-1} ?

5. [14] Find all multipliers a that satisfy the conditions of Theorem A when $m = 2^{35} + 1$. (The prime factors of m may be found in Table 3.2.1.1-1.)

► 6. [20] Find all multipliers a that satisfy the conditions of Theorem A when $m = 10^6 - 1$. (See Table 3.2.1.1-1.)

► 7. [M23] The period of a congruential sequence need not start with X_0 , but we can always find indices $\mu \geq 0$ and $\lambda > 0$ such that $X_{n+\lambda} = X_n$ whenever $n \geq \mu$, and for which μ and λ are the smallest possible values with this property. (Cf. exercises 3.1-6 and 3.2.1-1.) If μ_j and λ_j are the indices corresponding to the sequences $(X_0 \bmod p_j^{e_j}, a \bmod p_j^{e_j}, c \bmod p_j^{e_j}, p_j^{e_j})$, and if μ and λ correspond to the sequence $(X_0, a, c, p_1^{e_1} \dots p_t^{e_t})$, Lemma Q states that λ is the least common multiple of $\lambda_1, \dots, \lambda_t$. What is the value of μ in terms of the values of μ_1, \dots, μ_t ? What is the maximum possible value of μ obtainable by varying X_0 , a , and c , when $m = p_1^{e_1} \dots p_t^{e_t}$ is fixed?

8. [M20] Show that if $a \bmod 4 = 3$, we have $(a^{2^e-1} - 1)/(a - 1) \equiv 0$ (modulo 2^e) when $e > 1$. (Use Lemma P.)

► 9. [M22] (W. E. Thomson.) When $c = 0$ and $m = 2^e \geq 16$, Theorems B and C say that the period has length 2^{e-2} if and only if the multiplier a satisfies $a \bmod 8 = 3$ or $a \bmod 8 = 5$. Show that every such sequence is essentially a linear congruential sequence with $m = 2^{e-2}$, having full period, in the following sense:

a) If $X_{n+1} = (4c + 1)X_n \bmod 2^e$, and $X_n = 4Y_n + 1$, then

$$Y_{n+1} = ((4c + 1)Y_n + c) \bmod 2^{e-2}.$$

b) If $X_{n+1} = (4c - 1)X_n \bmod 2^e$, and $X_n = ((-1)^n(4Y_n + 1)) \bmod 2^e$, then

$$Y_{n+1} = ((1 - 4c)Y_n - c) \bmod 2^{e-2}.$$

[Note: In these formulas, c is an odd integer. The literature contains several statements to the effect that sequences with $c = 0$ satisfying Theorem B are somehow more random than sequences satisfying Theorem A, in spite of the fact that the period is only one-fourth as long in the case of Theorem B. This exercise refutes such statements; in essence, one gives up two bits of the word length in order to save the addition of c , when m is a power of 2.]

10. [M21] For what values of m is $\lambda(m) = \varphi(m)$?

► 11. [M28] Let x be an odd integer greater than 1. (a) Show that there exists a unique integer $f > 1$ such that $x \equiv 2^f \pm 1$ (modulo 2^{f+1}). (b) Given that $1 < x < 2^e - 1$ and that f is the corresponding integer from part (a), show that the order of x modulo 2^e is 2^{e-f} . (c) In particular, this proves Theorem C(i).

12. [M26] Let p be an odd prime. If $e > 1$, prove that a is a primitive element modulo p^e if and only if a is a primitive element modulo p and $a^{p-1} \not\equiv 1$ (modulo p^2). (For the purposes of this exercise, assume that $\lambda(p^e) = p^{e-1}(p-1)$. This fact is proved in exercises 14 and 16 below.)

13. [M22] Let p be prime. Given that a is not a primitive element modulo p , show that either a is a multiple of p or $a^{(p-1)/q} \equiv 1$ (modulo p) for some prime number q that divides $p-1$.

14. [M18] If $e > 1$ and p is an odd prime, and if a is a primitive element modulo p , prove that either a or $a + p$ is a primitive element modulo p^e . [Hint: See exercise 12.]

15. [M29] (a) Let a_1, a_2 be relatively prime to m , and let their orders modulo m be λ_1, λ_2 , respectively. If λ is the least common multiple of λ_1 and λ_2 , prove that $a_1^{\kappa_1} a_2^{\kappa_2}$ has order λ modulo m , for suitable integers κ_1, κ_2 . [Hint: Consider first the case that λ_1 is relatively prime to λ_2 .] (b) Let $\lambda(m)$ be the maximum order of any element modulo m . Prove that $\lambda(m)$ is a multiple of the order of each element modulo m ; that is, prove that $a^{\lambda(m)} \equiv 1$ (modulo m) whenever a is relatively prime to m .

► 16. [M24] Let p be a prime number. (a) Let $f(x) = x^n + c_1x^{n-1} + \dots + c_n$, where the c 's are integers. Given that a is an integer for which $f(a) \equiv 0$ (modulo p), show that there exists a polynomial $q(x) = x^{n-1} + q_1x^{n-2} + \dots + q_{n-1}$ with integer coefficients such that $f(x) \equiv (x - a)q(x)$ (modulo p) for all integers x . (b) Let $f(x)$ be a polynomial as in (a). Show that $f(x)$ has at most n distinct “roots” modulo p ; that is, there are at most n integers a , with $0 \leq a < p$, such that $f(a) \equiv 0$ (modulo p). (c) Because of exercise 15(b), the polynomial $f(x) = x^{\lambda(p)} - 1$ has $p - 1$ distinct roots; hence there is an integer a with order $p - 1$.

17. [M26] Not all of the values listed in Theorem D would be found by the text's construction; for example, 11 is not primitive modulo 5^e . How can this be possible, when 11 is primitive modulo 10^e , according to Theorem D? Which of the values listed in Theorem D are primitive elements modulo both 2^e and 5^e ?

18. [M25] Prove Theorem D. (Cf. the previous exercise.)

19. [40] Make a table of some suitable multipliers, a , for each of the values of m listed in Table 3.2.1.1-1, assuming that $c = 0$.

► 20. [M24] (G. Marsaglia.) The purpose of this exercise is to study the period length of an arbitrary linear congruential sequence. Let $Y_n = 1 + a + \dots + a^{n-1}$, so that $X_n = (AY_n + X_0) \bmod m$ for some constant A by Eq. 3.2.1-8. (a) Prove that the period length of $\langle X_n \rangle$ is the period length of $\langle Y_n \bmod m' \rangle$, where $m' = m/\gcd(A, m)$. (b) Prove that the period length of $\langle Y_n \bmod p^e \rangle$ satisfies the following when p is prime: (i) If $a \bmod p = 0$, it is 1. (ii) If $a \bmod p = 1$, it is p^e , except when $p = 2$ and $e \geq 2$ and $a \bmod 4 = 3$. (iii) If $p = 2$, $e \geq 2$, and $a \bmod 4 = 3$, it is twice the order of a modulo p^e (cf. exercise 11), unless $a \equiv -1$ (modulo 2^e) when it is 2. (iv) If $a \bmod p > 1$, it is the order of a modulo p^e .

21. [M25] In a linear congruential sequence of maximum period, let $X_0 = 0$ and let s be the least positive integer such that $a^s \equiv 1$ (modulo m). Prove that $\gcd(X_s, m) = s$.

3.2.1.3. Potency. In the preceding section, we showed that the maximum period can be obtained when $b = a - 1$ is a multiple of each prime dividing m ; and b must also be a multiple of 4 if m is a multiple of 4. If z is the radix of the machine being used—so that $z = 2$ for a binary computer, and $z = 10$ for a decimal computer—and if m is the word size z^e , the multiplier .

$$a = z^k + 1, \quad 2 \leq k < e \tag{1}$$

satisfies these conditions. Theorem 3.2.1.2A also says that we may take $c = 1$. The recurrence relation now has the form

$$X_{n+1} = ((z^k + 1)X_n + 1) \bmod z^e, \tag{2}$$

and this equation suggests that we can avoid the multiplication; merely shifting and adding will suffice.

For example, suppose that $a = B^2 + 1$, where B is the byte size of MIX. The code

LDA	X
SLA	2
ADD	X
INCA	1

(3)

can be used in place of the instructions given in Section 3.2.1.1, and the execution time decreases from $16u$ to $7u$.

For this reason, multipliers having form (1) have been widely discussed in the literature, and indeed they have been recommended by many authors. However, the early years of experimentation with this method showed that *multipliers having the simple form in (1) should be avoided*. The generated numbers just aren't random enough.

Later in this chapter we shall be discussing some rather sophisticated theory that accounts for the badness of all the linear congruential random number generators known to be bad. However, some generators (such as (2)) are sufficiently awful that a comparatively simple theory can be used to dispense with them. This simple theory is related to the concept of "potency," which we shall now discuss.

The *potency* of a linear congruential sequence with maximum period is defined to be the least integer s such that

$$b^s \equiv 0 \pmod{m}. \quad (4)$$

(Such an integer s will always exist when the multiplier satisfies the conditions of Theorem 3.2.1.2A, since b is a multiple of every prime dividing m .)

We may analyze the randomness of the sequence by taking $X_0 = 0$, since 0 occurs somewhere in the period. With this assumption, we have

$$X_n = ((a^n - 1)c/b) \pmod{m},$$

and if we expand $a^n - 1 = (b + 1)^n - 1$ by the binomial theorem, we find that

$$X_n = c \left(n + \binom{n}{2} b + \cdots + \binom{n}{s} b^{s-1} \right) \pmod{m}. \quad (5)$$

All terms in b^s, b^{s+1} , etc., may be ignored, since they are multiples of m .

Equation (5) can be instructive, so we shall consider some special cases. If $a = 1$, the potency is 1; and $X_n \equiv cn \pmod{m}$, as we have already observed, so the sequence is surely not random. If the potency is 2, we have $X_n \equiv cn + cb\binom{n}{2}$, and again the sequence is not very random; indeed,

$$X_{n+1} - X_n \equiv c + cbn$$

in this case, so the differences between consecutively generated numbers change in a simple way from one value of n to the next. The point (X_n, X_{n+1}, X_{n+2}) always lies on one of the four planes

$$\begin{array}{ll} x - 2y + z = d + m, & x - 2y + z = d - m, \\ x - 2y + z = d, & x - 2y + z = d - 2m, \end{array}$$

in three-dimensional space, where $d = cb \bmod m$.

If the potency is 3, the sequence begins to look somewhat more random, but there is a high degree of dependency between X_n , X_{n+1} , and X_{n+2} ; tests show that sequences with potency 3 are still not sufficiently good. Reasonable results have been reported when the potency is 4 or more, but these have been disputed by other people. A potency of at least 5 would seem to be required for sufficiently random values.

Suppose, for example, that $m = 2^{35}$ and $a = 2^k + 1$. Then $b = 2^k$, so we find that when $k \geq 18$, the value $b^2 = 2^{2k}$ is a multiple of m : the potency is 2. If $k = 17, 16, \dots, 12$, the potency is 3, and a potency of 4 is achieved for $k = 11, 10, 9$. The only acceptable multipliers, from the standpoint of potency, therefore have $k \leq 8$. This means $a \leq 257$, and we shall see later that small multipliers are also to be avoided. We have now eliminated all multipliers of the form $2^k + 1$ when $m = 2^{35}$.

When m is equal to $w \pm 1$, where w is the word size, m is generally not divisible by high powers of primes, and a high potency is impossible (see exercise 6). So in this case, the maximum-period method should not be used; the pure-multiplication method with $c = 0$ should be applied instead.

It must be emphasized that high potency is necessary but not sufficient for randomness; we use the concept of potency only to reject impotent generators, not to accept the potent ones. Linear congruential sequences should pass the “spectral test” discussed in Section 3.3.4 before they are considered to be acceptably random.

EXERCISES

1. [M10] Show that, no matter what the byte size B of MIX happens to be, the code (3) yields a random number generator of maximum period.
2. [10] What is the potency of the generator represented by the MIX code (3)?
3. [11] When $m = 2^{35}$, what is the potency of the linear congruential sequence with $a = 3141592621$? What is the potency if the multiplier is $a = 2^{23} + 2^{14} + 2^2 + 1$?
4. [15] Show that if $m = 2^e \geq 8$, maximum potency is achieved when $a \bmod 8 = 5$.
5. [M20] Given that $m = p_1^{e_1} \dots p_t^{e_t}$ and $a = 1 + kp_1^{f_1} \dots p_t^{f_t}$, where a satisfies the conditions of Theorem 3.2.1.2A and k is relatively prime to m , show that the potency is $\max(\lceil e_1/f_1 \rceil, \dots, \lceil e_t/f_t \rceil)$.
- 6. [20] Which of the values of $m = w \pm 1$ in Table 3.2.1.1-1 can be used in a linear congruential sequence of maximum period whose potency is 4 or more? (Use the result of exercise 5.)

7. [M20] When a satisfies the conditions of Theorem 3.2.1.2A, it is relatively prime to m ; hence there is a number a' such that $aa' \equiv 1$ (modulo m). Show that a' can be expressed simply in terms of b .

► 8. [M26] A random number generator defined by $X_{n+1} = (2^{17} + 3)X_n \bmod 2^{35}$ and $X_0 = 1$ was subjected to the following test: Let $Y_n = \lfloor 10X_n/2^{35} \rfloor$; then Y_n should be a random digit between 0 and 9, and the triples $(Y_{3n}, Y_{3n+1}, Y_{3n+2})$ should take on each of the 1000 possible values from $(0, 0, 0)$ to $(9, 9, 9)$ with equal probability. But with 30000 values of n tested, some triples hardly ever occurred, and others occurred much more often than they should have. Can you account for this failure?

3.2.2. Other Methods

Of course, linear congruential sequences are not the only sources of random numbers that have been proposed for computer use. In this section we shall review the most significant alternatives; some of these methods are quite important, while others are interesting chiefly because they are not as good as a person might expect.

One of the common fallacies encountered in connection with random number generation is the idea that we can take a good generator and modify it a little, in order to get an “even-more-random” sequence. This is often false. For example, we know that

$$X_{n+1} = (aX_n + c) \bmod m \quad (1)$$

leads to reasonably good random numbers; wouldn’t the sequence produced by

$$X_{n+1} = ((aX_n) \bmod (m+1) + c) \bmod m \quad (2)$$

be even *more* random? The answer is, the new sequence is probably a great deal less random. For the whole theory breaks down, and in the absence of any theory about the behavior of the sequence (2), we come into the area of generators of the type $X_{n+1} = f(X_n)$ with the function f chosen at random; exercises 3.1–11 through 3.1–15 show that these sequences probably behave much more poorly than the sequences obtained from the more disciplined function (1).

Let us consider another approach, in an attempt to get “more random” numbers. The linear congruential method can be generalized to, say, a quadratic congruential method:

$$X_{n+1} = (dX_n^2 + aX_n + c) \bmod m. \quad (3)$$

Exercise 8 generalizes Theorem 3.2.1.2A to obtain necessary and sufficient conditions on a , c , and d such that the sequence defined by (3) has a period of the maximum length m ; the restrictions are not much more severe than in the linear method.

An interesting quadratic method has been proposed by R. R. Coveyou when m is a power of two; let

$$X_0 \bmod 4 = 2, \quad X_{n+1} = X_n(X_n + 1) \bmod 2^e, \quad n \geq 0. \quad (4)$$

This sequence can be computed with about the same efficiency as (1), without any worries of overflow. It has an interesting connection with von Neumann's original middle-square method: If we let Y_n be $2^e X_n$, so that Y_n is a double-precision number obtained by placing e zeros to the right of the binary representation of X_n , then Y_{n+1} consists of precisely the middle $2e$ digits of $Y_n^2 + 2^e Y_n$! In other words, Coveyou's method is almost identical to a somewhat degenerate double-precision middle-square method, yet it is guaranteed to have a long period; further evidence of its randomness is proved in exercise 3.3.4-25.

Other generalizations of Eq. (1) also suggest themselves; for example, we might try to extend the period length of the sequence. The period of a linear congruential sequence is extremely long; when m is approximately the word size of the computer, we usually get periods on the order of 10^9 or more, so that typical calculations will use only a very small portion of the sequence. On the other hand, when we discuss the idea of "accuracy" in Section 3.3.4 we will see that the period length influences the degree of randomness achievable in a sequence. Therefore it is occasionally desirable to seek a longer period, and several methods are available for this purpose. One technique is to make X_{n+1} depend on both X_n and X_{n-1} , instead of just on X_n ; then the period length can be as high as m^2 , since the sequence will not begin to repeat until we have $(X_{n+\lambda}, X_{n+\lambda+1}) = (X_n, X_{n+1})$.

The simplest sequence in which X_{n+1} depends on more than one of the preceding values is the Fibonacci sequence,

$$X_{n+1} = (X_n + X_{n-1}) \bmod m. \quad (5)$$

This generator was considered in the early 1950s, and it usually gives a period length greater than m ; but tests have shown that the numbers produced by the Fibonacci recurrence (5) are definitely *not* satisfactorily random, and so at the present time the main interest in (5) as a source of random numbers is that it makes a nice "bad example." We may also consider generators of the form

$$X_{n+1} = (X_n + X_{n-k}) \bmod m, \quad (6)$$

when k is a comparatively large value. These were introduced by Green, Smith, and Klem [JACM 6 (1959), 527-537], who reported that, when $k \leq 15$, the sequence fails to pass the "gap test" described in Section 3.3.2, although when $k = 16$ the test was satisfactory.

A much better type of additive generator was devised in 1958 by G. J. Mitchell and D. P. Moore [unpublished], who suggested the somewhat unusual sequence defined by

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \quad n \geq 55, \quad (7)$$

where m is even, and where X_0, \dots, X_{54} are arbitrary integers not all even. The constants 24 and 55 in this definition were not chosen at random, they are special values that happen to have the property that the least significant bits $\langle X_n \bmod 2 \rangle$ will have a period of length $2^{55} - 1$. Therefore the sequence $\langle X_n \rangle$ must have a period at least this long. Exercise 11, which explains how to calculate the period length of such sequences, proves that (7) has a period of length $2^f(2^{55} - 1)$ for some f , $0 \leq f < e$, when $m = 2^e$.

At first glance Eq. (7) may not seem to be extremely well suited to machine implementation, but in fact there is a very efficient way to generate the sequence using a cyclic list:

Algorithm A (Additive number generator). Memory cells $Y[1], Y[2], \dots, Y[55]$ are initially set to the values $X_{54}, X_{53}, \dots, X_0$, respectively; j is initially equal to 24 and k is 55. Successive performances of this algorithm will produce the numbers X_{55}, X_{56}, \dots as output.

- A1. [Add.] (If we are about to output X_n at this point, $Y[j]$ now equals X_{n-24} and $Y[k]$ equals X_{n-55} .) Set $Y[k] \leftarrow (Y[k] + Y[j]) \bmod 2^e$, and output $Y[k]$.
- A2. [Advance.] Decrease j and k by 1. If now $j = 0$, set $j \leftarrow 55$; otherwise if $k = 0$, set $k \leftarrow 55$. ■

This algorithm in MIX is simply the following:

Program A (Additive number generator). Assuming that index registers 5 and 6 are not touched by the remainder of the program in which this routine is embedded, the following code performs Algorithm A and leaves the result in register A. $\text{rI5} \equiv j$, $\text{rI6} \equiv k$.

```

LDA  Y, 6  A1. Add.
ADD  Y, 5   $Y_k + Y_j$  (overflow possible)
STA  Y, 6   $\rightarrow Y_k$ .
DEC5 1   A2. Advance.  $j \leftarrow j - 1$ .
DEC6 1    $k \leftarrow k - 1$ .
J5P  **2
ENT5 55  If  $j = 0$ , set  $j \leftarrow 55$ .
J6P  **2
ENT6 55  If  $k = 0$ , set  $k \leftarrow 55$ . ■

```

This generator is usually faster than the previous methods we have been discussing, since it does not require any multiplication. Besides its speed, it has the longest period we have seen yet; and it has consistently produced reliable results, in extensive tests since its invention in 1958. Furthermore, as Richard Brent has observed, it can be made to work correctly with floating point numbers, avoiding the need to convert between integers and fractions (cf. exercise 23). Therefore it may well prove to be the very best source of random numbers for practical purposes. The only reason it is difficult to recommend sequence (7) wholeheartedly is that there is still very little theory to prove that it does or does not have desirable randomness properties; essentially all we know for sure

Table 1

SUBSCRIPT PAIRS YIELDING LONG PERIODS MOD 2

(1, 2)	(1, 15)	(5, 23)	(7, 31)	(5, 47)	(21, 52)	(18, 65)	(28, 73)	(2, 93)
(1, 3)	(4, 15)	(9, 23)	(13, 31)	(14, 47)	(24, 55)	(32, 65)	(31, 73)	(21, 94)
(1, 4)	(7, 15)	(3, 25)	(13, 33)	(20, 47)	(7, 57)	(9, 68)	(9, 79)	(11, 95)
(2, 5)	(3, 17)	(7, 25)	(2, 35)	(21, 47)	(22, 57)	(33, 68)	(19, 79)	(17, 95)
(1, 6)	(5, 17)	(3, 28)	(11, 36)	(9, 49)	(19, 58)	(6, 71)	(4, 81)	(6, 97)
(1, 7)	(6, 17)	(9, 28)	(4, 39)	(12, 49)	(1, 60)	(9, 71)	(16, 81)	(12, 97)
(3, 7)	(7, 18)	(13, 28)	(8, 39)	(15, 49)	(11, 60)	(18, 71)	(35, 81)	(33, 97)
(4, 9)	(3, 20)	(2, 29)	(14, 39)	(22, 49)	(1, 63)	(20, 71)	(13, 84)	(34, 97)
(3, 10)	(2, 21)	(3, 31)	(3, 41)	(3, 52)	(5, 63)	(35, 71)	(13, 87)	(11, 98)
(2, 11)	(1, 22)	(6, 31)	(20, 41)	(19, 52)	(31, 63)	(25, 73)	(38, 89)	(27, 98)

For each pair (l, k) , the pair $(k-l, k)$ is also valid (see exercise 24), hence only values of $l \leq k/2$ are listed here. For extensions of this table, see N. Zierler and J. Brillhart, *Information and Control* 13 (1968), 541–554; 14 (1969), 566–569; 15 (1969), 67–69.

is that the period is very long, and this is not enough. John Reiser (Ph. D. thesis, Stanford Univ., 1977) has shown, however, that an additive sequence like (7) will be well distributed in high dimensions, provided that a certain plausible conjecture is true (cf. exercise 26).

The fact that the special numbers (24, 55) in (7) work so well follows from theoretical results developed in some of the exercises below. Table 1 lists all pairs (l, k) for which the sequence $X_n = (X_{n-l} + X_{n-k}) \bmod 2$ has period length $2^k - 1$, when $k < 100$. The pairs (l, k) for small as well as larger k are shown, for the sake of completeness; the pair (1, 2) corresponds to the Fibonacci sequence mod 2, whose period has length 3. However, only pairs (l, k) for relatively large k should be used to generate random numbers in practice.

Instead of considering only additive sequences, we can construct useful random number generators by taking general linear combinations of X_{n-1}, \dots, X_{n-k} for small k . In this case the best results occur when the modulus m is a large prime; for example, m can be chosen to be the largest prime number that fits in a single computer word (see Table 4.5.4-1). When $m = p$ is prime, the theory of finite fields tells us that it is possible to find multipliers a_1, \dots, a_k such that the sequence defined by

$$X_n = (a_1 X_{n-1} + \dots + a_k X_{n-k}) \bmod p \quad (8)$$

has period length $p^k - 1$; here X_0, \dots, X_{k-1} may be chosen arbitrarily but not all zero. (The special case $k = 1$ corresponds to a multiplicative congruential sequence with prime modulus, with which we are already familiar.) The constants a_1, \dots, a_k in (8) have the desired property if and only if the polynomial

$$f(x) = x^k - a_1 x^{k-1} - \dots - a_k \quad (9)$$

is a “primitive polynomial modulo p ,” that is, if and only if this polynomial has a root that is a primitive element of the field with p^k elements (see exercise 4.6.2-16).

Of course, the mere fact that suitable constants a_1, \dots, a_k exist giving a period of length $p^k - 1$ is not enough for practical purposes; we must be able to *find* them, and we can't simply try all p^k possibilities, since p is on the order of the computer's word size. Fortunately there are exactly $\varphi(p^k - 1)/k$ suitable choices of (a_1, \dots, a_k) , so there is a fairly good chance of hitting one after making a few random tries. But we also need a way to tell quickly whether or not (9) is a primitive polynomial modulo p ; it is certainly unthinkable to generate up to $p^k - 1$ elements of the sequence and wait for a repetition! Methods of testing for primitivity modulo p are discussed by Alanen and Knuth in *Sankhyā* (A) 26 (1964), 305–328; the following criteria can be used: Let $r = (p^k - 1)/(p - 1)$.

- i) $(-1)^{k-1}a_k$ must be a primitive root modulo p . (Cf. Section 3.2.1.2.)
- ii) The polynomial x^r must be congruent to $(-1)^{k-1}a_k$, modulo $f(x)$ and p .
- iii) The degree of $x^{r/q} \bmod f(x)$, using polynomial arithmetic modulo p , must be positive, for each prime divisor q of r .

Efficient ways to compute the polynomial $x^n \bmod f(x)$, using polynomial arithmetic modulo a given prime p , are discussed in Section 4.6.2.

In order to carry out this test, we need to know the prime factorization of $r = (p^k - 1)/(p - 1)$, and this is the limiting factor in the calculation; r can be factored in a reasonable amount of time when $k = 2, 3$, and perhaps 4, but higher values of k are difficult to handle when p is large. Even $k = 2$ essentially doubles the number of “significant random digits” over what is achievable with $k = 1$, so larger values of k will rarely be necessary.

An adaptation of the spectral test (Section 3.3.4) can be used to rate the sequence of numbers generated by (8); see exercise 3.3.4–26. The considerations of that section show that we should *not* make the obvious choice of $a_1 = +1$ or -1 when it is possible to do so; it is better to pick large, essentially “random,” values of a_1, \dots, a_k that satisfy the conditions, and to verify the choice by applying the spectral test. A significant amount of computation is involved in finding a_1, \dots, a_k , but all known evidence indicates that the result will be a very satisfactory source of random numbers. We essentially achieve the randomness of a linear congruential generator with k -tuple precision, using only single precision operations.

The special case $p = 2$ is of independent interest. Sometimes a random number generator is desired that merely produces a random sequence of bits—zeros and ones—instead of fractions between zero and one. There is a simple way to generate a highly random bit sequence on a binary computer, manipulating k -bit words: Start with an arbitrary nonzero binary word X . To get the next random bit of the sequence, do the following operations, shown in MIX's language (see exercise 16):

```
LDA X      (Assume that overflow is now “off.”)
ADD X      Shift left one bit.
JNOV *+2   Jump if high bit was originally zero.          (10)
XOR A      Otherwise adjust number with “exclusive or.”
STA X      ■
```

The fourth instruction here is the “exclusive or” operation found on nearly all binary computers (cf. exercise 2.5–28 and Section 7.1); it changes each bit position in which location A has a “1” bit. The value in location A is the binary constant $(a_1 \dots a_k)_2$, where $x^k - a_1x^{k-1} - \dots - a_k$ is a primitive polynomial modulo 2 as above. After the code (10) has been executed, the next bit of the generated sequence may be taken as the least significant bit of word X (or, alternatively, we could consistently use the most significant bit of X, if it were more convenient to do so).

For example, consider Fig. 1, which illustrates the sequence generated for $k = 4$ and $\text{CONTENTS}(A) = (0011)_2$. This is, of course, an unusually small value for k . The right-hand column shows the sequence of bits of the sequence, namely $1101011110001001\dots$, repeating in a period of length $2^k - 1 = 15$. This sequence is quite random, considering that it was generated with only four bits of memory; to see this, consider the adjacent sets of four bits occurring in the period, namely 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000, 0001, 0010, 0100, 1001, 0011, 0110. In general, every possible adjacent set of k bits occurs exactly once in the period, except the set of all zeros, since the period length is $2^k - 1$; thus, adjacent sets of k bits are essentially independent. We shall see in Section 3.5 that this is a very strong criterion for randomness when k is, say, 30 or more. Theoretical results illustrating the randomness of this sequence are given in an article by R. C. Tausworthe, *Math. Comp.* **19** (1965), 201–209.

Primitive polynomials modulo 2 of degree ≤ 100 have been tabulated by E. J. Watson, *Math. Comp.* **16** (1962), 368–369. When $k = 35$, we may take

$$\text{CONTENTS}(A) = (00000000000000000000000000000000000000101)_2,$$

but the considerations of exercises 18 and 3.3.4–26 imply that it would be better to find “random” constants that define primitive polynomials modulo 2.

Caution: Several people have been trapped into believing that this random bit-generation technique can be used to generate random whole-word fractions $(.X_0X_1\dots X_{k-1})_2, (.X_kX_{k+1}\dots X_{2k-1})_2, \dots$; but it is actually a poor source of random fractions, even though the bits are individually quite random. Exercise 18 explains why.

Mitchell and Moore’s additive generator (7) is essentially based on the concept of primitive polynomials; the polynomial $x^{55} + x^{24} + 1$ is primitive, and Table 1 is essentially a listing of all the primitive trinomials modulo 2. A generator almost identical to that of Mitchell and Moore was independently discovered in 1971 by T. G. Lewis and W. H. Payne [cf. *JACM* **20** (1973), 456–468], but using “exclusive or” instead of addition so that the period is exactly $2^{55} - 1$; each bit position in their generated numbers runs through the same periodic sequence, but has its own starting point. (See Bright and Enison, *Computing Surveys* **11** (1979), 357–370, for further discussion of Lewis and Payne’s method.)

We have now seen that sequences with $0 \leq X_n < m$ and period $m^k - 1$ can be found, when X_n is a suitable function of X_{n-1}, \dots, X_{n-k} and when m is prime. The highest conceivable period for any sequence defined by a relation

```

1011
0101
1010
0111
1110
1111
1101
1001
0001
0010
0100
1000
0011
0110
1100
1011

```

Fig. 1. Successive contents of the computer word X in the binary method, assuming that $k = 4$ and $\text{CONTENTS}(A) = (0011)_2$.

of the form

$$X_n = f(X_{n-1}, \dots, X_{n-k}), \quad 0 \leq X_n < m, \quad (11)$$

is easily seen to be m^k . M. H. Martin [Bull. Amer. Math. Soc. **40** (1934), 859–864] was the first person to show that functions achieving this maximum period are possible for all m and k ; his method is easy to state, but it is unfortunately not suitable for programming (see exercise 17). From a computational standpoint, the simplest known functions f that yield the maximum period m^k appear in exercise 21; the corresponding programs are, in general, not as efficient for random number generation as other methods we have described, but they do give demonstrable randomness when the period as a whole is considered.

Another important class of techniques deals with the *combination* of random number generators, to get “more random” sequences. There will always be people who feel that the linear congruential methods, additive methods, etc., are all too simple to give sufficiently random sequences; and it may never be possible to prove that their skepticism is unjustified (although we believe it is), so it is pretty useless to argue the point. There are reasonably efficient methods for combining two sequences into a third one that should be haphazard enough to satisfy all but the most hardened skeptic.

Suppose we have two sequences X_0, X_1, \dots and Y_0, Y_1, \dots of random numbers between 0 and $m - 1$, preferably generated by two unrelated methods. One suggestion has been to add them together, mod m , obtaining the sequence $Z_n = (X_n + Y_n) \bmod m$; in this case, the period will be quite long if the period lengths of $\langle X_n \rangle$ and $\langle Y_n \rangle$ are relatively prime to each other (see exercise 13). Another approach, based on circular shifting and “exclusive or-ing”, has been suggested by W. J. Westlake, JACM **14** (1967), 337–340.

A considerably different method has been suggested by M. D. MacLaren and G. Marsaglia [JACM **12** (1965), 83–89; CACM **11** (1968), 759], who use one random sequence to permute the elements of another:

Algorithm M (*Randomizing by shuffling*). Given methods for generating two sequences $\langle X_n \rangle$ and $\langle Y_n \rangle$, this algorithm will successively output the terms of a “considerably more random” sequence. We use an auxiliary table $V[0], V[1], \dots, V[k - 1]$, where k is some number chosen for convenience, usually in the neighborhood of 100. Initially, the V -table is filled with the first k values of the X -sequence.

- M1. [Generate X, Y .] Set X and Y equal to the next members of the sequences $\langle X_n \rangle$ and $\langle Y_n \rangle$, respectively.
- M2. [Extract j .] Set $j \leftarrow \lfloor kY/m \rfloor$, where m is the modulus used in the sequence $\langle Y_n \rangle$; that is, j is a random value, $0 \leq j < k$, determined by Y .
- M3. [Exchange.] Output $V[j]$ and then set $V[j] \leftarrow X$. ■

As an example, assume that Algorithm M is applied to the following two sequences, with $k = 64$:

$$\begin{aligned} X_0 &= 5772156649, & X_{n+1} &= (3141592653X_n + 2718281829) \bmod 2^{35}; \\ Y_0 &= 1781072418, & Y_{n+1} &= (2718281829Y_n + 3141592653) \bmod 2^{35}. \end{aligned} \quad (12)$$

On intuitive grounds it appears safe to predict that the sequence obtained by applying Algorithm M will satisfy virtually anyone’s requirements for randomness in a computer-generated sequence, because the relationship between nearby terms of the output has been almost entirely obliterated. Furthermore, the time required to generate this sequence is only slightly more than twice as long as it takes to generate the sequence $\langle X_n \rangle$ alone.

Exercise 15 proves that the period length of Algorithm M’s output will be the least common multiple of the period lengths of $\langle X_n \rangle$ and $\langle Y_n \rangle$, in most situations of practical interest. In particular, the above example will have a period of length 2^{35} .

However, there is an even better way to shuffle the elements of a sequence, discovered by Carter Bays and S. D. Durham [ACM Trans. Math. Software 2 (1976), 59-64]. Their approach, although it appears to be superficially similar to Algorithm M, can give surprisingly better performance even though it requires only one input sequence $\langle X_n \rangle$ instead of two:

Algorithm B (*Randomizing by shuffling*). Given a method for generating a sequence $\langle X_n \rangle$, this algorithm will successively output the terms of a “considerably more random” sequence, using an auxiliary table $V[0], V[1], \dots, V[k - 1]$ as in Algorithm M. Initially the V -table is filled with the first k values of the X -sequence, and an auxiliary variable Y is set equal to the $(k + 1)$ st value.

- B1. [Extract j .] Set $j \leftarrow \lfloor kY/m \rfloor$, where m is the modulus used in the sequence $\langle X_n \rangle$; that is, j is a random value, $0 \leq j < k$, determined by Y .
- B2. [Exchange.] Set $Y \leftarrow V[j]$, output Y , and then set $V[j]$ to the next member of the sequence $\langle X_n \rangle$. ■

The reader is urged to work exercise 3, in order to get a feeling for the difference between Algorithms M and B.

On MIX we may implement Algorithm B by taking k equal to the byte size, obtaining the following simple generation scheme once the initialization has been done:

LD6 Y(1:1)	$j \leftarrow$ high-order byte of Y .
LDA X	$rA \leftarrow X_n$.
INCA 1	(cf. exercise 3.2.1.1-1)
MUL A	$rX \leftarrow X_{n+1}$.
STX X	$"n \leftarrow n + 1."$
LDA V, 6	
STA Y	$Y \leftarrow V[j]$.
STX V, 6	$V[j] \leftarrow X_n$. ■

(13)

The output appears in register A. Note that Algorithm B requires only four instructions of overhead per generated number.

F. Gebhardt [Math. Comp. 21 (1967), 708–709] found that satisfactory random sequences were produced by Algorithm M even when it was applied to a sequence as nonrandom as the Fibonacci sequence, with $X_n = F_{2n} \bmod m$ and $Y_n = F_{2n+1} \bmod m$. However, it is also possible for Algorithm M to produce a sequence less random than the original sequences, if $\langle X_n \rangle$ and $\langle Y_n \rangle$ are strongly related, as shown in exercise 3. Such problems do not seem to arise with Algorithm B. Since Algorithm B won't make a sequence any less random, and since it probably enhances the randomness substantially with very little extra cost, it can be recommended for use in combination with any other random number generator.

EXERCISES

- 1. [12] In practice, we form random numbers using $X_{n+1} = (aX_n + c) \bmod m$, where the X 's are integers, afterwards treating them as the fractions $U_n = X_n/m$. The recurrence relation for U_n is actually

$$U_{n+1} = (aU_n + c/m) \bmod 1.$$

Discuss the generation of random sequences using this relation *directly*, by making use of floating point arithmetic on the computer.

- 2. [M20] A good source of random numbers will have $X_{n-1} < X_{n+1} < X_n$ about one-sixth of the time, since each of the six possible relative orders of X_{n-1} , X_n , and X_{n+1} should be equally probable. However, show that the above ordering never occurs if the Fibonacci sequence (5) is used.

- 3. [23] (a) What sequence comes from Algorithm M if

$$X_0 = 0, \quad X_{n+1} = (5X_n + 3) \bmod 8, \quad Y_0 = 0, \quad Y_{n+1} = (5Y_n + 1) \bmod 8,$$

and $k = 4$? (Note that the potency is two, so $\langle X_n \rangle$ and $\langle Y_n \rangle$ aren't extremely random to start with.) (b) What happens if Algorithm B is applied to this same sequence $\langle X_n \rangle$ with $k = 4$?

4. [00] Why is the most significant byte used in the first line of program (13), instead of some other byte?
- 5. [20] Discuss using $X_n = Y_n$ in Algorithm M, in order to improve the speed of generation. Is the result analogous to Algorithm B?
6. [10] In the binary method (10), the text states that the low-order bit of X is random, if the code is performed repeatedly. Why isn't the entire word X random?
7. [20] Show that the full sequence of length 2^e (i.e., a sequence in which each of the 2^e possible sets of e adjacent bits occurs just once in the period) may be obtained if program (10) is changed to the following:

```

LDA  X
JANZ **2
LDA  A
ADD  X
JNOV **3
JAZ  **2
XOR  A
STA  X
■

```

8. [M39] Prove that the quadratic congruential sequence (3) has period length m if and only if the following conditions are satisfied:

- c is relatively prime to m ;
- d and $a - 1$ are both multiples of p , for all odd primes p dividing m ;
- d is even, and $d \equiv a - 1$ (modulo 4), if m is a multiple of 4;
 $d \equiv a - 1$ (modulo 2), if m is a multiple of 2;
- either $d \equiv 0$ or both $a \equiv 1$ and $cd \equiv 6$ (modulo 9), if m is a multiple of 9.

[Hint: The sequence defined by $X_0 = 0$, $X_{n+1} = dX_n^2 + aX_n + c$ has a period of length m , modulo m , only if its period length is r modulo any divisor r of m .]

- 9. [M24] (R. R. Coveyou.) Use the result of exercise 8 to prove that the modified middle-square method (4) has a period of length 2^{e-2} .
10. [M29] Show that if X_0 and X_1 are not both even and if $m = 2^e$, the period of the Fibonacci sequence (5) is $3 \cdot 2^{e-1}$.
11. [M36] The purpose of this exercise is to analyze certain properties of integer sequences satisfying the recurrence relation

$$X_n = a_1 X_{n-1} + \cdots + a_k X_{n-k}, \quad n \geq k;$$

if we can calculate the period length of this sequence modulo $m = p^e$, when p is prime, the period length with respect to an arbitrary modulus m is the least common multiple of the period lengths for the prime power factors of m .

- If $f(z)$, $a(z)$, $b(z)$ are polynomials with integer coefficients, let us write $a(z) \equiv b(z)$ (modulo $f(z)$ and m) if $a(z) = b(z) + f(z)u(z) + mv(z)$ for some polynomials $u(z)$, $v(z)$ with integer coefficients. Prove that when $f(0) = 1$ and $p^e > 2$, "If $z^\lambda \equiv 1$ (modulo $f(z)$ and p^e), $z^\lambda \not\equiv 1$ (modulo $f(z)$ and p^{e+1}), then $z^{p\lambda} \equiv 1$ (modulo $f(z)$ and p^{e+1}), $z^{p\lambda} \not\equiv 1$ (modulo $f(z)$ and p^{e+2})."
- Let $f(z) = 1 - a_1 z - \cdots - a_k z^k$, and let

$$G(z) = 1/f(z) = A_0 + A_1 z + A_2 z^2 + \cdots$$

Let $\lambda(m)$ denote the period length of $\langle A_n \bmod m \rangle$. Prove that $\lambda(m)$ is the smallest positive integer λ such that $z^\lambda \equiv 1$ (modulo $f(z)$ and m).

- c) Given that p is prime, $p^e > 2$, and $\lambda(p^e) \neq \lambda(p^{e+1})$, prove that $\lambda(p^{e+r}) = p^r \lambda(p^e)$ for all $r \geq 0$. (Thus, to find the period length of the sequence $\langle A_n \bmod 2^e \rangle$, we can compute $\lambda(4)$, $\lambda(8)$, $\lambda(16)$, ... until we find the smallest $e \geq 3$ such that $\lambda(2^e) \neq \lambda(4)$; then the period length is determined mod 2^e for all e . Exercise 4.6.3–26 explains how to calculate X_n for large n in $O(\log n)$ operations.)
- d) Show that any sequence of integers satisfying the recurrence stated at the beginning of this exercise has the generating function $g(z)/f(z)$, for some polynomial $g(z)$ with integer coefficients.
- e) Given that the polynomials $f(z)$ and $g(z)$ in part (d) are relatively prime modulo p (cf. Section 4.6.1), prove that the sequence $\langle X_n \bmod p^e \rangle$ has exactly the same period length as the special sequence $\langle A_n \bmod p^e \rangle$ in (b). (No longer period could be obtained by any choice of X_0, \dots, X_{k-1} , since the general sequence is a linear combination of “shifts” of the special sequence.) [Hint: By exercise 4.6.2–22 (Hensel’s lemma), there exist polynomials such that $a(z)f(z) + b(z)g(z) \equiv 1$ (modulo p^e).]

► 12. [M28] Find integers X_0, X_1, a, b , and c such that the sequence

$$X_{n+1} = (aX_n + bX_{n-1} + c) \bmod 2^e, \quad n \geq 1,$$

has the longest period length of all sequences of this type. [Hint: It follows that $X_{n+2} = ((a+1)X_{n+1} + (b-a)X_n - bX_{n-1}) \bmod 2^e$; see exercise 11(c).]

13. [M20] Let $\langle X_n \rangle$ and $\langle Y_n \rangle$ be sequences of integers mod m with periods of lengths λ_1 and λ_2 , and form the sequence $Z_n = (X_n + Y_n) \bmod m$. Show that if λ_1 and λ_2 are relatively prime, the sequence $\langle Z_n \rangle$ has a period of length $\lambda_1 \lambda_2$.

14. [M24] Let $X_n, Y_n, Z_n, \lambda_1, \lambda_2$ be as in the previous exercise. Suppose that the prime factorization of λ_1 is $2^{e_1} 3^{e_2} 5^{e_3} \dots$, and similarly suppose that $\lambda_2 = 2^{f_1} 3^{f_2} 5^{f_3} \dots$. Let $g_p = (\max(e_p, f_p) \text{ if } e_p \neq f_p, \text{ otherwise } 0)$, and let $\lambda_0 = 2^{g_2} 3^{g_3} 5^{g_5} \dots$. Show that the period length λ' of the sequence $\langle Z_n \rangle$ is a multiple of λ_0 , but it is a divisor of $\lambda = \text{lcm}(\lambda_1, \lambda_2)$. In particular, $\lambda' = \lambda$ if $(e_p \neq f_p \text{ or } e_p = f_p = 0)$ for each prime p .

15. [M27] Let the sequence $\langle X_n \rangle$ in Algorithm M have period length λ_1 , and assume that all elements of its period are distinct. Let $q_n = \min\{r \mid r > 0 \text{ and } \lfloor kY_{n-r}/m \rfloor = \lfloor kY_n/m \rfloor\}$. Assume that $q_n < \frac{1}{2}\lambda_1$ for all $n \geq n_0$, and that the sequence $\langle q_n \rangle$ has period length λ_2 . Let λ be the least common multiple of λ_1 and λ_2 . Prove that the output sequence $\langle Z_n \rangle$ produced by Algorithm M has a period of length λ .

► 16. [M28] Let CONTENTS(A) in method (10) be $(a_1 a_2 \dots a_k)_2$ in binary notation. Show that the generated sequence of low-order bits X_0, X_1, \dots satisfies the relation

$$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + \dots + a_k X_{n-k}) \bmod 2.$$

[This may be regarded as another way to define the sequence, although the connection between this relation and the efficient code (10) is not apparent at first glance!]

17. [M33] (M. H. Martin, 1934.) Let m and k be positive integers, and let $X_1 = X_2 = \dots = X_k = 0$. For all $n > 0$, set X_{n+k} equal to the largest nonnegative value $y < m$ such that the k -tuple $(X_{n+1}, \dots, X_{n+k-1}, y)$ has not already occurred in the sequence; in other words, $(X_{n+1}, \dots, X_{n+k-1}, y)$ must differ from $(X_{r+1}, \dots, X_{r+k})$ for $0 \leq r < n$. In this way, each possible k -tuple will occur at most once in the

sequence. Eventually the process will terminate, when we reach a value of n such that $(X_{n+1}, \dots, X_{n+k-1}, y)$ has already occurred in the sequence for all nonnegative $y < m$. For example, if $m = k = 3$ the sequence is 00022212202112102012001110100, and the process terminates at this point. (a) Prove that when the sequence terminates, we have $X_{n+1} = \dots = X_{n+k-1} = 0$. (b) Prove that every k -tuple (a_1, a_2, \dots, a_k) of elements with $0 \leq a_j < m$ occurs in the sequence; hence the sequence terminates when $n = m^k$. [Hint: Prove that the k -tuple $(a_1, \dots, a_s, 0, \dots, 0)$ appears, when $a_s \neq 0$, by induction on s .] Note that if we now define $f(X_n, \dots, X_{n+k-1}) = X_{n+k}$ for $1 \leq n \leq m^k$, setting $X_{m^k+k} = 0$, we obtain a function of maximum possible period.

18. [M22] Let $\langle X_n \rangle$ be the sequence of bits generated by method (10), with $k = 35$ and $\text{CONTENTS}(A) = (00000000000000000000000000000000000000101)_2$. Let U_n be the binary fraction $(.X_{nk}X_{nk+1}\dots X_{nk+k-1})_2$; show that this sequence $\langle U_n \rangle$ fails the serial test on pairs (Section 3.3.2B) when $d = 8$.

19. [M41] For each prime p specified in the first column of Table 1 in Section 4.5.4, find suitable constants a_1, a_2 as suggested in the text, such that the period length of (8), when $k = 2$, is $p^2 - 1$. (See Eq. 3.3.4-39 for an example.)

20. [M40] Calculate constants suitable for use as $\text{CONTENTS}(A)$ in method (10), having approximately the same number of zeros as ones, for $2 \leq k \leq 64$.

21. [M35] (D. Rees.) The text explains how to find functions f such that the sequence (11) has period length $m^k - 1$, provided that m is prime and X_0, \dots, X_{k-1} are not all zero. Show that such functions can be modified to obtain sequences of type (11) with period length m^k , for all integers m . [Hints: Consider Lemma 3.2.1.2Q, the trick of exercise 7, and sequences such as $\langle pX_{2n} + X_{2n+1} \rangle$.]

► 22. [M24] The text restricts discussion of the extended linear sequences (8) to the case that m is prime. Prove that reasonably long periods can also be obtained when m is “square-free,” i.e., the product of distinct primes. (Examination of Table 3.2.1.1-1 shows that $m = w \pm 1$ often satisfies this hypothesis; many of the results of the text can therefore be carried over to that case, which is somewhat more convenient for calculation.)

► 23. [20] Discuss the sequence defined by $X_n = (X_{n-55} - X_{n-24}) \bmod m$ as an alternative to (7).

24. [M20] Let $0 < k < m$. Prove that the sequence of bits defined by the recurrence $X_n = (X_{n-m+k} + X_{n-m}) \bmod 2$ has period length $2^m - 1$ whenever the sequence defined by $Y_n = (Y_{n-k} + Y_{n-m}) \bmod 2$ does.

25. [26] Discuss the alternative to Program A that changes all 55 entries of the Y table every 55th time a random number is required.

26. [M48] (J. F. Reiser.) Let p be prime and let k be a positive integer. Given integers a_1, \dots, a_k and x_1, \dots, x_k , let λ_α be the period of the sequence $\langle X_n \rangle$ generated by the recurrence

$$X_n = x_n \bmod p^\alpha, \quad 0 \leq n < k; \quad X_n = (a_1 X_{n-1} + \dots + a_k X_{n-k}) \bmod p^\alpha, \quad n \geq k;$$

and let N_α be the number of 0's that occur in the period (i.e., the number of indices j such that $\mu_\alpha \leq j < \mu_\alpha + \lambda_\alpha$ and $X_j = 0$). Prove or disprove the following conjecture: There exists a constant c (depending possibly on p and k and a_1, \dots, a_k) such that $N_\alpha \leq c p^{\alpha(k-2)/(k-1)}$ for all α and all x_1, \dots, x_k .

[Notes: Reiser has proved that if the recurrence has maximum period length mod p (i.e., if $\lambda_1 = p^k - 1$), and if the conjecture holds, then the k -dimensional discrepancy of $\langle X_n \rangle$ will be $O(\alpha^k p^{-\alpha/(k-1)})$ as $\alpha \rightarrow \infty$; thus an additive generator like (7) would be well distributed in 55 dimensions, when $m = 2^e$ and the entire period is considered. (See Section 3.3.4 for the definition of discrepancy in k dimensions.) The conjecture is a very weak condition, for if $\langle X_n \rangle$ takes on each value about equally often and if $\lambda_\alpha = p^{\alpha-1}(p^k - 1)$, the quantity $N_\alpha \approx (p^k - 1)/p$ does not grow at all as α increases. Reiser has verified the conjecture for $k = 3$. On the other hand he has shown that it is possible to find unusually bad starting values x_1, \dots, x_k (depending on α) so that $N_{2\alpha} \geq p^\alpha$, provided that $\lambda_\alpha = p^{\alpha-1}(p^k - 1)$ and $k \geq 3$ and α is sufficiently large.]

27. [M30] Suppose Algorithm B is being applied to a sequence $\langle X_n \rangle$ whose period length is λ , where $\lambda \gg k$. Show that for fixed k and all sufficiently large λ , the output of the sequence will eventually be periodic with the same period length λ , unless $\langle X_n \rangle$ isn't very random to start with. [Hint: Find a pattern of consecutive values of $\lfloor kX_n/m \rfloor$ that causes Algorithm B to "synchronize" its subsequent behavior.]

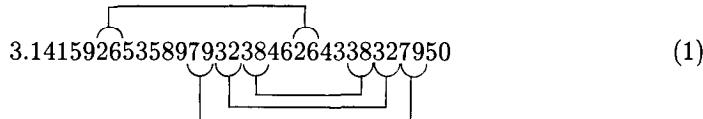
28. [40] (A. G. Waterman.) Experiment with linear congruential sequences with m the square or cube of the computer word size, while a and c are single-precision numbers.

3.3. STATISTICAL TESTS

OUR MAIN PURPOSE is to obtain sequences that behave as if they are random. So far we have seen how to make the period of a sequence so long that for practical purposes it never will repeat; this is an important criterion, but it by no means guarantees that the sequence will be useful in applications. How then are we to decide whether a sequence is sufficiently random?

If we were to give some man a pencil and paper and ask him to write down 100 random decimal digits, chances are very slim that he will give a satisfactory result. People tend to avoid things that seem nonrandom, such as pairs of equal adjacent digits (although about one out of every 10 digits should equal its predecessor). And if we would show someone a table of truly random digits, he would quite probably tell us they are not random at all; his eye would spot certain apparent regularities.

According to Dr. I. J. Matrix (as quoted by Martin Gardner in *Scientific American*, January, 1965), "Mathematicians consider the decimal expansion of π a random series, but to a modern numerologist it is rich with remarkable patterns." Dr. Matrix has pointed out, for example, that the first repeated two-digit number in π 's expansion is 26, and its second appearance comes in the middle of a curious repetition pattern:



After listing a dozen or so further properties of these digits, he observed that π , when correctly interpreted, conveys the entire history of the human race!

We all notice patterns in our telephone numbers, license numbers, etc., as aids to memory. The point of these remarks is that we cannot be trusted to judge by ourselves whether a sequence of numbers is random or not. Some unbiased mechanical tests must be applied.

The theory of statistics provides us with some quantitative measures for randomness. There is literally no end to the number of tests that can be conceived; we will discuss those tests that have proved to be most useful, most instructive, and most readily adapted to computer calculation.

If a sequence behaves randomly with respect to tests T_1, T_2, \dots, T_n , we cannot be sure in general that it will not be a miserable failure when it is subjected to a further test T_{n+1} ; yet each test gives us more and more confidence in the randomness of the sequence. In practice, we apply about half a dozen different kinds of statistical tests to a sequence, and if it passes these satisfactorily we consider it to be random—it is then presumed innocent until proven guilty.

Every sequence that is to be used extensively should be tested carefully, so the following sections explain how to carry out these tests in the right way. Two kinds of tests are distinguished: *empirical* tests, for which the computer manipulates groups of numbers of the sequence and evaluates certain statistics; and *theoretical* tests, for which we establish characteristics of the sequence by

using number-theoretic methods based on the recurrence rule used to form the sequence.

If the evidence doesn't come out as desired, the reader may wish to try the techniques in *How to Lie With Statistics* by Darrell Huff (Norton, 1954).

3.3.1. General Test Procedures for Studying Random Data

A. "Chi-square" tests. The chi-square test (χ^2 test) is perhaps the best known of all statistical tests, and it is a basic method that is used in connection with many other tests. Before considering the method in general, let us consider a particular example of the chi-square test as it might be applied to dice throwing. Using two "true" dice (each of which, independently, is assumed to register 1, 2, 3, 4, 5, or 6 with equal probability), the following table gives the probability of obtaining a given total, s , on a single throw:

value of $s =$	2	3	4	5	6	7	8	9	10	11	12	(1)
probability, $p_s =$	$\frac{1}{36}$	$\frac{1}{18}$	$\frac{1}{12}$	$\frac{1}{9}$	$\frac{5}{36}$	$\frac{1}{6}$	$\frac{5}{36}$	$\frac{1}{9}$	$\frac{1}{12}$	$\frac{1}{18}$	$\frac{1}{36}$	

(For example, a value of 4 can be thrown in three ways: 1 + 3, 2 + 2, 3 + 1; this constitutes $\frac{3}{36} = \frac{1}{12} = p_4$ of the 36 possible outcomes.)

If we throw the dice n times, we should obtain the value s approximately np_s times on the average. For example, in 144 throws we should get the value 4 about 12 times. The following table shows what results were actually obtained in a particular sequence of 144 throws of the dice:

value of $s =$	2	3	4	5	6	7	8	9	10	11	12	
observed number, $Y_s =$	2	4	10	12	22	29	21	15	14	9	6	(2)
expected number, $np_s =$	4	8	12	16	20	24	20	16	12	8	4	

Note that the observed number is different from the expected number in all cases; in fact, random throws of the dice will hardly ever come out with exactly the right frequencies. There are 36^{144} possible sequences of 144 throws, all of which are equally likely. One of these sequences consists of all 2's ("snake eyes"), and anyone throwing 144 snake eyes in a row would be convinced that the dice were loaded. Yet the sequence of all 2's is just as probable as any other particular sequence if we specify the outcome of each throw of each die.

In view of this, how can we test whether or not a given pair of dice is loaded? The answer is that we can't make a definite yes-no statement, but we can give a probabilistic answer. We can say how probable or improbable certain types of events are.

A fairly natural way to proceed in the above example is to consider the squares of the differences between the observed numbers Y_s and the expected numbers np_s . We can add these together, obtaining

$$V = (Y_2 - np_2)^2 + (Y_3 - np_3)^2 + \cdots + (Y_{12} - np_{12})^2. \quad (3)$$

A bad set of dice should result in a relatively high value of V ; and for any given value of V we can ask, "What is the probability that V is this high, using true dice?" If this probability is very small, say $\frac{1}{100}$, we would know that only about one time in 100 would true dice give results so far away from the expected numbers, and we would have definite grounds for suspicion. (Remember, however, that even good dice would give such a high value of V about one time in a hundred, so a cautious person would repeat the experiment to see if the high value of V is repeated.)

The statistic V in (3) gives equal weight to $(Y_7 - np_7)^2$ and $(Y_2 - np_2)^2$, although $(Y_7 - np_7)^2$ is likely to be a good deal higher than $(Y_2 - np_2)^2$ since 7's occur about six times as often as 2's. It turns out that the "right" statistic, at least one that has proved to be most important, will give $(Y_7 - np_7)^2$ only $\frac{1}{6}$ as much weight as $(Y_2 - np_2)^2$, and we should change (3) to the following formula:

$$V = \frac{(Y_2 - np_2)^2}{np_2} + \frac{(Y_3 - np_3)^2}{np_3} + \cdots + \frac{(Y_{12} - np_{12})^2}{np_{12}}. \quad (4)$$

This is called the "chi-square" statistic of the observed quantities Y_2, \dots, Y_{12} in this dice-throwing experiment. For the data in (2), we find that

$$V = \frac{(2-4)^2}{4} + \frac{(4-8)^2}{8} + \cdots + \frac{(9-8)^2}{8} + \frac{(6-4)^2}{4} = 7\frac{7}{48}. \quad (5)$$

The important question now is, of course, "does $7\frac{7}{48}$ constitute an improbably high value for V to assume?" Before answering this question, let us consider the general application of the chi-square method.

In general, suppose that every observation can fall into one of k categories. We take n independent observations; this means that the outcome of one observation has absolutely no effect on the outcome of any of the others. Let p_s be the probability that each observation falls into category s , and let Y_s be the number of observations that actually do fall into category s . We form the statistic

$$V = \sum_{1 \leq s \leq k} \frac{(Y_s - np_s)^2}{np_s}. \quad (6)$$

In our example above, there are eleven possible outcomes of each throw of the dice, so $k = 11$. (Eq. (6) is a slight change of notation from Eq. (4), since we are numbering the possibilities from 1 to k instead of from 2 to 12.)

By expanding $(Y_s - np_s)^2 = Y_s^2 - 2np_s Y_s + n^2 p_s^2$ in (6), and using the facts that

$$\begin{aligned} Y_1 + Y_2 + \cdots + Y_k &= n, \\ p_1 + p_2 + \cdots + p_k &= 1, \end{aligned} \quad (7)$$

we arrive at the formula

$$V = \frac{1}{n} \sum_{1 \leq s \leq k} \left(\frac{Y_s^2}{p_s} \right) - n, \quad (8)$$

which often makes the computation of V somewhat easier.

Table 1

SELECTED PERCENTAGE POINTS OF THE CHI-SQUARE DISTRIBUTION

	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
$\nu = 1$	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
$\nu = 2$	0.02010	0.1026	0.5753	1.386	2.773	5.991	9.210
$\nu = 3$	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
$\nu = 4$	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
$\nu = 5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
$\nu = 6$	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
$\nu = 7$	1.239	2.167	4.255	6.346	9.037	14.07	18.48
$\nu = 8$	1.646	2.733	5.071	7.344	10.22	15.51	20.09
$\nu = 9$	2.088	3.325	5.899	8.343	11.39	16.92	21.67
$\nu = 10$	2.558	3.940	6.737	9.342	12.55	18.31	23.21
$\nu = 11$	3.053	4.575	7.584	10.34	13.70	19.68	24.73
$\nu = 12$	3.571	5.226	8.438	11.34	14.84	21.03	26.22
$\nu = 15$	5.229	7.261	11.04	14.34	18.25	25.00	30.58
$\nu = 20$	8.260	10.85	15.45	19.34	23.83	31.41	37.57
$\nu = 30$	14.95	18.49	24.48	29.34	34.80	43.77	50.89
$\nu = 50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15
$\nu > 30$	$\nu + \sqrt{2\nu}x_p + \frac{2}{3}x_p^2 - \frac{2}{3} + O(1/\sqrt{\nu})$						
$x_p =$	-2.33	-1.64	-0.675	0.00	0.675	1.64	2.33

(For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U.S. Government Printing Office, 1964), Table 26.8.)

Now we turn to the important question, what constitutes a reasonable value of V ? This is found by referring to a table such as Table 1, which gives values of "the chi-square distribution with ν degrees of freedom" for various values of ν . The line of the table with $\nu = k - 1$ is to be used; the number of "degrees of freedom" is $k - 1$, one less than the number of categories. (Intuitively, this means that Y_1, Y_2, \dots, Y_k are not completely independent, since Eq. (7) shows that Y_1 can be computed if Y_2, \dots, Y_k are known; hence, $k - 1$ degrees of freedom are present. This argument is not rigorous, but the theory below justifies it.)

If the table entry in row ν under column p is x , it means, "The quantity V in Eq. (8) will be less than or equal to x with approximate probability p , if n is large enough." For example, the 95 percent entry in row 10 is 18.31; this says we will have $V > 18.31$ only about 5 percent of the time.

Let us assume that the above dice-throwing experiment is simulated on a computer using some sequence of supposedly random numbers, with the following results:

value of $s =$	2	3	4	5	6	7	8	9	10	11	12
Experiment 1, $Y_s =$	4	10	10	13	20	18	18	11	13	14	13
Experiment 2, $Y_s =$	3	7	11	15	19	24	21	17	13	9	5

We can compute the chi-square statistic in the first case, getting the value $V_1 = 29 \frac{59}{120}$, and in the second case we get $V_2 = 1 \frac{17}{120}$. Referring to the table entries for 10 degrees of freedom, we see that V_1 is *much too high*; V will be greater than 23.21 only about one percent of the time! (By using more extensive tables, we find in fact that V will be as high as V_1 only 0.1 percent of the time.) Therefore Experiment 1 represents a significant departure from random behavior.

On the other hand, V_2 is quite low, since the observed values Y_s in Experiment 2 are quite close to the expected values np_s in (2). The chi-square table tells us, in fact, that V_2 is *much too low*: the observed values are so close to the expected values, we cannot consider the result to be random! (Indeed, reference to other tables shows that such a low value of V occurs only 0.03 percent of the time when there are 10 degrees of freedom.) Finally, the value $V = 7 \frac{7}{48}$ computed in (5) can also be checked with Table 1. It falls between the entries for 25 percent and 50 percent, so we cannot consider it to be significantly high or significantly low; thus the observations in (2) are satisfactorily random with respect to this test.

It is somewhat remarkable that the same table entries are used no matter what the value of n is, and no matter what the probabilities p_s are. Only the number $\nu = k - 1$ affects the results. In actual fact, however, the table entries are not exactly correct: the chi-square distribution is an approximation that is valid only for large enough values of n . How large should n be? A common rule of thumb is to take n large enough so that each of the expected values np_s is five or more; preferably, however, take n much larger than this, to get a more powerful test. Note that in our examples above we took $n = 144$, so np_2 was only 4, violating the stated "rule of thumb." This was done only because the author tired of throwing the dice; it makes the entries in Table 1 less accurate for our application. Experiments run on a computer, with $n = 1000$, or 10000, or even 100000, would be much better than this. We could also combine the data for $s = 2$ and $s = 12$; then the test would have only nine degrees of freedom but the chi-square approximation would be more accurate.

We can get an idea of how crude an approximation is involved by considering the case when there are only two categories, having respective probabilities p_1 and p_2 . Suppose $p_1 = \frac{1}{4}$ and $p_2 = \frac{3}{4}$. According to the stated rule of thumb, we should have $n \geq 20$ to have a satisfactory approximation, so let's check this out. When $n = 20$, the possible values of V are $\frac{4}{15}r^2$ for $-5 \leq r \leq 15$; we wish to know how well the $\nu = 1$ row of Table 1 describes the distribution of V . The chi-square distribution varies continuously, while the actual distribution

of V has rather big jumps, so we need some convention for representing the exact distribution. If the distinct possible outcomes of the experiment lead to the values $V_0 \leq V_1 \leq \dots \leq V_n$ with respective probabilities $\pi_0, \pi_1, \dots, \pi_n$, suppose that a given percentage p falls in the range $\pi_0 + \dots + \pi_{j-1} < p \leq \pi_0 + \dots + \pi_{j-1} + \pi_j$. We would like to represent p by a "percentage point" x such that V is less than x with probability $\leq p$ and V is greater than x with probability $\leq 1 - p$. It is not difficult to see that the only such number is $x = V_j$. In our example for $n = 20$ and $\nu = 1$, it turns out that the percentage points of the exact distribution, corresponding to the approximations in Table 1 for $p = 1\%, 5\%, 25\%, 50\%, 75\%, 95\%$, and 99% , respectively, are

$$0, \quad 0, \quad .27, \quad .27, \quad 1.07, \quad 4.27, \quad 6.67.$$

For example, the percentage point for $p = 95\%$ is 4.27, while Table 1 gives the estimate 3.841. The latter value is too low; it tells us (incorrectly) to reject the value $V = 4.27$ at the 95% level, while in fact the probability that $V \geq 4.27$ is more than 6.5%. When $n = 21$, the situation changes slightly because the expected values $np_1 = 5.25$ and $np_2 = 15.75$ can never be obtained exactly; the percentage points for $n = 21$ are

$$.02, \quad .02, \quad .14, \quad .40, \quad 1.29, \quad 3.57, \quad 5.73.$$

We would expect Table 1 to be a better approximation when $n = 50$, but the corresponding tableau actually turns out to be further from Table 1 in some respects than it was for $n = 20$:

$$.03, \quad .03, \quad .03, \quad .67, \quad 1.31, \quad 3.23, \quad 6.0.$$

Here are the values when $n = 300$:

$$0, \quad 0, \quad .07, \quad .44, \quad 1.44, \quad 4.0, \quad 6.42.$$

Even in this case, when np_s is ≥ 75 in each category, the entries in Table 1 are good to only about one significant digit.

The proper choice of n is somewhat obscure. If the dice are actually biased, the fact will be detected as n gets larger and larger. (Cf. exercise 12.) But large values of n will tend to smooth out *locally* nonrandom behavior, i.e., blocks of numbers with a strong bias followed by blocks of numbers with the opposite bias. This type of behavior would not happen when actual dice are rolled, since the same dice are used throughout the test, but a sequence of numbers generated on a computer might very well display such locally nonrandom behavior. Perhaps a chi-square test should be made for a number of different values of n . At any rate, n should always be rather large.

We can summarize the chi-square test as follows. A fairly large number, n , of independent observations is made. (It is important to avoid using the chi-square method unless the observations are independent. See, for example,

A	B	C	D	E	F
○		○		○ ○	● ●
	○ ●	○		○ ○	● ● ●
	○		● ○		● ● ●
○				○	● ● ●
Range of V	Indication	Code			
0–1 percent, 99–100 percent	Reject	●			
1–5 percent, 95–99 percent	Suspect	●○			
5–10 percent, 90–95 percent	Almost suspect	○			

Fig. 2. Indications of “significant” deviations in 90 chi-square tests (cf. also Fig. 5).

exercise 10, which considers the case when half of the observations depend on the other half.) We count the number of observations falling into each of k categories and compute the quantity V given in Eqs. (6) and (8). Then V is compared with the numbers in Table 1, with $\nu = k - 1$. If V is less than the 1% entry or greater than the 99% entry, we reject the numbers as not sufficiently random. If V lies between the 1% and 5% entries or between the 95% and 99% entries, the numbers are “suspect”; if (by interpolation in the table) V lies between the 5% and 10% entries, or the 90% and 95% entries, the numbers might be “almost suspect.” The chi-square test is often done at least three times on different sets of data, and if at least two of the three results are suspect the numbers are regarded as not sufficiently random.

For example, see Fig. 2, which shows schematically the results of applying five different types of chi-square tests on each of six sequences of random numbers. Each test has been applied to three different blocks of numbers of the sequence. Generator A is the MacLaren–Marsaglia method (Algorithm 3.2.2M applied to the sequences in 3.2.2–12), Generator E is the Fibonacci method, and the other generators are linear congruential sequences with the following parameters:

Generator B: $X_0 = 0$, $a = 3141592653$, $c = 2718281829$, $m = 2^{35}$.

Generator C: $X_0 = 0$, $a = 2^7 + 1$, $c = 1$, $m = 2^{35}$.

Generator D: $X_0 = 47594118$, $a = 23$, $c = 0$, $m = 10^8 + 1$.

Generator F: $X_0 = 314159265$, $a = 2^{18} + 1$, $c = 1$, $m = 2^{35}$.

From Fig. 2 we conclude that (so far as these tests are concerned) Generators A, B, D are satisfactory, Generator C is on the borderline and should probably be rejected, Generators E and F are definitely unsatisfactory. Generator F has, of course, low potency; Generators C and D have been discussed in the

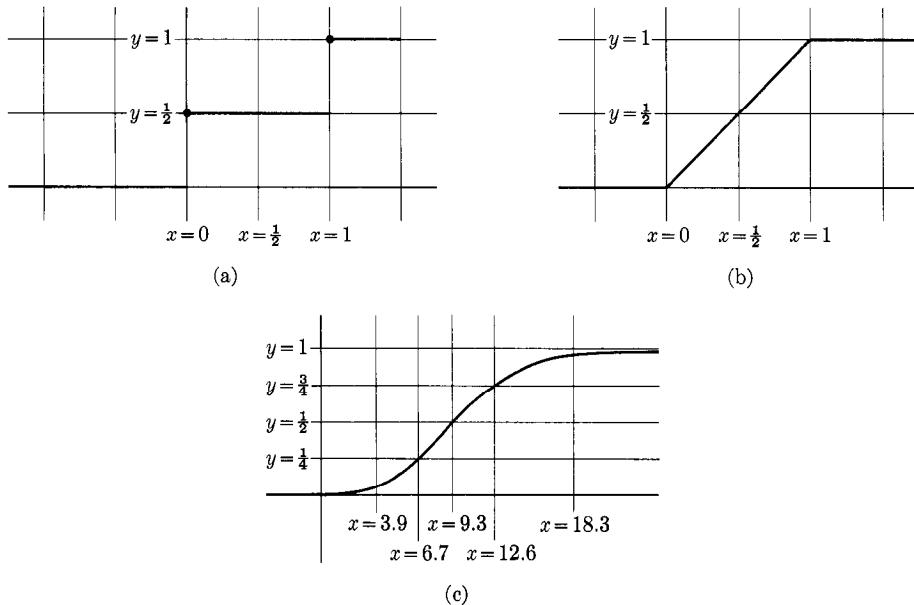


Fig. 3. Examples of distribution functions.

literature, but their multipliers are too small. (Generator D is the original multiplicative generator proposed by Lehmer in 1948; Generator C is the original linear congruential generator with $c \neq 0$ proposed by Rotenberg in 1960.)

Instead of using the “suspect,” “almost suspect,” etc., criteria for judging the results of chi-square tests, there is a less *ad hoc* procedure available, which will be discussed later in this section.

B. The Kolmogorov-Smirnov test. As we have seen, the chi-square test applies to the situation when observations can fall into a finite number of categories. It is not unusual, however, to consider random quantities that may assume infinitely many values. For example, a random real number between 0 and 1 may take on infinitely many values; even though only a finite number of these can be represented in the computer, we want our random values to behave essentially as though they are random real numbers.

A general notation for specifying probability distributions, whether they are finite or infinite, is commonly used in the study of probability and statistics. Suppose we want to specify the distribution of the values of a random quantity, X ; we do this in terms of the *distribution function* $F(x)$, where

$$F(x) = \text{probability that } (X \leq x).$$

Three examples are shown in Fig. 3. First we see the distribution function for a *random bit*, i.e., for the case when X takes on only the two values 0 and 1, each with probability $\frac{1}{2}$. Part (b) of the figure shows the distribution function

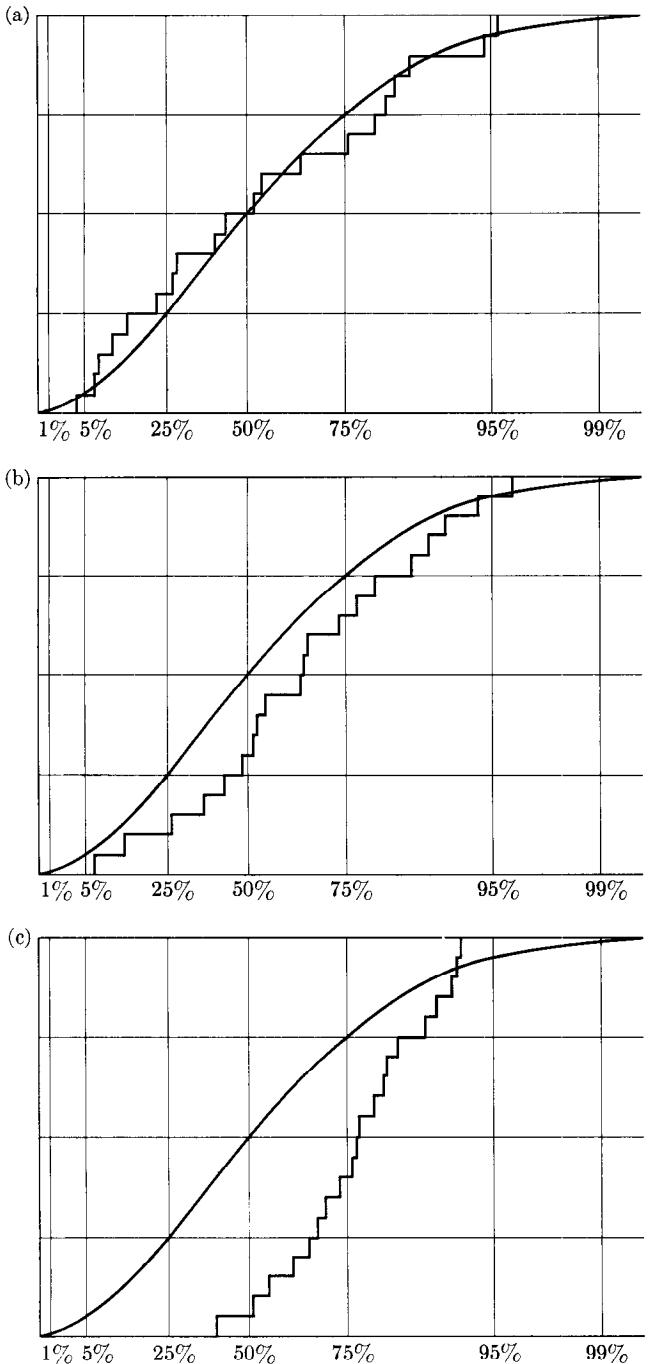


Fig. 4. Examples of empirical distributions.

for a uniformly distributed random real number between zero and one, so the probability that $X \leq x$ is simply equal to x when $0 \leq x \leq 1$; for example, the probability that $X \leq \frac{2}{3}$ is, naturally, $\frac{2}{3}$. And part (c) shows the limiting distribution of the value V in the chi-square test (shown here with 10 degrees of freedom); this is a distribution that we have already seen represented in another way in Table 1. Note that $F(x)$ always increases from 0 to 1 as x increases from $-\infty$ to $+\infty$.

If we make n independent observations of the random quantity X , thereby obtaining the values X_1, X_2, \dots, X_n , we can form the empirical distribution function $F_n(x)$, where

$$F_n(x) = \frac{\text{number of } X_1, X_2, \dots, X_n \text{ that are } \leq x}{n}. \quad (10)$$

Figure 4 illustrates three empirical distribution functions (shown as zigzag lines, although strictly speaking the vertical lines are not part of the graph of $F_n(x)$), superimposed on a graph of the assumed actual distribution function $F(x)$. As n gets large, $F_n(x)$ should be a better and better approximation to $F(x)$.

The Kolmogorov-Smirnov test (KS test) may be used when $F(x)$ has no jumps. It is based on the difference between $F(x)$ and $F_n(x)$. A bad source of random numbers will give empirical distribution functions that do not approximate $F(x)$ sufficiently well. Figure 4(b) shows an example in which the X_i are consistently too high, so the empirical distribution function is too low. Part (c) of the figure shows an even worse example; it is plain that such great deviations between $F_n(x)$ and $F(x)$ are extremely improbable, and the KS test is used to tell us how improbable they are.

To make the test, we form the following statistics:

$$\begin{aligned} K_n^+ &= \sqrt{n} \max_{-\infty < x < +\infty} (F_n(x) - F(x)); \\ K_n^- &= \sqrt{n} \max_{-\infty < x < +\infty} (F(x) - F_n(x)). \end{aligned} \quad (11)$$

Here K_n^+ measures the greatest amount of deviation when F_n is greater than F , and K_n^- measures the maximum deviation when F_n is less than F . The statistics for the examples of Fig. 4 are

	Part (a)	Part (b)	Part (c)	
K_{20}^+	0.492	0.134	0.313	
K_{20}^-	0.536	1.027	2.101	

(Note: The factor \sqrt{n} that appears in Eqs. (11) may seem puzzling at first. Exercise 6 shows that, for fixed x , the standard deviation of $F_n(x)$ is proportional to $1/\sqrt{n}$; hence the factor \sqrt{n} magnifies the statistics K_n^+, K_n^- in such a way that this standard deviation is independent of n .)

Table 2SELECTED PERCENTAGE POINTS OF THE DISTRIBUTIONS K_n^+ AND K_n^-

	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
$n = 1$	0.01000	0.05000	0.2500	0.5000	0.7500	0.9500	0.9900
$n = 2$	0.01400	0.06749	0.2929	0.5176	0.7071	1.0980	1.2728
$n = 3$	0.01699	0.07919	0.3112	0.5147	0.7539	1.1017	1.3589
$n = 4$	0.01943	0.08789	0.3202	0.5110	0.7642	1.1304	1.3777
$n = 5$	0.02152	0.09471	0.3249	0.5245	0.7674	1.1392	1.4024
$n = 6$	0.02336	0.1002	0.3272	0.5319	0.7703	1.1463	1.4144
$n = 7$	0.02501	0.1048	0.3280	0.5364	0.7755	1.1537	1.4246
$n = 8$	0.02650	0.1086	0.3280	0.5392	0.7797	1.1586	1.4327
$n = 9$	0.02786	0.1119	0.3274	0.5411	0.7825	1.1624	1.4388
$n = 10$	0.02912	0.1147	0.3297	0.5426	0.7845	1.1658	1.4440
$n = 11$	0.03028	0.1172	0.3330	0.5439	0.7863	1.1688	1.4484
$n = 12$	0.03137	0.1193	0.3357	0.5453	0.7880	1.1714	1.4521
$n = 15$	0.03424	0.1244	0.3412	0.5500	0.7926	1.1773	1.4606
$n = 20$	0.03807	0.1298	0.3461	0.5547	0.7975	1.1839	1.4698
$n = 30$	0.04354	0.1351	0.3509	0.5605	0.8036	1.1916	1.4801
$n > 30$	$y_p - 1/(6\sqrt{n}) + O(1/n)$, where $y_p^2 = \frac{1}{2} \ln(1/(1-p))$						
$y_p =$	0.07089	0.1601	0.3793	0.5887	0.8326	1.2239	1.5174

As in the chi-square test, we may now look up the values K_n^+ , K_n^- in a "percentile" table to determine if they are significantly high or low. Table 2 may be used for this purpose, both for K_n^+ and K_n^- . For example, the probability is 75 percent that K_{20}^- will be 0.7975 or less. Unlike the chi-square test, the table entries are not merely approximations that hold for large values of n ; Table 2 gives exact values (except, of course, for roundoff error), and the KS test may be reliably used for any value of n .

As they stand, formulas (11) are not readily adapted to computer calculation, since we are asking for a maximum over infinitely many values of x . From the fact that $F(x)$ is increasing and the fact that $F_n(x)$ increases only in finite steps, however, we can derive a simple procedure for evaluating the statistics K_n^+ and K_n^- :

Step 1. Obtain the observations X_1, X_2, \dots, X_n .

Step 2. Rearrange the observations so that they are sorted into ascending order, i.e., so that $X_1 \leq X_2 \leq \dots \leq X_n$. (Efficient sorting algorithms are the subject

of Chapter 5. On the other hand, it is possible to avoid sorting in this particular application, as shown in exercise 23.)

Step 3. The desired statistics are now given by the formulas

$$\begin{aligned} K_n^+ &= \sqrt{n} \max_{1 \leq j \leq n} \left(\frac{j}{n} - F(X_j) \right); \\ K_n^- &= \sqrt{n} \max_{1 \leq j \leq n} \left(F(X_j) - \frac{j-1}{n} \right). \end{aligned} \quad (13)$$

An appropriate choice of the number of observations, n , is slightly easier to make for this test than it is for the χ^2 test, although some of the considerations are similar. If the random variables X_j actually belong to the probability distribution $G(x)$, while they were assumed to belong to the distribution given by $F(x)$, it will take a comparatively large value of n to reject the hypothesis that $G(x) = F(x)$; for we need n large enough that the empirical distributions $G_n(x)$ and $F_n(x)$ are expected to be observably different. On the other hand, large values of n will tend to average out locally nonrandom behavior, and such behavior is an undesirable characteristic that is of significant importance in most computer applications of random numbers; this makes a case for smaller values of n . A good compromise would be to take n equal to, say, 1000, and to make a fairly large number of calculations of K_{1000}^+ on different parts of a random sequence, thereby obtaining values

$$K_{1000}^+(1), \quad K_{1000}^+(2), \quad \dots, \quad K_{1000}^+(r). \quad (14)$$

We can also apply the KS test again to these results: Let $F(x)$ now be the distribution function for K_{1000}^+ , and determine the empirical distribution $F_r(x)$ obtained from the observed values in (14). Fortunately, the function $F(x)$ in this case is very simple; for a large value of n like $n = 1000$, the distribution of K_n^+ is closely approximated by

$$F_\infty(x) = 1 - e^{-2x^2}, \quad x \geq 0. \quad (15)$$

The same remarks apply to K_n^- , since K_n^+ and K_n^- have the same expected behavior. This method of using several tests for moderately large n , then combining the observations later in another KS test, will tend to detect both local and global nonrandom behavior.

An experiment of this type (although on a much smaller scale) was made by the author as this chapter was being written. The “maximum of 5” test described in the next section was applied to a set of 1000 uniform random numbers, yielding 200 observations X_1, X_2, \dots, X_{200} that were supposed to belong to the distribution $F(x) = x^5$ ($0 \leq x \leq 1$). The observations were divided into 20 groups of 10 each, and the statistic K_{10}^+ was computed for each group. The 20 values of K_{10}^+ , thus obtained, led to the empirical distributions shown in Fig. 4. The smooth curve shown in each of the diagrams in Fig. 4

is the actual distribution the statistic K_{10}^+ should have. Figure 4(a) shows the empirical distribution of K_{10}^+ obtained from the sequence

$$Y_{n+1} = (3141592653Y_n + 2718281829) \bmod 2^{35}, \quad U_n = Y_n/2^{35},$$

and it is satisfactorily random. Part (b) of the figure came from the Fibonacci method; this sequence has *globally* nonrandom behavior, i.e., it can be shown that the observations X_n in the "maximum of 5" test do not have the correct distribution $F(x) = x^5$. Part (c) came from the notorious and impotent linear congruential sequence $Y_{n+1} = ((2^{18} + 1)Y_n + 1) \bmod 2^{35}$, $U_n = Y_n/2^{35}$.

The KS test applied to the data in Fig. 4 gives the results shown in (12). Referring to Table 2 for $n = 20$, we see that the values of K_{20}^+ and K_{20}^- for Fig. 4(b) are almost suspect (they lie at about the 5 percent and 88 percent levels) but not quite bad enough to be rejected outright. The value of K_{20}^- for Part (c) is, of course, completely out of line, so the "maximum of 5" test shows a definite failure of that random number generator.

We would expect the KS test in this experiment to have more difficulty locating global nonrandomness than local nonrandomness, since the basic observations in Fig. 4 were made on samples of only 10 numbers each. If we were to take 20 groups of 1000 numbers each, part (b) would show a much more significant deviation. To illustrate this point, a *single* KS test was applied to all 200 of the observations that led to Fig. 4, and the following results were obtained:

	Part (a)	Part (b)	Part (c)	
K_{200}^+	0.477	1.537	2.819	
K_{200}^-	0.817	0.194	0.058	(16)

The global nonrandomness of the Fibonacci generator has definitely been detected here.

We may summarize the Kolmogorov-Smirnov test as follows. We are given n independent observations X_1, \dots, X_n taken from some distribution specified by a continuous function $F(x)$. That is, $F(x)$ must be like the functions shown in Fig. 3(b) and 3(c), having no jumps like those in Fig. 3(a). The procedure explained just before Eqs. (13) is carried out on these observations, so we obtain the statistics K_n^+ and K_n^- . These statistics should be distributed according to Table 2.

Some comparisons between the KS test and the χ^2 test can now be made. In the first place, we should observe that the KS test may be used in conjunction with the χ^2 test, to give a better procedure than the *ad hoc* method we mentioned when summarizing the χ^2 test. (That is, there is a better way to proceed than to make three tests and to consider how many of the results were "suspect".) Suppose we have made, say, 10 independent χ^2 tests on different parts of a random sequence, so that values V_1, V_2, \dots, V_{10} have been obtained. It is not a good policy simply to count how many of the V 's are suspiciously large or small. This procedure will work in extreme cases, and very large or very small values

may mean that the sequence has too much local nonrandomness; but a better general method would be to plot the empirical distribution of these 10 values and to compare it to the correct distribution, which may be obtained from Table 1. This would give a clearer picture of the results of the χ^2 tests, and in fact the statistics K_{10}^+ and K_{10}^- could be determined as an indication of the success or failure. With only 10 values or even as many as 100 this could all be done easily by hand, using graphical methods; with a larger number of V 's, a computer subroutine for calculating the chi-square distribution would be necessary. Notice that *all 20 of the observations in Fig. 4(c) fall between the 5 and 95 percent levels*, so we would not have regarded any of them as suspicious, individually; yet collectively the empirical distribution shows that these observations are not at all right.

An important difference between the KS test and the chi-square test is that the KS test applies to distributions $F(x)$ having no jumps, while the chi-square test applies to distributions having nothing but jumps (since all observations are divided into k categories). The two tests are thus intended for different sorts of applications. Yet it is possible to apply the χ^2 test even when $F(x)$ is continuous, if we divide the domain of $F(x)$ into k parts and ignore all variations within each part. For example, if we want to test whether or not U_1, U_2, \dots, U_n can be considered to come from the uniform distribution between zero and one, we want to test if they have the distribution $F(x) = x$ for $0 \leq x \leq 1$. This is a natural application for the KS test. But we might also divide up the interval from 0 to 1 into $k = 100$ equal parts, count how many U 's fall into each part, and apply the chi-square test with 99 degrees of freedom. There are not many theoretical results available at the present time to compare the effectiveness of the KS test versus the chi-square test. The author has found some examples in which the KS test pointed out nonrandomness more clearly than the χ^2 test, and others in which the χ^2 test gave a more significant result. If, for example, the 100 categories mentioned above are numbered 0, 1, ..., 99, and if the deviations from the expected values are positive in compartments 0 to 49 but negative in compartments 50 to 99, then the empirical distribution function will be much further from $F(x)$ than the χ^2 value would indicate; but if the positive deviations occur in compartments 0, 2, ..., 98 and the negative ones occur in 1, 3, ..., 99, the empirical distribution function will tend to hug $F(x)$ much more closely. The kinds of deviations measured are therefore somewhat different. A χ^2 test was applied to the 200 observations that led to Fig. 4, with $k = 10$, and the respective values of V were 9.4, 17.7, and 39.3; so in this particular case the values are quite comparable to the KS values given in (16). Since the χ^2 test is intrinsically less accurate, and since it requires comparatively large values of n , the KS test has several advantages when a continuous distribution is to be tested.

A further example will also be of interest. The data that led to Fig. 2 were chi-square statistics based on $n = 200$ observations of the "maximum-of- t " criterion for $1 \leq t \leq 5$, with the range divided into 10 equally probable parts. KS statistics K_{200}^+ and K_{200}^- can be computed from exactly the same sets of 200 observations, and the results can be tabulated in just the same way as we did

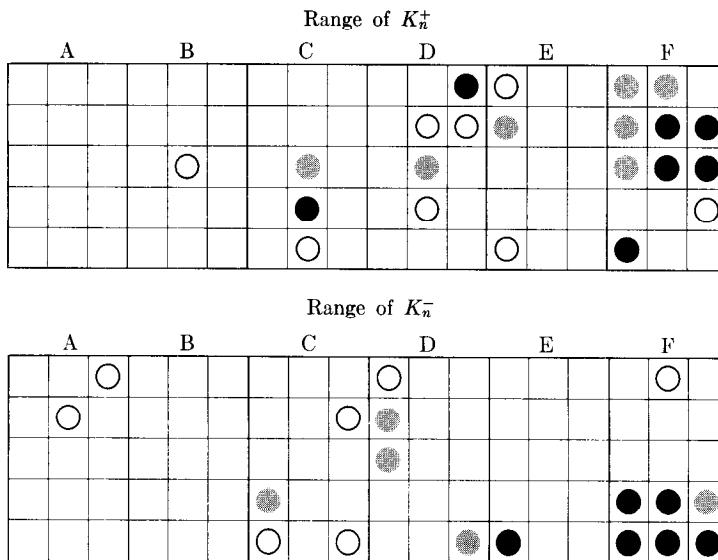


Fig. 5. The KS tests applied to the same data as Fig. 2.

in Fig. 2 (showing which KS values are beyond the 99-percent level, etc.); the results in this case are shown in Fig. 5. Note that Generator D (Lehmer's original method) shows up very badly in Fig. 5, while chi-square tests on the very same data revealed no difficulty in Fig. 2; contrariwise, Generator E (the Fibonacci method) does not look so bad in Fig. 5. The good generators, A and B, passed all tests satisfactorily. The reasons for the discrepancies between Fig. 2 and Fig. 5 are primarily that (a) the number of observations, 200, is really not large enough for a powerful test, and (b) the "reject," "suspect," "almost suspect" ranking criterion is itself suspect.

(Incidentally, it is not fair to blame Lehmer for using a "bad" random number generator in the 1940s, since his actual use of Generator D was quite valid. The ENIAC computer was a highly parallel machine, programmed by means of a plugboard; Lehmer set it up so that one of its accumulators was repeatedly multiplying its own contents by $23 \bmod (10^8 + 1)$, yielding a new value every few microseconds. Since this multiplier 23 is too small, we know that each value obtained by such a process was too strongly related to the preceding value to be considered sufficiently random; but the durations of time between actual uses of the values in the special accumulator by the accompanying program were comparatively long and subject to some fluctuation. So the effective multiplier was 23^k for large, varying values of k .)

C. History, bibliography, and theory. The chi-square test was introduced by Karl Pearson in 1900 [*Philosophical Magazine*, Series 5, 50, 157–175]. Pearson's important paper is regarded as one of the foundations of modern statistics, since before that time people would simply plot experimental results graphically and assert that they were correct. In his paper, Pearson gave several interesting

examples of the previous misuse of statistics; and he also proved that certain runs at roulette (which he had experienced during two weeks at Monte Carlo in 1892) were so far from the expected frequencies that odds against the assumption of an honest wheel were some 10^{29} to one! A general discussion of the chi-square test and an extensive bibliography appear in the survey article by William G. Cochran, *Annals Math. Stat.* **23** (1952), 315–345.

Let us now consider a brief derivation of the theory behind the chi-square test. The exact probability that $Y_1 = y_1, \dots, Y_k = y_k$ is easily seen to be

$$\frac{n!}{y_1! \dots y_k!} p_1^{y_1} \dots p_k^{y_k}. \quad (17)$$

If we assume that Y_s has the value y_s with the Poisson probability

$$\frac{e^{-np_s} (np_s)^{y_s}}{y_s!},$$

and that the Y 's are independent, then (Y_1, \dots, Y_k) will equal (y_1, \dots, y_k) with probability

$$\prod_{1 \leq s \leq k} \frac{e^{-np_s} (np_s)^{y_s}}{y_s!},$$

and $Y_1 + \dots + Y_k$ will equal n with probability

$$\sum_{\substack{y_1 + \dots + y_k = n \\ y_1, \dots, y_k \geq 0}} \prod_{1 \leq s \leq k} \frac{e^{-np_s} (np_s)^{y_s}}{y_s!} = \frac{e^{-n} n^n}{n!}.$$

If we assume that they are independent except for the condition $Y_1 + \dots + Y_k = n$, the probability that $(Y_1, \dots, Y_k) = (y_1, \dots, y_k)$ is the quotient

$$\left(\prod_{1 \leq s \leq k} \frac{e^{-np_s} (np_s)^{y_s}}{y_s!} \right) / \left(\frac{e^{-n} n^n}{n!} \right),$$

which equals (17). We may therefore regard the Y 's as independently Poisson distributed, except for the fact that they have a fixed sum.

It is convenient to make a change of variables,

$$Z_s = \frac{Y_s - np_s}{\sqrt{np_s}}, \quad (18)$$

so that $V = Z_1^2 + \dots + Z_k^2$. The condition $Y_1 + \dots + Y_k = n$ is equivalent to requiring that

$$\sqrt{p_1} Z_1 + \dots + \sqrt{p_k} Z_k = 0. \quad (19)$$

Let us consider the $(k - 1)$ -dimensional space S of all vectors (Z_1, \dots, Z_k) such that (19) holds. For large values of n , each Z_s has approximately the

normal distribution (cf. exercise 1.2.10–16); therefore points in a differential volume $dz_2 \dots dz_k$ of S occur with probability approximately proportional to $\exp(-(z_1^2 + \dots + z_k^2)/2)$. (It is at this point in the derivation that the chi-square method becomes only an approximation for large n .) The probability that $V \leq v$ is now

$$\frac{\int_{(z_1, \dots, z_k) \text{ in } S \text{ and } z_1 + \dots + z_k \leq v} \exp(-(z_1^2 + \dots + z_k^2)/2) dz_2 \dots dz_k}{\int_{(z_1, \dots, z_k) \text{ in } S} \exp(-(z_1^2 + \dots + z_k^2)/2) dz_2 \dots dz_k}. \quad (20)$$

Since the hyperplane (19) passes through the origin of k -dimensional space, the numerator in (20) is an integration over the interior of a $(k - 1)$ -dimensional hypersphere centered at the origin. An appropriate transformation to generalized polar coordinates with radius χ and angles $\omega_1, \dots, \omega_{k-2}$ transforms (20) into

$$\frac{\int_{\chi^2 \leq v} e^{-\chi^2/2} \chi^{k-2} f(\omega_1, \dots, \omega_{k-2}) d\chi d\omega_1 \dots d\omega_{k-2}}{\int e^{-\chi^2/2} \chi^{k-2} f(\omega_1, \dots, \omega_{k-2}) d\chi d\omega_1 \dots d\omega_{k-2}}$$

for some function f (see exercise 15); then integration over the angles $\omega_1, \dots, \omega_{k-2}$ gives a constant factor that cancels from numerator and denominator. We finally obtain the formula

$$\frac{\int_0^{\sqrt{v}} e^{-x^2/2} x^{k-2} dx}{\int_0^\infty e^{-x^2/2} x^{k-2} dx} \quad (21)$$

for the approximate probability that $V \leq v$.

The above derivation uses the symbol χ to stand for the radial length, just as Pearson did in his original paper; this is how the χ^2 test got its name. Substituting $t = \chi^2/2$, the integrals can be expressed in terms of the incomplete gamma function, which we discussed in Section 1.2.11.3:

$$\lim_{n \rightarrow \infty} \text{probability that } (V \leq v) = \gamma\left(\frac{k-1}{2}, \frac{v}{2}\right) / \Gamma\left(\frac{k-1}{2}\right). \quad (22)$$

This is the definition of the chi-square distribution with $k - 1$ degrees of freedom.

We now turn to the KS test. In 1933, A. N. Kolmogorov proposed a test based on the statistic

$$K_n = \sqrt{n} \max_{-\infty < x < +\infty} |F_n(x) - F(x)| = \max(K_n^+, K_n^-). \quad (23)$$

N. V. Smirnov gave several modifications of this test in 1939, including the individual examination of K_n^+ and K_n^- as we have suggested above. There is a large family of similar tests, but the K_n^+ and K_n^- statistics seem to be most convenient for computer application. A comprehensive review of the literature concerning KS tests and their generalizations, including an extensive bibliography, appears in a monograph by J. Durbin, *Regional Conf. Series on Applied Math.* 9 (SIAM, 1973).

To study the distribution of K_n^+ and K_n^- , we begin with the following basic fact: If X is a random variable with the continuous distribution $F(x)$, then $F(X)$ is a uniformly distributed real number between 0 and 1. To prove this, we need only verify that if $0 \leq y \leq 1$ we have $F(X) \leq y$ with probability y . Since F is continuous, $F(x_0) = y$ for some x_0 ; thus the probability that $F(X) \leq y$ is the probability that $X \leq x_0$. By definition, the latter probability is $F(x_0)$, that is, it is y .

Let $Y_j = n F(X_j)$, for $1 \leq j \leq n$, where the X 's have been sorted as in Step 2 above. Then the variables Y_j are essentially the same as independent, uniformly distributed random numbers between 0 and 1 that have been sorted into nondecreasing order, $Y_1 \leq Y_2 \leq \dots \leq Y_n$; and the first equation of (13) may be transformed into

$$K_n^+ = \frac{1}{\sqrt{n}} \max(1 - Y_1, 2 - Y_2, \dots, n - Y_n).$$

If $0 \leq t \leq n$, the probability that $K_n^+ \leq t/\sqrt{n}$ is therefore the probability that $Y_j \geq j-t$ for $1 \leq j \leq n$. This is not hard to express in terms of n -dimensional integrals,

$$\frac{\int_{\alpha_n}^n dy_n \int_{\alpha_{n-1}}^{y_n} dy_{n-1} \dots \int_{\alpha_1}^{y_2} dy_1}{\int_0^n dy_n \int_0^{y_n} dy_{n-1} \dots \int_0^{y_2} dy_1}, \quad \text{where } \alpha_j = \max(j-t, 0). \quad (24)$$

The denominator here is immediately evaluated: it is found to be $n^n/n!$, which makes sense since the hypercube of all vectors (y_1, y_2, \dots, y_n) with $0 \leq y_j < n$ has volume n^n , and it can be divided into $n!$ equal parts corresponding to each possible ordering of the y 's. The integral in the numerator is a little more difficult, but it yields to the attack suggested in exercise 17, and we get the general formula

$$\text{probability that } \left(K_n^+ \leq \frac{t}{\sqrt{n}} \right) = \frac{t}{n^n} \sum_{0 \leq k \leq t} \binom{n}{k} (k-t)^k (t+n-k)^{n-k-1}. \quad (25)$$

The distribution of K_n^- is exactly the same. Equation (25) was first obtained by Z. W. Birnbaum and Fred H. Tingey [*Annals Math. Stat.* **22** (1951), 592–596]; it may be used to extend Table 2.

In his original paper, Smirnov proved that

$$\lim_{n \rightarrow \infty} \text{probability that } (K_n^+ \leq s) = 1 - e^{-2s^2}, \quad \text{if } s \geq 0. \quad (26)$$

This together with (25) implies that, for all $s \geq 0$, we have

$$\lim_{n \rightarrow \infty} \frac{s}{\sqrt{n}} \sum_{\sqrt{n}s < k \leq n} \binom{n}{k} \left(\frac{k}{n} - \frac{s}{\sqrt{n}} \right)^k \left(\frac{s}{\sqrt{n}} + 1 - \frac{k}{n} \right)^{n-k-1} = e^{-2s^2}. \quad (27)$$

The more precise asymptotic formulas in Table 2 follow from results obtained by D. A. Darling [*Theory of Prob. and Appl.* 5 (1960), 356-361], who proved among other things that $K_n^+ \leq s$ with probability

$$1 - e^{-2s^2} \left(1 - \frac{2}{3}s/\sqrt{n} + O(1/n) \right) \quad (28)$$

for all fixed $s \geq 0$.

EXERCISES

1. [00] What line of the chi-square table should be used to check whether or not the value $V = 7\frac{7}{48}$ of Eq. (5) is improbably high?

2. [20] If two dice are “loaded” so that, on one die, the value 1 will turn up exactly twice as often as any of the other values, and the other die is similarly biased towards 6, compute the probability p_s that a total of exactly s will appear on the two dice, for $2 \leq s \leq 12$.

► 3. [23] Some dice that were loaded as described in the previous exercise were rolled 144 times, and the following values were observed:

value of $s =$	2	3	4	5	6	7	8	9	10	11	12
observed number, $Y_s =$	2	6	10	16	18	32	20	13	16	9	2

Apply the chi-square test to these values, using the probabilities in (1), pretending it is not known that the dice are in fact faulty. Does the chi-square test detect the bad dice? If not, explain why not.

► 4. [23] The author actually obtained the data in experiment 1 of (9) by simulating dice in which one was normal, the other was loaded so that it always turned up 1 or 6. (The latter two possibilities were equally probable.) Compute the probabilities that replace (1) in this case, and by using a chi-square test decide if the results of that experiment are consistent with the dice being loaded in this way.

5. [22] Let $F(x)$ be the uniform distribution, Fig. 3(b). Find K_{20}^+ and K_{20}^- for the following 20 observations:

$$\begin{aligned} 0.414, & \quad 0.732, & \quad 0.236, & \quad 0.162, & \quad 0.259, & \quad 0.442, & \quad 0.189, & \quad 0.693, & \quad 0.098, & \quad 0.302, \\ 0.442, & \quad 0.434, & \quad 0.141, & \quad 0.017, & \quad 0.318, & \quad 0.869, & \quad 0.772, & \quad 0.678, & \quad 0.354, & \quad 0.718, \end{aligned}$$

and state whether these observations are significantly different from expected behavior with respect to either of these two tests.

6. [M20] Consider $F_n(x)$, as given in Eq. (10), for fixed x . What is the probability that $F_n(x) = s/n$, given an integer s ? What is the mean value of $F_n(x)$? What is the standard deviation?

7. [M15] Show that K_n^+ and K_n^- can never be negative. What is the largest possible value K_n^+ can be?

8. [00] The text describes an experiment in which 20 values of the statistic K_{10}^+ were obtained in the study of a random sequence. These values were plotted, to obtain Fig. 4, and a KS statistic was computed from the resulting graph. Why were the table entries for $n = 20$ used to study the resulting statistic, instead of the table entries for $n = 10$?

► 9. [20] The experiment described in the text consisted of plotting 20 values of K_{10}^+ , computed from the “maximum of 5” test applied to different parts of a random sequence. We could have computed also the corresponding 20 values of K_{10}^- ; since K_{10}^- has the same distribution as K_{10}^+ , we could lump together the 40 values thus obtained (that is, 20 of the K_{10}^+ ’s and 20 of the K_{10}^- ’s), and a KS test could be applied so that we would get new values K_{40}^+, K_{40}^- . Discuss the merits of this idea.

► 10. [20] Suppose a chi-square test is done by making n observations, and the value V is obtained. Now we repeat the test on these same n observations over again (getting, of course, the same results), and we put together the data from both tests, regarding it as a single chi-square test with $2n$ observations. (This procedure violates the text’s stipulation that all of the observations must be independent of one another.) How is the second value of V related to the first one?

11. [10] Solve exercise 10 substituting the KS test for the chi-square test.

12. [M28] Suppose a chi-square test is made on a set of n observations, assuming that p_s is the probability that each observation falls into category s ; but suppose that in actual fact the observations have probability $q_s \neq p_s$ of falling into category s . (Cf. exercise 3.) We would, of course, like the chi-square test to detect the fact that the p_s assumption was incorrect. Show that this *will* happen, if n is large enough. Prove also the analogous result for the KS test.

13. [M24] Prove that Eqs. (13) are equivalent to Eqs. (11).

► 14. [HM26] Let Z_s be given by Eq. (18). Show directly by using Stirling’s approximation that the multinomial probability

$$n! p_1^{Y_1} \dots p_k^{Y_k} / Y_1! \dots Y_k! = e^{-V/2} / \sqrt{(2n\pi)^{k-1} p_1 \dots p_k} + O(n^{-k/2}),$$

if Z_1, Z_2, \dots, Z_k are bounded as $n \rightarrow \infty$. (This idea leads to a proof of the chi-square test that is much closer to “first principles,” and requires less handwaving, than the derivation in the text.)

15. [HM24] Polar coordinates in two dimensions are conventionally defined by the equations $x = r \cos \theta$ and $y = r \sin \theta$. For the purposes of integration, we have $dx dy = r dr d\theta$. More generally, in n -dimensional space we can let

$$x_k = r \sin \theta_1 \dots \sin \theta_{k-1} \cos \theta_k, \quad 1 \leq k < n, \quad x_n = r \sin \theta_1 \dots \sin \theta_{n-1}.$$

Show that in this case

$$dx_1 dx_2 \dots dx_n = |r^{n-1} \sin^{n-2} \theta_1 \dots \sin \theta_{n-2} dr d\theta_1 \dots d\theta_{n-1}|.$$

- 16. [HM35] Generalize Theorem 1.2.11.3A to find the value of

$$\gamma(x+1, x+z\sqrt{2x}+y)/\Gamma(x+1),$$

for large x and fixed y, z . Disregard terms of the answer that are $O(1/x)$. Use this result to find the approximate solution, t , to the equation

$$\gamma\left(\frac{\nu}{2}, \frac{t}{2}\right) / \Gamma\left(\frac{\nu}{2}\right) = p,$$

for large ν and fixed p , thereby accounting for the asymptotic formulas indicated in Table 1. [Hint: See exercise 1.2.11.3-8.]

17. [HM26] Let t be a fixed real number. For $0 \leq k \leq n$, let

$$P_{nk}(x) = \int_{n-t}^x dx_n \int_{n-1-t}^{x_n} dx_{n-1} \dots \int_{k+1-t}^{x_{k+2}} dx_{k+1} \int_0^{x_{k+1}} dx_k \dots \int_0^{x_2} dx_1;$$

by convention, let $P_{00}(x) = 1$. Prove the following relations:

a) $P_{nk}(x) = \int_n^{x+t} dx_n \int_{n-1}^{x_n} dx_{n-1} \dots \int_{k+1}^{x_{k+2}} dx_{k+1} \int_t^{x_{k+1}} dx_k \dots \int_t^{x_2} dx_1.$

b) $P_{n0}(x) = (x+t)^n/n! - (x+t)^{n-1}/(n-1)!$.

c) $P_{nk}(x) - P_{n(k-1)}(x) = \frac{(k-t)^k}{k!} P_{(n-k)0}(x-k)$, if $1 \leq k \leq n$.

d) Obtain a general formula for $P_{nk}(x)$, and apply it to the evaluation of Eq. (24).

18. [M20] Give a “simple” reason why K_n^- has the same probability distribution as K_n^+ .

19. [HM48] Develop tests, analogous to the Kolmogorov-Smirnov test, for use with multivariate distributions $F(x_1, \dots, x_r) = \text{probability that } (X_1 \leq x_1, \dots, X_r \leq x_r)$. (Such procedures could be used, for example, in place of the “serial test” in the next section.)

20. [HM41] Deduce further terms of the asymptotic behavior of the KS distribution, extending (28).

21. [M40] Although the text states that the KS test should be applied only when $F(x)$ is a continuous distribution function, it is, of course, possible to try to compute K_n^+ and K_n^- even when the distribution has jumps. Analyze the probable behavior of K_n^+ and K_n^- for various discontinuous distributions $F(x)$. Compare the effectiveness of the resulting statistical test with the chi-square test on several samples of random numbers.

22. [HM46] Investigate the “improved” KS test suggested in the answer to exercise 6.

23. [M22] (T. Gonzalez, S. Sahni, and W. R. Franta.) (a) Suppose that the maximum value in formula (13) for the KS statistic K_n^+ occurs at a given index j where $\lfloor nF(X_j) \rfloor = k$. Prove that $F(X_j) = \max_{1 \leq i \leq n} \{F(X_i) \mid \lfloor nF(X_i) \rfloor = k\}$. (b) Design an algorithm that calculates K_n^+ and K_n^- in $O(n)$ steps (without sorting).

- 24. [40] Experiment with various probability distributions (p, q, r) on three categories, where $p + q + r = 1$, by computing the exact distribution of the chi-square statistic V for various n , thereby determining how accurate an approximation the chi-square distribution with two degrees of freedom really is.

3.3.2. Empirical Tests

In this section we shall discuss ten kinds of specific tests that have been applied to sequences in order to investigate their randomness. The discussion of each test has two parts: (a) a “plug-in” description of how to perform the test; and (b) a study of the theoretical basis for the test. (Readers lacking mathematical training may wish to skip over the theoretical discussions. Conversely, mathematically-inclined readers may find the associated theory quite interesting, even if they never intend to test random number generators, since some instructive combinatorial questions are involved here.)

Each test is applied to a sequence

$$\langle U_n \rangle = U_0, U_1, U_2, \dots \quad (1)$$

of real numbers, which purports to be independently and uniformly distributed between zero and one. Some of the tests are designed primarily for integer-valued sequences, instead of the real-valued sequence (1). In this case, the auxiliary sequence

$$\langle Y_n \rangle = Y_0, Y_1, Y_2, \dots, \quad (2)$$

which is defined by the rule

$$Y_n = \lfloor dU_n \rfloor, \quad (3)$$

is used instead. This is a sequence of integers that purports to be independently and uniformly distributed between 0 and $d - 1$. The number d is chosen for convenience; for example, we might have $d = 64 = 2^6$ on a binary computer, so that Y_n represents the six most significant bits of the binary representation of U_n . The value of d should be large enough so that the test is meaningful, but not so large that the test becomes impractically difficult to carry out.

The quantities U_n , Y_n , and d will have the above significance throughout this section, although the value of d will probably be different in different tests.

A. Equidistribution test (Frequency test). The first requirement that sequence (1) must meet is that its numbers are, in fact, uniformly distributed between zero and one. There are two ways to make this test: (a) Use the Kolmogorov-Smirnov test, with $F(x) = x$ for $0 \leq x \leq 1$. (b) Let d be a convenient number, e.g., 100 on a decimal computer, 64 or 128 on a binary computer, and use the sequence (2) instead of (1). For each integer r , $0 \leq r < d$, count the number of times that $Y_j = r$ for $0 \leq j < n$, and then apply the chi-square test using $k = d$ and probability $p_s = 1/d$ for each category.

The theory behind this test has been covered in Section 3.3.1.

The equanimity of your average tosser of coins depends upon a law . . . which ensures that he will not upset himself by losing too much nor upset his opponent by winning too often.

—TOM STOPPARD, *Rosencrantz & Guildenstern are Dead* (1966)

B. Serial test. More generally, we want pairs of successive numbers to be uniformly distributed in an independent manner. The sun comes up just about as often as it goes down, in the long run, but this doesn't make its motion random.

To carry out the serial test, we simply count the number of times that the pair $(Y_{2j}, Y_{2j+1}) = (q, r)$ occurs, for $0 \leq j < n$; these counts are to be made for each pair of integers (q, r) with $0 \leq q, r < d$, and the chi-square test is applied to these $k = d^2$ categories with probability $1/d^2$ in each category. As with the equidistribution test, d may be any convenient number, but it will be somewhat smaller than the values suggested above since a valid chi-square test should have n large compared to k (say $n \geq 5d^2$ at least).

Clearly we can generalize this test to triples, quadruples, etc., instead of pairs (see exercise 2); however, the value of d must then be severely reduced in order to avoid having too many categories. When quadruples and larger numbers of adjacent elements are considered, we therefore make use of less exact tests such as the poker test or the maximum test described below.

Note that $2n$ numbers of the sequence (2) are used in this test in order to make n observations. It would be a mistake to perform the serial test on the pairs $(Y_0, Y_1), (Y_1, Y_2), \dots, (Y_{n-1}, Y_n)$; can the reader see why? We might perform another serial test on the pairs (Y_{2j+1}, Y_{2j+2}) , and expect the sequence to pass both tests. Alternatively, I. J. Good has proved that if d is prime, and if the pairs $(Y_0, Y_1), (Y_1, Y_2), \dots, (Y_{n-1}, Y_n)$ are used, and if we use the usual chi-square method to compute both the statistics V_2 for the serial test and V_1 for the frequency test on Y_0, \dots, Y_{n-1} with the same value of d , then $V_2 - 2V_1$ should have the chi-square distribution with $(d-1)^2$ degrees of freedom when n is large. (See *Proc. Cambridge Phil. Soc.* **49** (1953), 276–284; *Annals Math. Stat.* **28** (1957), 262–264.)

C. Gap test. Another test is used to examine the length of “gaps” between occurrences of U_j in a certain range. If α and β are two real numbers with $0 \leq \alpha < \beta \leq 1$, we want to consider the lengths of consecutive subsequences $U_j, U_{j+1}, \dots, U_{j+r}$ in which U_{j+r} lies between α and β but the other U 's do not. (This subsequence of $r + 1$ numbers represents a gap of length r .)

Algorithm G (*Data for gap test*). The following algorithm, applied to the sequence (1) for any given values of α and β , counts the number of gaps of lengths $0, 1, \dots, t-1$ and the number of gaps of length $\geq t$, until n gaps have been tabulated.

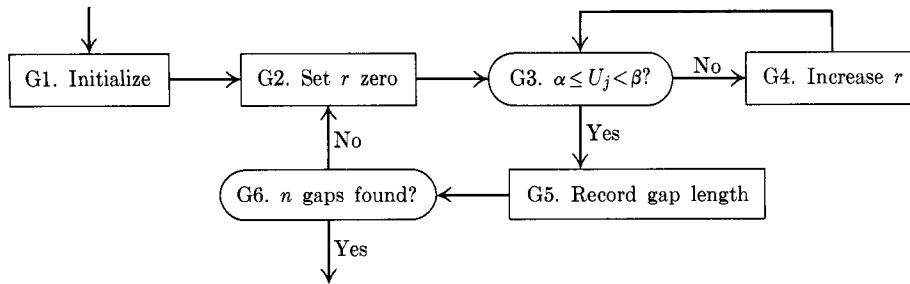


Fig. 6. Gathering data for the gap test. (Algorithms for the “coupon-collector’s test” and the “run test” are similar.)

G1. [Initialize.] Set $j \leftarrow -1$, $s \leftarrow 0$, and set $\text{COUNT}[r] \leftarrow 0$ for $0 \leq r \leq t$.

G2. [Set r zero.] Set $r \leftarrow 0$.

G3. [$\alpha \leq U_j < \beta?$] Increase j by 1. If $U_j \geq \alpha$ and $U_j < \beta$, go to step G5.

G4. [Increase r .] Increase r by one, and return to step G3.

G5. [Record gap length.] (A gap of length r has now been found.) If $r \geq t$, increase $\text{COUNT}[t]$ by one, otherwise increase $\text{COUNT}[r]$ by one.

G6. [n gaps found?] Increase s by one. If $s < n$, return to step G2. ■

After this algorithm has been performed, the chi-square test is applied to the $k = t + 1$ values of $\text{COUNT}[0]$, $\text{COUNT}[1]$, ..., $\text{COUNT}[t]$, using the following probabilities:

$$p_0 = p, \quad p_1 = p(1-p), \quad p_2 = p(1-p)^2, \quad \dots, \\ p_{t-1} = p(1-p)^{t-1}, \quad p_t = (1-p)^t. \quad (4)$$

Here $p = \beta - \alpha$, the probability that $\alpha \leq U_j < \beta$. The values of n and t are to be chosen, as usual, so that each of the values of $\text{COUNT}[r]$ is expected to be 5 or more, preferably more.

The gap test is often applied with $\alpha = 0$ or $\beta = 1$ in order to omit one of the comparisons in step G3. The special cases $(\alpha, \beta) = (0, \frac{1}{2})$ or $(\frac{1}{2}, 1)$ give rise to tests that are sometimes called “runs above the mean” and “runs below the mean,” respectively.

The probabilities in Eq. (4) are easily deduced, so this derivation is left to the reader. Note that the gap test as described above observes the lengths of n gaps; it does not observe the gap lengths among n numbers. If the sequence $\langle U_n \rangle$ is sufficiently nonrandom, Algorithm G may not terminate. Other gap tests that examine a fixed number of U ’s have also been proposed (see exercise 5).

D. Poker test (Partition test). The “classical” poker test considers n groups of five successive integers, $(Y_{5j}, Y_{5j+1}, \dots, Y_{5j+4})$ for $0 \leq j < n$, and observes which of the following seven patterns is matched by each quintuple:

All different:	$abcde$	Full house:	$aaabb$
One pair:	$aabcd$	Four of a kind:	$aaaab$
Two Pairs:	$aabbc$	Five of a kind:	$aaaaa$
Three of a kind:	$aaabc$		

A chi-square test is based on the number of quintuples in each category.

It is reasonable to ask for a somewhat simpler version of this test, to facilitate the programming involved. A good compromise would simply be to count the number of *distinct* values in the set of five. We would then have five categories:

- 5 different = all different;
- 4 different = one pair;
- 3 different = two pairs, or three of a kind;
- 2 different = full house, or four of a kind;
- 1 different = five of a kind.

This breakdown is easier to determine systematically, and the test is nearly as good.

In general we can consider n groups of k successive numbers, and we can count the number of k -tuples with r different values. A chi-square test is then made, using the probability

$$p_r = \frac{d(d-1)\dots(d-r+1)}{d^k} \left\{ \begin{matrix} k \\ r \end{matrix} \right\} \quad (5)$$

that there are r different. (The Stirling numbers $\left\{ \begin{matrix} k \\ r \end{matrix} \right\}$ are defined in Section 1.2.6, and they can readily be computed using the formulas given there.) Since the probability p_r is very small when $r = 1$ or 2, we generally lump a few categories of low probability together before the chi-square test is applied.

To derive the proper formula for p_r , we must count how many of the d^k k -tuples of numbers between 0 and $d-1$ have exactly r different elements, and divide the total by d^k . Since $d(d-1)\dots(d-r+1)$ is the number of ordered choices of r things from a set of d objects, we need only show that $\left\{ \begin{matrix} k \\ r \end{matrix} \right\}$ is the number of ways to partition a set of k elements into exactly r parts. Therefore exercise 1.2.6-64 completes the derivation of Eq. (5).

E. Coupon collector's test. This test is related to the poker test somewhat as the gap test is related to the frequency test. The sequence Y_0, Y_1, \dots is used, and we observe the lengths of segments $Y_{j+1}, Y_{j+2}, \dots, Y_{j+r}$ required to get a “complete set” of integers from 0 to $d-1$. Algorithm C describes this precisely:

Algorithm C (*Data for coupon collector's test*). Given a sequence of integers Y_0, Y_1, \dots , with $0 \leq Y_j < d$, this algorithm counts the lengths of n consecutive "coupon collector" segments. At the conclusion of the algorithm, $\text{COUNT}[r]$ is the number of segments with length r , for $d \leq r < t$, and $\text{COUNT}[t]$ is the number of segments with length $\geq t$.

- C1. [Initialize.] Set $j \leftarrow -1$, $s \leftarrow 0$, and set $\text{COUNT}[r] \leftarrow 0$ for $d \leq r \leq t$.
- C2. [Set q, r zero.] Set $q \leftarrow r \leftarrow 0$, and set $\text{OCCURS}[k] \leftarrow 0$ for $0 \leq k < d$.
- C3. [Next observation.] Increase r and j by 1. If $\text{OCCURS}[Y_j] \neq 0$, repeat this step.
- C4. [Complete set?] Set $\text{OCCURS}[Y_j] \leftarrow 1$ and $q \leftarrow q + 1$. (The subsequence observed so far contains q distinct values; if $q = d$, we therefore have a complete set.) If $q < d$, return to step C3.
- C5. [Record the length.] If $r \geq t$, increase $\text{COUNT}[t]$ by one, otherwise increase $\text{COUNT}[r]$ by one.
- C6. [n found?] Increase s by one. If $s < n$, return to step C2. ■

For an example of this algorithm, see exercise 7. We may think of a boy collecting d types of coupons, which are randomly distributed in his breakfast cereal boxes; he must keep eating more cereal until he has one coupon of each type.

A chi-square test is to be applied to $\text{COUNT}[d], \text{COUNT}[d+1], \dots, \text{COUNT}[t]$, with $k = t-d+1$, after Algorithm C has counted n lengths. The corresponding probabilities are

$$p_r = \frac{d!}{d^r} \binom{r-1}{d-1}, \quad d \leq r < t; \quad p_t = 1 - \frac{d!}{d^{t-1}} \binom{t-1}{d}. \quad (6)$$

To derive these probabilities, we simply note that if q_r denotes the probability that a subsequence of length r is incomplete, then

$$q_r = 1 - \frac{d!}{d^r} \binom{r}{d}$$

by Eq. (5); for this means we have an r -tuple of elements that do not have all d different values. Then (6) follows from the relations $p_r = q_{r-1} - q_r$ for $d \leq r < t$; $p_t = q_{t-1}$.

For formulas that arise in connection with generalizations of the coupon collector's test, see exercises 9 and 10 and also the paper by Hermann von Schelling, *AMM* 61 (1954), 306–311.

F. Permutation test. Divide the input sequence into n groups of t elements each, that is, $(U_{jt}, U_{jt+1}, \dots, U_{jt+t-1})$ for $0 \leq j < n$. The elements in each group can have $t!$ possible relative orderings; the number of times each ordering appears is counted, and a chi-square test is applied with $k = t!$ and with probability $1/t!$ for each ordering.

For example, if $t = 3$ we would have six possible categories, according to whether $U_{3j} < U_{3j+1} < U_{3j+2}$ or $U_{3j} < U_{3j+2} < U_{3j+1}$ or \dots or $U_{3j+2} < U_{3j+1} < U_{3j}$. We assume in this test that equality between U 's does not occur; such an assumption is justified, for the probability that two U 's are equal is zero.

A convenient way to perform the permutation test on a computer makes use of the following algorithm, which is of interest in itself:

Algorithm P (Analyze a permutation). Given a sequence of distinct elements (U_1, \dots, U_t) , we compute an integer $f(U_1, \dots, U_t)$ such that

$$0 \leq f(U_1, \dots, U_t) < t!,$$

and $f(U_1, \dots, U_t) = f(V_1, \dots, V_t)$ if and only if (U_1, \dots, U_t) and (V_1, \dots, V_t) have the same relative ordering.

- P1. [Initialize.] Set $r \leftarrow t$, $f \leftarrow 0$. (During this algorithm we will have $0 \leq f < t!/r!$.)
- P2. [Find maximum.] Find the maximum of $\{U_1, \dots, U_r\}$, and suppose that U_s is the maximum. Set $f \leftarrow r \cdot f + s - 1$.
- P3. [Exchange.] Exchange $U_r \leftrightarrow U_s$.
- P4. [Decrease r .] Decrease r by one. If $r > 1$, return to step P2. ■

Note that the sequence (U_1, \dots, U_t) will have been sorted into ascending order when this algorithm stops. To prove that the result f uniquely characterizes the *initial* order of (U_1, \dots, U_t) , we note that Algorithm P can be run backwards: For $r = 2, 3, \dots, t$, set $s \leftarrow f \bmod r$, $f \leftarrow \lfloor f/r \rfloor$, and exchange U_r, U_s . It is easy to see that this will undo the effects of steps P2–P4; hence no two permutations can yield the same value of f , and Algorithm P performs as advertised.

The essential idea that underlies Algorithm P is a mixed-radix representation called the “factorial number system”: Every integer in the range $0 \leq f < t!$ can be uniquely written in the form

$$\begin{aligned} f &= (\dots(c_{t-1} \times (t-1) + c_{t-2}) \times (t-2) + \dots + c_2) \times 2 + c_1 \\ &= (t-1)!c_{t-1} + (t-2)!c_{t-2} + \dots + 2!c_2 + 1!c_1 \end{aligned} \tag{7}$$

where the “digits” c_j are integers satisfying

$$0 \leq c_j \leq j, \quad \text{for } 1 \leq j < t. \tag{8}$$

In Algorithm P, $c_{r-1} = s - 1$ when step P2 is performed for a given value of r .

G. Run test. A sequence may also be tested for “runs up” and “runs down.” This means we examine the length of *monotone* subsequences of the original sequence, i.e., segments that are increasing or decreasing.

As an example of the precise definition of a run, consider the sequence of ten numbers “1298536704”; putting a vertical line at the left and right and between X_j and X_{j+1} whenever $X_j > X_{j+1}$, we obtain

$$| 1 \ 2 \ 9 | 8 | 5 | 3 \ 6 \ 7 | 0 \ 4 |, \quad (9)$$

which displays the “runs up”: there is a run of length 3, followed by two runs of length 1, followed by another run of length 3, followed by a run of length 2. The algorithm of exercise 12 shows how to tabulate the length of “runs up.”

Unlike the gap test and the coupon collector’s test (which are in many other respects similar to this test), we *should not apply a chi-square test to the above data*, since adjacent runs are *not* independent. A long run will tend to be followed by a short run, and conversely. This lack of independence is enough to invalidate a straightforward chi-square test. Instead, the following statistic may be computed, when the run lengths have been determined as in exercise 12:

$$V = \frac{1}{n} \sum_{1 \leq i, j \leq 6} (\text{COUNT}[i] - nb_i)(\text{COUNT}[j] - nb_j)a_{ij}, \quad (10)$$

where n is the length of the sequence, and the coefficients a_{ij} and b_i are equal to

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{pmatrix} = \begin{pmatrix} 4529.4 & 9044.9 & 13568 & 18091 & 22615 & 27892 \\ 9044.9 & 18097 & 27139 & 36187 & 45234 & 55789 \\ 13568 & 27139 & 40721 & 54281 & 67852 & 83685 \\ 18091 & 36187 & 54281 & 72414 & 90470 & 111580 \\ 22615 & 45234 & 67852 & 90470 & 113262 & 139476 \\ 27892 & 55789 & 83685 & 111580 & 139476 & 172860 \end{pmatrix} \quad (11)$$

$$(b_1 \ b_2 \ b_3 \ b_4 \ b_5 \ b_6) = \left(\frac{1}{6} \ \frac{5}{24} \ \frac{11}{120} \ \frac{19}{720} \ \frac{29}{5040} \ \frac{1}{840} \right).$$

(The values of a_{ij} shown here are approximate only; exact values may be obtained by using formulas derived below.) The statistic V in (10) should have the chi-square distribution with six (not five) degrees of freedom, when n is large. The value of n should be, say, 4000 or more. The same test can be applied to “runs down.”

A vastly simpler and more practical run test appears in exercise 14, so a reader who is interested only in testing random number generators should skip the next few pages and go on to the “maximum-of- t test” after looking at exercise 14. On the other hand it is instructive from a mathematical standpoint to see how a complicated run test with interdependent runs can be treated, so we shall now digress for a moment.

Given any permutation on n elements, let $Z_{pi} = 1$ if position i is the beginning of an ascending run of length p or more, and let $Z_{pi} = 0$ otherwise. For example, consider the permutation (9) with $n = 10$; we have

$$Z_{11} = Z_{21} = Z_{31} = Z_{14} = Z_{15} = Z_{16} = Z_{26} = Z_{36} = Z_{19} = Z_{29} = 1,$$

and all other Z 's are zero. With this notation,

$$R'_p = Z_{p1} + Z_{p2} + \cdots + Z_{pn} \quad (12)$$

is the number of runs of length $\geq p$, and

$$R_p = R'_p - R'_{p+1} \quad (13)$$

is the number of runs of length p exactly. Our goal is to compute the mean value of R_p , and also the covariance

$$\text{covar}(R_p, R_q) = \text{mean}((R_p - \text{mean}(R_p))(R_q - \text{mean}(R_q))),$$

which measures the interdependence of R_p and R_q . These mean values are to be computed as the average over the set of all $n!$ permutations.

Equations (12) and (13) show that the answers can be expressed in terms of the mean values of Z_{pi} and of $Z_{pi}Z_{qj}$, so as the first step of the derivation we obtain the following results (assuming that $i < j$):

$$\begin{aligned} \frac{1}{n!} \sum Z_{pi} &= \begin{cases} (p + \delta_{i1})/(p + 1)! & \text{if } i \leq n - p + 1; \\ 0 & \text{otherwise.} \end{cases} \\ \frac{1}{n!} \sum Z_{pi}Z_{pj} &= \begin{cases} (p + \delta_{i1})q/(p + 1)!(q + 1)! & \text{if } i + p < j \leq n - q + 1; \\ (p + \delta_{i1})/(p + 1)!q! - (p + q + \delta_{i1})/(p + q + 1)! & \text{if } i + p = j \leq n - q + 1; \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \quad (14)$$

The \sum -signs stand for a summation over all possible permutations. To illustrate the calculations involved here, we will work the most difficult case, when $i + p = j \leq n - q + 1$, and when $i > 1$. Note that $Z_{pi}Z_{qj}$ is either zero or one, so the summation consists of counting all permutations $U_1U_2\dots U_n$ for which $Z_{pi} = Z_{qj} = 1$, that is, all permutations such that

$$U_{i-1} > U_i < \cdots < U_{i+p-1} > U_{i+p} < \cdots < U_{i+p+q-1}. \quad (15)$$

The number of such permutations may be enumerated as follows: there are $\binom{n}{p+q+1}$ ways to choose the elements for the positions indicated in (15); there

are

$$(p+q+1)\binom{p+q}{p} - \binom{p+q+1}{p+1} - \binom{p+q+1}{1} + 1 \quad (16)$$

ways to arrange them in the order (15), as shown in exercise 13; and there are $(n-p-q-1)!$ ways to arrange the remaining elements. Thus there are $\binom{n}{p+q+1}(n-p-q-1)!$ times (16) ways in all, and dividing by $n!$ we get the desired formula.

From relations (14) a rather lengthy calculation leads to

$$\begin{aligned} \text{mean}(R'_p) &= \text{mean}(Z_{p1} + \dots + Z_{pn}) \\ &= (n+1)p/(p+1)! - (p-1)/p!, \quad 1 \leq p \leq n; \end{aligned} \quad (17)$$

$$\begin{aligned} \text{covar}(R'_p, R'_q) &= \text{mean}(R'_p R'_q) - \text{mean}(R'_p) \text{mean}(R'_q) \\ &= \sum_{1 \leq i, j \leq n} \frac{1}{n!} \sum Z_{pi} Z_{pj} - \text{mean}(R'_p) \text{mean}(R'_q) \\ &= \begin{cases} \text{mean}(R'_t) + f(p, q, n), & \text{if } p+q \leq n, \\ \text{mean}(R'_t) - \text{mean}(R'_p) \text{mean}(R'_q), & \text{if } p+q > n, \end{cases} \end{aligned} \quad (18)$$

where $t = \max(p, q)$, $s = p+q$, and

$$\begin{aligned} f(p, q, n) &= (n+1) \left(\frac{s(1-pq) + pq}{(p+1)!(q+1)!} - \frac{2s}{(s+1)!} \right) + 2 \left(\frac{s-1}{s!} \right) \\ &\quad + \frac{(s^2 - s - 2)pq - s^2 - p^2q^2 + 1}{(p+1)!(q+1)!}. \end{aligned} \quad (19)$$

This expression for the covariance is unfortunately quite complicated, but it is necessary for a successful run test as described above. From these formulas it is easy to compute

$$\begin{aligned} \text{mean}(R_p) &= \text{mean}(R'_p) - \text{mean}(R'_{p+1}), \\ \text{covar}(R_p, R'_q) &= \text{covar}(R'_p, R'_q) - \text{covar}(R'_{p+1}, R'_q), \\ \text{covar}(R_p, R_q) &= \text{covar}(R_p, R'_q) - \text{covar}(R_p, R'_{q+1}). \end{aligned} \quad (20)$$

In *Annals Math. Stat.* 15 (1944), 163–165, J. Wolfowitz proved that the quantities $R_1, R_2, \dots, R_{t-1}, R'_t$ become normally distributed as $n \rightarrow \infty$, subject to the mean and covariance expressed above; this implies that the following test for runs is valid: Given a sequence of n random numbers, compute the number of runs R_p of length p for $1 \leq p < t$, and also the number of runs R'_t of length t or more. Let

$$\begin{aligned} Q_1 &= R_1 - \text{mean}(R_1), \quad \dots, \quad Q_{t-1} = R_{t-1} - \text{mean}(R_{t-1}), \\ Q_t &= R'_t - \text{mean}(R'_t). \end{aligned} \quad (21)$$

Form the matrix C of the covariances of the R 's; for example, $C_{13} = \text{covar}(R_1, R_3)$, while $C_{1t} = \text{covar}(R_1, R'_t)$. When $t = 6$, we have

$$C = nC_1 + C_2$$

$$= n \begin{pmatrix} \frac{23}{180} & \frac{-7}{360} & \frac{-5}{336} & \frac{-433}{60480} & \frac{-13}{5670} & \frac{-121}{181440} \\ \frac{-7}{360} & \frac{2843}{20160} & \frac{-989}{20160} & \frac{-7159}{362880} & \frac{-10019}{1814400} & \frac{-1303}{907200} \\ \frac{-5}{336} & \frac{-989}{20160} & \frac{54563}{907200} & \frac{-21311}{1814400} & \frac{-62369}{19958400} & \frac{-7783}{9979200} \\ \frac{-433}{60480} & \frac{-7159}{362880} & \frac{-21311}{1814400} & \frac{886657}{39916800} & \frac{-257699}{239500800} & \frac{-62611}{239500800} \\ \frac{-13}{5670} & \frac{-10019}{1814400} & \frac{-62369}{19958400} & \frac{-257699}{239500800} & \frac{29874811}{5448643200} & \frac{-1407179}{21794572800} \\ \frac{-121}{181440} & \frac{-1303}{907200} & \frac{-7783}{9979200} & \frac{-62611}{239500800} & \frac{-1407179}{21794572800} & \frac{2134697}{1816214400} \end{pmatrix}$$

$$+ \begin{pmatrix} \frac{83}{180} & \frac{-29}{180} & \frac{-11}{210} & \frac{-41}{12096} & \frac{91}{25920} & \frac{41}{18144} \\ \frac{-29}{180} & \frac{-305}{4032} & \frac{319}{20160} & \frac{2557}{72576} & \frac{10177}{604800} & \frac{413}{64800} \\ \frac{-11}{210} & \frac{319}{20160} & \frac{-58747}{907200} & \frac{19703}{604800} & \frac{239471}{19958400} & \frac{39517}{9979200} \\ \frac{-41}{12096} & \frac{2557}{72576} & \frac{19703}{604800} & \frac{-220837}{4435200} & \frac{1196401}{239500800} & \frac{360989}{239500800} \\ \frac{91}{25920} & \frac{10177}{604800} & \frac{239471}{19958400} & \frac{1196401}{239500800} & \frac{139126639}{7264857600} & \frac{4577641}{10897286400} \\ \frac{41}{18144} & \frac{413}{64800} & \frac{39517}{9979200} & \frac{360989}{239500800} & \frac{4577641}{10897286400} & \frac{-122953057}{21794572800} \end{pmatrix} \quad (22)$$

if $n \geq 12$. Now form $A = (a_{ij})$, the inverse of the matrix C , and compute $\sum_{1 \leq i, j \leq t} Q_i Q_j a_{ij}$. The result for large n should have approximately the chi-square distribution with t degrees of freedom.

The matrix (11) given earlier is the inverse of C_1 to five significant figures. When n is large, A will be approximately $(1/n)C_1^{-1}$; a test with $n = 100$ showed that the entries a_{ij} in (11) were each about 1 percent lower than the true values obtained by inverting (22).

H. Maximum-of- t test. For $0 \leq j < n$, let $V_j = \max(U_{tj}, U_{tj+1}, \dots, U_{tj+t-1})$. Now apply the Kolmogorov-Smirnov test to the sequence V_0, V_1, \dots, V_{n-1} , with the distribution function $F(x) = x^t$, $0 \leq x \leq 1$. Alternatively, apply the equidistribution test to the sequence $V_0^t, V_1^t, \dots, V_{n-1}^t$.

To verify this test, we must show that the distribution function for the V_j is $F(x) = x^t$. The probability that $\max(U_1, U_2, \dots, U_t) \leq x$ is the probability that $U_1 \leq x$ and $U_2 \leq x$ and ... and $U_t \leq x$, which is the product of the individual probabilities, namely $xx\dots x = x^t$.

I. Collision test. Chi-square tests can be made only when there is a nontrivial number of items expected in each category. But there is another kind of test that can be used when the number of categories is much larger than the number of observations; this test is related to "hashing," an important method for information retrieval that we shall study in Chapter 6.

Suppose we have m urns and we throw n balls at random into those urns, where m is much greater than n . Most of the balls will land in urns that were previously empty, but if a ball falls into an urn that already contains at least one ball we say that a “collision” has occurred. The collision test counts the number of collisions, and a generator passes this test if it doesn’t induce too many or too few collisions.

To fix the ideas, suppose $m = 2^{20}$ and $n = 2^{14}$. Then each urn will receive only one 64th of a ball, on the average. The probability that a given urn will contain exactly k balls is $p_k = \binom{n}{k} m^{-k} (1 - m^{-1})^{n-k}$, so the expected number of collisions per urn is $\sum_{k \geq 1} (k-1)p_k = \sum_{k \geq 0} kp_k - \sum_{k \geq 1} p_k = n/m - 1 + p_0$. Since $p_0 = (1 - m^{-1})^n = 1 - n/m + \binom{n}{2} m^{-2} + \text{smaller terms}$, we find that the average total number of collisions taken over all m urns is very slightly less than $n^2/2m = 128$.

We can use the collision test to rate a random number generator in a large number of dimensions. For example, when $m = 2^{20}$ and $n = 2^{14}$ we can test the 20-dimensional randomness of a number generator by letting $d = 2$ and forming 20-dimensional vectors $V_j = (Y_{20j}, Y_{20j+1}, \dots, Y_{20j+19})$ for $0 \leq j < n$. It suffices to keep a table of $m = 2^{20}$ bits to determine collisions, one bit for each possible value of the vector V_j ; on a computer with 32 bits per word, this amounts to 2^{15} words. Initially all 2^{20} bits of this table are cleared to zero; then for each V_j , if the corresponding bit is already 1 we record a collision, otherwise we set the bit to 1. This test can also be used in 10 dimensions with $d = 4$, and so on.

To decide if the test is passed, we can use the following table of percentage points when $m = 2^{20}$ and $n = 2^{14}$:

collisions \leq	101	108	119	126	134	145	153
with probability	.009	.043	.244	.476	.742	.946	.989

The theory underlying these probabilities is the same we used in the poker test, Eq. (5); the probability that c collisions occur is the probability that $n - c$ urns are occupied, namely

$$\frac{m(m-1)\dots(m-n+c+1)}{m^n} \begin{Bmatrix} n \\ n-c \end{Bmatrix}.$$

Although m and n are very large, it is not difficult to compute these probabilities using the following method:

Algorithm S (Percentage points for collision test). Given m and n , this algorithm determines the distribution of the number of collisions that occur when n balls are scattered into m urns. An auxiliary array $A[0], A[1], \dots, A[n]$ of floating point numbers is used for the computation; actually $A[j]$ will be nonzero only for $j_0 \leq j \leq j_1$, and $j_1 - j_0$ will be at most of order $\log n$, so it would be possible to get by with considerably less storage.

- S1.** [Initialize.] Set $A[j] \leftarrow 0$ for $0 \leq j \leq n$; then set $A[1] \leftarrow 1$ and $j_0 \leftarrow j_1 \leftarrow 1$. Then do step S2 exactly $n - 1$ times and go on to step S3.
- S2.** [Update probabilities.] (Each time we do this step it corresponds to tossing a ball into an urn; $A[j]$ represents the probability that exactly j of the urns are occupied.) Set $j_1 \leftarrow j_1 + 1$. Then for $j \leftarrow j_1, j_1 - 1, \dots, j_0$ (in this order), set $A[j] \leftarrow (j/m)A[j] + ((1 + 1/m) - (j/m))A[j - 1]$. If $A[j]$ has become very small as a result of this calculation, say $A[j] < 10^{-20}$, set $A[j] \leftarrow 0$; and in such a case, if $j = j_1$ decrease j_1 by 1, or if $j = j_0$ increase j_0 by 1.
- S3.** [Compute the answers.] In this step we make use of an auxiliary table $(T_1, T_2, \dots, T_{t_{\max}}) = (.01, .05, .25, .50, .75, .95, .99, 1.00)$ containing the specified percentage points of interest. Set $p \leftarrow 0$, $t \leftarrow 1$, and $j \leftarrow j_0 - 1$. Do the following iteration until $t = t_{\max}$: Increase j by 1, and set $p \leftarrow p + A[j]$; then if $p > T_t$, output $n - j - 1$ and $1 - p$ (meaning that with probability $1 - p$ there are at most $n - j - 1$ collisions) and repeatedly increase t by 1 until $p \leq T_t$. ■

J. Serial correlation test. We may also compute the following statistic:

$$C = \frac{n(U_0U_1 + U_1U_2 + \dots + U_{n-2}U_{n-1} + U_{n-1}U_0) - (U_0 + U_1 + \dots + U_{n-1})^2}{n(U_0^2 + U_1^2 + \dots + U_{n-1}^2) - (U_0 + U_1 + \dots + U_{n-1})^2}. \quad (23)$$

This is the “serial correlation coefficient,” which is a measure of the amount U_{j+1} depends on U_j .

Correlation coefficients appear frequently in statistics; if we have n quantities U_0, U_1, \dots, U_{n-1} and n others V_0, V_1, \dots, V_{n-1} , the correlation coefficient between them is defined to be

$$C = \frac{n \sum (U_j V_j) - (\sum U_j)(\sum V_j)}{\sqrt{(n \sum U_j^2 - (\sum U_j)^2)(n \sum V_j^2 - (\sum V_j)^2)}}. \quad (24)$$

All summations in this formula are to be taken over the range $0 \leq j < n$; Eq. (23) is the special case $V_j = U_{(j+1) \bmod n}$. (Note: The denominator of (24) is zero when $U_0 = U_1 = \dots = U_{n-1}$ or $V_0 = V_1 = \dots = V_{n-1}$; we exclude this case from discussion.)

A correlation coefficient always lies between -1 and $+1$. When it is zero or very small, it indicates that the quantities U_j and V_j are (relatively speaking) independent of each other, but when the correlation coefficient is ± 1 it indicates total linear dependence; in fact $V_j = \alpha \pm \beta U_j$ for all j in such a case, for some constants α and β . (See exercise 17.)

Therefore it is desirable to have C in Eq. (23) close to zero. In actual fact, since U_0U_1 is not completely independent of U_1U_2 , the serial correlation

coefficient is not expected to be exactly zero. (See exercise 18.) A "good" value of C will be between $\mu_n - 2\sigma_n$ and $\mu_n + 2\sigma_n$, where

$$\mu_n = \frac{-1}{n-1}, \quad \sigma_n = \frac{1}{n-1} \sqrt{\frac{n(n-3)}{n+1}}, \quad n > 2. \quad (25)$$

We expect C to be between these limits about 95 percent of the time.

Equations (25) are only conjectured at this time, since the exact distribution of C is not known when the U 's are uniformly distributed. For the theory when the U 's have the *normal* distribution, see the paper by Wilfrid J. Dixon, *Annals Math. Stat.* 15 (1944), 119–144. Empirical evidence suggests that we may safely use the formulas for the mean and standard deviation that have been derived from the assumption of the normal distribution, without much error; these are the values that have been listed in (25). It is known that $\lim_{n \rightarrow \infty} \sqrt{n}\sigma_n = 1$; cf. the article by Anderson and Walker, *Annals Math. Stat.* 35 (1964), 1296–1303, where more general results about serial correlations of dependent sequences are derived.

Instead of simply computing the correlation coefficient between the observations $(U_0, U_1, \dots, U_{n-1})$ and their immediate successors $(U_1, \dots, U_{n-1}, U_0)$, we can also compute it between $(U_0, U_1, \dots, U_{n-1})$ and any cyclically shifted sequence $(U_q, \dots, U_{n-1}, U_0, \dots, U_{q-1})$; the cyclic correlations should be small for $0 < q < n$. A straightforward computation of Eq. (24) for all q would require about n^2 multiplications, but it is actually possible to compute all the correlations in only $O(n \log n)$ steps by using "fast Fourier transforms." (See Section 4.6.4; cf. also L. P. Schmid, *CACM* 8 (1965), 115.)

K. Tests on subsequences. It frequently happens that the external program using our random sequence will call for numbers in batches. For example, if the program works with three random variables X , Y , and Z , it may consistently invoke the generation of three random numbers at a time. In such applications it is important that the subsequences consisting of every *third* term of the original sequence be random. If the program requires q numbers at a time, the sequences

$$U_0, U_q, U_{2q}, \dots; \quad U_1, U_{q+1}, U_{2q+1}, \dots; \quad \dots; \quad U_{q-1}, U_{2q-1}, U_{3q-1}, \dots$$

can each be put through the tests described above for the original sequence U_0, U_1, U_2, \dots

Experience with linear congruential sequences has shown that these derived sequences rarely if ever behave less randomly than the original sequence, unless q has a large factor in common with the period length. On a binary computer with m equal to the word size, for example, a test of the subsequences for $q = 8$ will tend to give the poorest randomness for all $q < 16$; and on a decimal computer, $q = 10$ yields the subsequences most likely to be unsatisfactory. (This can be explained somewhat on the grounds of potency, since such values of q will tend to lower the potency.)

L. Historical remarks and further discussion. Statistical tests arose naturally in the course of scientists' efforts to "prove" or "disprove" hypotheses about various observed data. The best known early papers dealing with the testing of artificially generated numbers for randomness are two articles by M. G. Kendall and B. Babington-Smith in the *Journal of the Royal Statistical Society* 101 (1938), 147–166, and in the supplement to that journal, 6 (1939), 51–61. These papers were concerned with the testing of random digits between 0 and 9, rather than random real numbers; for this purpose, the authors discussed the frequency test, serial test, gap test, and poker test, although they misapplied the serial test. Kendall and Babington-Smith also used a variant of the coupon collector's test; the method described in this section was introduced by R. E. Greenwood in *Math. Comp.* 9 (1955), 1–4.

The run test has a rather interesting history. Originally, tests were made on runs up and down at once: a run up would be followed by a run down, then another run up, and so on. Note that the run test and the permutation test do not depend on the uniform distribution of the U 's, they depend only on the fact that $U_i = U_j$ occurs with probability zero when $i \neq j$; therefore these tests can be applied to many types of random sequences. The run test in primitive form was originated by J. Bienaymé [*Comptes Rendus* 81 (Paris: Acad. Sciences, 1875), 417–423]. Some sixty years later, W. O. Kermack and A. G. McKendrick published two extensive papers on the subject (*Proc. Royal Society Edinburgh* 57 (1937), 228–240, 332–376]; as an example they stated that Edinburgh rainfall between the years 1785 and 1930 was "entirely random in character" with respect to the run test (although they examined only the mean and standard deviation of the run lengths). Several other people began using the test, but it was not until 1944 that the use of the chi-square method in connection with this test was shown to be incorrect. The paper by H. Levene and J. Wolfowitz in *Annals Math. Stat.* 15 (1944), 58–69, introduced the correct run test (for runs up and down, alternately) and discussed the fallacies in earlier misuses of that test. Separate tests for runs up and runs down, as proposed in the text above, are more suited to computer application, so we have not given the more complex formulas for the alternate-up-and-down case. See the survey paper by D. E. Barton and C. L. Mallows, *Annals Math. Stat.* 36 (1965), 236–260.

Of all the tests we have discussed, the frequency test and the serial correlation test seem to be the weakest, in the sense that nearly all random number generators pass these tests. Theoretical grounds for the weakness of these tests are discussed briefly in Section 3.5 (cf. exercise 3.5–26). The run test, on the other hand, is a rather strong test: the results of exercises 3.3.3–23 and 24 suggest that linear congruential generators tend to have runs somewhat longer than normal if the multiplier is not large enough, so the run test of exercise 14 is definitely to be recommended.

The collision test is also highly recommended, since it has been especially designed to detect the deficiencies of many poor generators that have unfortunately become widespread. This test, which is based on ideas of H. Delgas Christiansen [Inst. Math. Stat. and Oper. Res., Tech. Univ. Denmark (Oct. 1975),

unpublished], is unlike the others in that it was not developed before the advent of computers; it is specifically intended for computer use.

The reader probably wonders, "Why are there so many tests?" It has been said that more computer time is spent testing random numbers than using them in applications! This is untrue, although it is possible to go overboard in testing.

The need for making several tests has been amply documented. It has been recorded, for example, that some numbers generated by a variant of the middle-square method have passed the frequency test, gap test, and poker test, yet flunked the serial test. Linear congruential sequences with small multipliers have been known to pass many tests, yet fail on the run test because there are too few runs of length one. The maximum-of- t test has also been used to ferret out some bad generators that otherwise seemed to perform respectably.

Perhaps the main reason for doing extensive testing on random number generators is that people misusing Mr. X's random number generator will hardly ever admit that their programs are at fault: they will blame the generator, until Mr. X can prove to them that his numbers are sufficiently random. On the other hand, if the source of random numbers is only for Mr. X's personal use, he might decide not to bother to test them, since the techniques recommended in this chapter have a high probability of being satisfactory.

EXERCISES

1. [10] Why should the serial test described in part B be applied to $(Y_0, Y_1), (Y_2, Y_3), \dots, (Y_{2n-2}, Y_{2n-1})$ instead of to $(Y_0, Y_1), (Y_1, Y_2), \dots, (Y_{n-1}, Y_n)$?
2. [10] State an appropriate way to generalize the serial test to triples, quadruples, etc., instead of pairs.
- 3. [M20] How many U 's need to be examined in the gap test (Algorithm G) before n gaps have been found, on the average, assuming that the sequence is random? What is the standard deviation of this quantity?
4. [12] Prove that the probabilities in (4) are correct for the gap test.
5. [M23] The "classical" gap test used by Kendall and Babington-Smith considers the numbers U_0, U_1, \dots, U_{N-1} to be a cyclic sequence with U_{N+j} identified with U_j . Here N is a fixed number of U 's that are to be subjected to the test. If n of the numbers U_0, \dots, U_{N-1} fall into the range $\alpha \leq U_j < \beta$, there are n gaps in the cyclic sequence. Let Z_r be the number of gaps of length r , for $0 \leq r < t$, and let Z_t be the number of gaps of length $\geq t$; show that the quantity $V = \sum_{0 \leq r \leq t} (Z_r - np_r)^2 / np_r$ should have the chi-square distribution with t degrees of freedom, in the limit as N goes to infinity, where p_r is given in Eq. (4).
6. [40] (H. Geiringer.) A frequency count of the first 2000 decimal digits in the representation of $e = 2.71828\dots$ gave a χ^2 value of 1.06, indicating that the actual frequencies of the digits 0, 1, ..., 9 are much too close to their expected values to be considered randomly distributed. (In fact, $\chi^2 \geq 1.15$ with probability 99.9 percent.) The same test applied to the first 10,000 digits of e gives the reasonable value $\chi^2 = 8.61$; but the fact that the first 2000 digits are so evenly distributed is still surprising. Does the same phenomenon occur in the representation of e to other bases? [See *AMM* 72 (1965), 483–500.]

7. [08] Apply the coupon collector's test procedure (Algorithm C) with $d = 3$ and $n = 7$, to the following sequence: 1101221022120202001212201010201121. What lengths do the seven subsequences have?

► 8. [M22] How many U 's need to be examined, on the average, in the coupon collector's test (Algorithm C) before n complete sets have been found, assuming that the sequence is random? What is the standard deviation? [Hint: See Eq. 1.2.9-28.]

9. [M21] Generalize the coupon collector's test so that the search stops as soon as w distinct values have been found, where w is a fixed positive integer less than or equal to d . What probabilities should be used in place of (6)?

10. [M23] Solve exercise 8 for the more general coupon collector's test described in exercise 9.

11. [00] The "runs up" in a particular permutation are displayed in (9); what are the "runs down" in that permutation?

12. [20] Let U_0, U_1, \dots, U_{n-1} be n distinct numbers. Write an algorithm that determines the lengths of all ascending runs in the sequence. When your algorithm terminates, COUNT[r] should be the number of runs of length r , for $1 \leq r \leq 5$, and COUNT[6] should be the number of runs of length 6 or more.

13. [M23] Show that (16) is the number of permutations of $p+q+1$ distinct elements having the pattern (15).

► 14. [M15] If we "throw away" the element that immediately follows a run, so that when X_j is greater than X_{j+1} we start the next run with X_{j+2} , the run lengths are independent, and a simple chi-square test may be used (instead of the horribly complicated method derived in the text). What are the appropriate run-length probabilities for this simple run test?

15. [M10] In the maximum-of- t test, why are $V_0^t, V_1^t, \dots, V_{n-1}^t$ supposed to be uniformly distributed between zero and one?

► 16. [15] (a) Mr. J. H. Quick (a student) wanted to perform the maximum-of- t test for various values of t . Letting $Z_{jt} = \max(U_j, U_{j+1}, \dots, U_{j+t-1})$, he found a clever way to go from the sequence $Z_{0(t-1)}, Z_{1(t-1)}, \dots$, to the sequence Z_{0t}, Z_{1t}, \dots , using very little time and space. What was his bright idea?

(b) He decided to modify the maximum-of- t method so that the j th observation would be $\max(U_j, \dots, U_{j+t-1})$; in other words, he took $V_j = Z_{jt}$ instead of $V_j = Z_{(tj)t}$ as the text says. He reasoned that *all* of the Z 's should have the same distribution, so the test is even stronger if each Z_{jt} , $0 \leq j < n$, is used instead of just every t th one. But when he tried a chi-square equidistribution test on the values of V_j^t , he got extremely high values of the statistic V , which got even higher as t increased. Why did this happen?

17. [M25] (a) Given any numbers $U_0, \dots, U_{n-1}, V_0, \dots, V_{n-1}$, let

$$\bar{u} = \frac{1}{n} \sum_{0 \leq k < n} U_k, \quad \bar{v} = \frac{1}{n} \sum_{0 \leq k < n} V_k.$$

Let $U'_k = U_k - \bar{u}$, $V'_k = V_k - \bar{v}$. Show that the correlation coefficient C given in Eq. (24) is equal to

$$\sum_{0 \leq k < n} U'_k V'_k / \sqrt{\sum_{0 \leq k < n} {U'_k}^2} \sqrt{\sum_{0 \leq k < n} {V'_k}^2}.$$

(b) Let $C = N/D$, where N and D denote the numerator and denominator of the expression in part (a). Show that $N^2 \leq D^2$, hence $-1 \leq C \leq 1$; and obtain a formula for the difference $D^2 - N^2$. [Hint: See exercise 1.2.3–30.]

(c) If $C = \pm 1$, show that $\alpha X_k + \beta Y_k = \tau$, $0 \leq k < n$, for some constants α , β , and τ , not all zero.

18. [M20] (a) Show that if $n = 2$, the serial correlation coefficient (23) is always equal to -1 (unless the denominator is zero). (b) Similarly, show that when $n = 3$, the serial correlation coefficient always equals $-\frac{1}{2}$. (c) Show that the denominator in (23) is zero if and only if $U_0 = U_1 = \dots = U_{n-1}$.

19. [M40] What are the mean and standard deviation of the serial correlation coefficient (23) when $n = 4$ and the U 's are independent and uniformly distributed between zero and one?

20. [M47] Find the distribution of the serial correlation coefficient (23), for general n , assuming that the U_j are independent random variables uniformly distributed between zero and one.

21. [19] What value of f is computed by Algorithm P if it is presented with the permutation $(1, 2, 9, 8, 5, 3, 6, 7, 0, 4)$?

22. [18] For what permutation of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ will Algorithm P produce the value $f = 1024$?

*3.3.3. Theoretical Tests

Although it is always possible to test a random number generator using the methods in the previous section, it is far better to have “*a priori* tests,” i.e., theoretical results that tell us in advance how well those tests will come out. Such theoretical results give us much more understanding about the generation methods than empirical, “trial-and-error” results do. In this section we shall study the linear congruential sequences in more detail; if we know what the results of certain tests will be before we actually generate the numbers, we have a better chance of choosing a , m , and c properly.

The development of this kind of theory is quite difficult, although some progress has been made. The results obtained so far are generally for statistical tests made over the entire period. Not all statistical tests make sense when they are applied over a full period—for example, the equidistribution test will give results that are too perfect—but the serial test, gap test, permutation test, maximum test, etc. can be fruitfully analyzed in this way. Such studies will detect global nonrandomness of a sequence, i.e., improper behavior in very large samples.

The theory we shall discuss is quite illuminating, but it does not eliminate the need for testing local nonrandomness by the methods of Section 3.3.2. Indeed, it appears to be extremely hard to prove anything useful about short subsequences. Only a few theoretical results are known about the behavior of linear congruential sequences over less than a full period; these will be discussed at the end of Section 3.3.4. (See also exercise 18.)

Let us begin with a proof of a simple *a priori* law, for the least complicated case of the permutation test. The gist of our first theorem is that we have $X_{n+1} < X_n$ about half the time, provided that the sequence has high potency.

Theorem P. *Let a , c , and m generate a linear congruential sequence with maximum period; let $b = a - 1$ and let d be the greatest common divisor of m and b . The probability that $X_{n+1} < X_n$ is equal to $\frac{1}{2} + r$, where*

$$r = (2(c \bmod d) - d)/2m; \quad (1)$$

hence $|r| < d/2m$.

Proof. The proof of this theorem involves some techniques that are of interest in themselves. First we define

$$s(x) = (ax + c) \bmod m. \quad (2)$$

Thus, $X_{n+1} = s(X_n)$, and the theorem reduces to counting the number of integers x such that $0 \leq x < m$ and $s(x) < x$ (since each such integer occurs somewhere in the period). We want to show that this number is

$$\frac{1}{2}(m + 2(c \bmod d) - d). \quad (3)$$

The function $[(x - s(x))/m]$ is equal to 1 when $x > s(x)$, and it is 0 otherwise; hence the count we wish to obtain can be written simply as

$$\begin{aligned} \sum_{0 \leq x < m} \left\lceil \frac{x - s(x)}{m} \right\rceil &= \sum_{0 \leq x < m} \left\lceil \frac{x}{m} - \left(\frac{ax + c}{m} - \left\lfloor \frac{ax + c}{m} \right\rfloor \right) \right\rceil \\ &= \sum_{0 \leq x < m} \left(\left\lfloor \frac{ax + c}{m} \right\rfloor - \left\lfloor \frac{bx + c}{m} \right\rfloor \right). \end{aligned} \quad (4)$$

(Recall that $[-y] = -[y]$ and $b = a - 1$.) Such sums can be evaluated by the method of exercise 1.2.4-37, where we have proved that

$$\sum_{0 \leq j < k} \left\lfloor \frac{hj + c}{k} \right\rfloor = \frac{(h-1)(k-1)}{2} + \frac{g-1}{2} + g[c/g], \quad g = \gcd(h, k), \quad (5)$$

whenever h and k are integers and $k > 0$. Since a is relatively prime to m , this formula yields

$$\begin{aligned} \sum_{0 \leq x < m} \left\lfloor \frac{ax + c}{m} \right\rfloor &= \frac{(a-1)(m-1)}{2} + c, \\ \sum_{0 \leq x < m} \left\lfloor \frac{bx + c}{m} \right\rfloor &= \frac{(b-1)(m-1)}{2} + \frac{d-1}{2} + c - (c \bmod d), \end{aligned}$$

and (3) follows immediately. ■

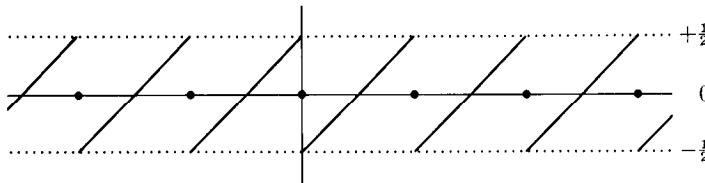


Fig. 7. The sawtooth function $((z))$.

The proof of Theorem P indicates that *a priori* tests can indeed be carried out, provided that we are able to deal satisfactorily with sums involving the $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ functions. In many cases the most powerful technique for dealing with floor and ceiling functions is to replace them by two somewhat more symmetrical ones:

$$\delta(z) = \lfloor z \rfloor + 1 - \lceil z \rceil = \begin{cases} 1, & \text{if } z \text{ is an integer;} \\ 0, & \text{if } z \text{ is not an integer;} \end{cases} \quad (6)$$

$$((z)) = z - \lfloor z \rfloor - \frac{1}{2} + \frac{1}{2}\delta(z) = z - \lceil z \rceil + \frac{1}{2} - \frac{1}{2}\delta(z). \quad (7)$$

The latter function is a “sawtooth” function familiar in the study of Fourier series; its graph is shown in Fig. 7. The reason for choosing to work with $((z))$ rather than $\lfloor z \rfloor$ or $\lceil z \rceil$ is that $((z))$ possesses several very useful properties:

$$((-z)) = -((z)); \quad (8)$$

$$((z+n)) = ((z)), \quad \text{integer } n; \quad (9)$$

$$((nz)) = ((z)) + \left(\left(z + \frac{1}{n} \right) \right) + \cdots + \left(\left(z + \frac{n-1}{n} \right) \right), \quad \text{integer } n \geq 1. \quad (10)$$

(See exercise 2.)

In order to get some practice working with these functions, let us prove Theorem P again, this time without relying on exercise 1.2.4–37. With the help of Eqs. (7), (8), (9), we can show that

$$\begin{aligned} \left\lceil \frac{x-s(x)}{m} \right\rceil &= \frac{x-s(x)}{m} - \left(\left(\frac{x-s(x)}{m} \right) \right) + \frac{1}{2} - \frac{1}{2}\delta\left(\frac{x-s(x)}{m}\right) \\ &= \frac{x-s(x)}{m} - \left(\left(\frac{x-(ax+c)}{m} \right) \right) + \frac{1}{2} \\ &= \frac{x-s(x)}{m} + \left(\left(\frac{bx+c}{m} \right) \right) + \frac{1}{2} \end{aligned} \quad (11)$$

since $(x-s(x))/m$ is never an integer. Now

$$\sum_{0 \leq x < m} \frac{x-s(x)}{m} = 0$$

since both x and $s(x)$ take on each value of $\{0, 1, \dots, m-1\}$ exactly once; hence (11) yields

$$\sum_{0 \leq x < m} \left\lceil \frac{x - s(x)}{m} \right\rceil = \sum_{0 \leq x < m} \left(\left(\frac{bx + c}{m} \right) \right) + \frac{m}{2}. \quad (12)$$

Let $b = b_0 d$, $m = m_0 d$, where b_0 and m_0 are relatively prime. We know that $(b_0 x) \bmod m_0$ takes on the values $\{0, 1, \dots, m_0 - 1\}$ in some order as x varies from 0 to $m_0 - 1$. By (9) and (10) and the fact that

$$\left(\left(\frac{b(x + m_0) + c}{m} \right) \right) = \left(\left(\frac{bx + c}{m} \right) \right)$$

we have

$$\begin{aligned} \sum_{0 \leq x < m} \left(\left(\frac{bx + c}{m} \right) \right) &= d \sum_{0 \leq x \leq m_0} \left(\left(\frac{bx + c}{m} \right) \right) \\ &= d \sum_{0 \leq x < m_0} \left(\left(\frac{c}{m} + \frac{b_0 x}{m} \right) \right) = d \left(\left(\frac{c}{d} \right) \right). \end{aligned} \quad (13)$$

Theorem P follows immediately from (12) and (13).

One consequence of Theorem P is that practically any choice of a and c will give a reasonable probability that $X_{n+1} < X_n$, at least over the entire period, except those that have large d . A large value of d corresponds to low potency, and we already know that generators of low potency are undesirable.

The next theorem gives us a more stringent condition for the choice of a and c ; we will consider the *serial correlation test* applied over the entire period. The quantity C defined in Section 3.3.2, Eq. (23), is

$$C = \left(m \sum_{0 \leq x < m} xs(x) - \left(\sum_{0 \leq x < m} x \right)^2 \right) / \left(m \sum_{0 \leq x < m} x^2 - \left(\sum_{0 \leq x < m} x \right)^2 \right). \quad (14)$$

Let x' be the element such that $s(x') = 0$. We have

$$s(x) = m \left(\left(\frac{ax + c}{m} \right) \right) + \frac{m}{2}, \quad \text{if } x \neq x'. \quad (15)$$

The formulas we are about to derive can be expressed most easily in terms of the function

$$\sigma(h, k, c) = 12 \sum_{0 \leq j < k} \left(\left(\frac{j}{k} \right) \right) \left(\left(\frac{hj + c}{k} \right) \right), \quad (16)$$

an important function that arises in several mathematical problems. It is called a *generalized Dedekind sum*, since Richard Dedekind introduced the function $\sigma(h, k, 0)$ in 1876 when commenting on one of Riemann's incomplete manuscripts. [See B. Riemann's *Gesammelte Math. Werke*, 2nd ed. (1892), 466–478.]

Using the well-known formulas

$$\sum_{0 \leq x < m} x = \frac{m(m-1)}{2} \quad \text{and} \quad \sum_{0 \leq x < m} x^2 = \frac{m(m-1)(2m-1)}{6},$$

it is a straightforward matter to transform Eq. (14) into

$$C = \frac{m\sigma(a, m, c) - 3 + 6(m - x' - c)}{m^2 - 1}. \quad (17)$$

(See exercise 5.) Since m is usually very large, we may discard terms of order $1/m$, and we have the approximation

$$C \approx \sigma(a, m, c)/m, \quad (18)$$

with an error of less than $6/m$ in absolute value.

The serial correlation test now reduces to determining the value of the Dedekind sum $\sigma(a, m, c)$. Evaluating $\sigma(a, m, c)$ directly from its definition (16) is hardly any easier than evaluating the correlation coefficient itself directly, but fortunately there are simple methods available for computing Dedekind sums quite rapidly.

Lemma B ("Reciprocity law" for Dedekind sums). *Let h, k, c be integers. If $0 \leq c < k$, $0 < h \leq k$, and if h is relatively prime to k , then*

$$\sigma(h, k, c) + \sigma(k, h, c) = \frac{h}{k} + \frac{k}{h} + \frac{1}{hk} + \frac{6c^2}{hk} - 6 \left\lfloor \frac{c}{h} \right\rfloor - 3e(h, c), \quad (19)$$

where

$$e(h, c) = \begin{cases} 1, & \text{if } c = 0 \text{ or } c \bmod h \neq 0; \\ 0, & \text{if } c > 0 \text{ and } c \bmod h = 0. \end{cases} \quad (20)$$

Proof. We leave it to the reader to prove that, under these hypotheses,

$$\sigma(h, k, c) + \sigma(k, h, c) = \sigma(h, k, 0) + \sigma(k, h, 0) + \frac{6c^2}{hk} - 6 \left\lfloor \frac{c}{h} \right\rfloor - 3e(h, c) + 3. \quad (21)$$

(See exercise 6.) The lemma now must be proved only in the case $c = 0$.

The proof we will give, based on complex roots of unity, is essentially due to L. Carlitz. There is actually a simpler proof that uses only elementary manipulations of sums (see exercise 7)—but the following method reveals more of the mathematical tools that are available for problems of this kind and it is therefore much more instructive.

Let $f(x)$ and $g(x)$ be polynomials defined as follows:

$$\begin{aligned} f(x) &= 1 + x + \cdots + x^{k-1} = (x^k - 1)/(x - 1) \\ g(x) &= x + 2x^2 + \cdots + (k-1)x^{k-1} = xf'(x) \\ &= kx^k/(x-1) - x(x^k - 1)/(x-1)^2. \end{aligned} \quad (22)$$

If ω is the complex k th root of unity $e^{2\pi i/k}$, we have by Eq. 1.2.9-13

$$\frac{1}{k} \sum_{0 \leq j < k} \omega^{-jr} g(\omega^j x) = rx^r, \quad \text{if } 0 \leq r < k. \quad (23)$$

Set $x = 1$; then $g(\omega^j x) = k/(\omega^j - 1)$ if $j \neq 0$, otherwise it equals $k(k-1)/2$, therefore

$$r \bmod k = \sum_{0 < j < k} \frac{\omega^{-jr}}{\omega^j - 1} + \frac{1}{2}(k-1), \quad \text{if } r \text{ is an integer.}$$

(Eq. (23) shows that the right-hand side equals r when $0 \leq r < k$, and it is unchanged when multiples of k are added to r .) Hence

$$\left(\left(\frac{r}{k} \right) \right) = \frac{1}{k} \sum_{0 < j < k} \frac{\omega^{-jr}}{\omega^j - 1} - \frac{1}{2k} + \frac{1}{2} \delta\left(\frac{r}{k}\right). \quad (24)$$

This important formula, which holds whenever r is an integer, allows us to reduce many calculations involving $((r/k))$ to sums involving k th roots of unity, and it brings a whole new range of techniques into the picture. In particular, we get the following formula:

$$\sigma(h, k, 0) + \frac{3(k-1)}{k^2} = \frac{12}{k^2} \sum_{0 < r < k} \sum_{0 < i < k} \sum_{0 < j < k} \frac{\omega^{-ir}}{\omega^i - 1} \frac{\omega^{-jhr}}{\omega^j - 1}. \quad (25)$$

The right-hand side of this formula may be simplified by carrying out the sum on r ; we have $\sum_{0 \leq r < k} \omega^{rs} = f(\omega^s) = 0$ if $s \bmod k \neq 0$. Equation (25) now reduces to

$$\sigma(h, k, 0) + \frac{3(k-1)}{k} = \frac{12}{k} \sum_{0 < j < k} \frac{1}{(\omega^{-jh} - 1)(\omega^j - 1)}. \quad (26)$$

A similar formula is obtained for $\sigma(k, h, 0)$, with $\zeta = e^{2\pi i/h}$ replacing ω .

It is not obvious what we can do with the sum in (26), but there is an elegant way to proceed, based on the fact that each term of the sum is a function of ω^j , where $0 < j < k$; hence the sum is essentially taken over the k th roots of unity other than 1. Whenever x_1, x_2, \dots, x_n are distinct complex numbers, we have the identity

$$\begin{aligned} \sum_{1 \leq j \leq n} \frac{1}{(x_j - x_1) \dots (x_j - x_{j-1})(x - x_j)(x_j - x_{j+1}) \dots (x_j - x_n)} \\ = \frac{1}{(x - x_1) \dots (x - x_n)}, \end{aligned} \quad (27)$$

which follows from the usual method of expanding the right-hand side into partial fractions. Moreover, if $q(x) = (x - y_1)(x - y_2)\dots(x - y_m)$, we have

$$q'(y_j) = (y_j - y_1)\dots(y_j - y_{j-1})(y_j - y_{j+1})\dots(y_j - y_m); \quad (28)$$

this identity may often be used to simplify expressions like those in the left-hand side of (27). When h and k are relatively prime, the numbers $\omega, \omega^2, \dots, \omega^{k-1}, \zeta, \zeta^2, \dots, \zeta^{h-1}$ are all distinct; we can therefore consider formula (27) in the special case of the polynomial $(x - \omega)\dots(x - \omega^{k-1})(x - \zeta)\dots(x - \zeta^{h-1}) = (x^k - 1)(x^h - 1)/(x - 1)^2$, obtaining the following identity in x :

$$\frac{1}{h} \sum_{0 < j < h} \frac{\zeta^j(\zeta^j - 1)^2}{(\zeta^{jk} - 1)(x - \zeta^j)} + \frac{1}{k} \sum_{0 < j < k} \frac{\omega^j(\omega^j - 1)^2}{(\omega^{jh} - 1)(x - \omega^j)} = \frac{(x - 1)^2}{(x^h - 1)(x^k - 1)}. \quad (29)$$

This identity has many interesting consequences, and it leads to numerous reciprocity formulas for sums of the type given in Eq. (26). For example, if we differentiate (29) twice with respect to x and let $x \rightarrow 1$, we find that

$$\begin{aligned} \frac{2}{h} \sum_{0 < j < h} \frac{\zeta^j(\zeta^j - 1)^2}{(\zeta^{jk} - 1)(1 - \zeta^j)^3} + \frac{2}{k} \sum_{0 < j < k} \frac{\omega^j(\omega^j - 1)^2}{(\omega^{jh} - 1)(1 - \omega^j)^3} \\ = \frac{1}{6} \left(\frac{h}{k} + \frac{k}{h} + \frac{1}{hk} \right) + \frac{1}{2} - \frac{1}{2h} - \frac{1}{2k}. \end{aligned}$$

Replace j by $h - j$ and by $k - j$ in these sums and use (26) to get

$$\begin{aligned} \frac{1}{6} \left(\sigma(k, h, 0) + \frac{3(h-1)}{h} \right) + \frac{1}{6} \left(\sigma(h, k, 0) + \frac{3(k-1)}{k} \right) \\ = \frac{1}{6} \left(\frac{h}{k} + \frac{k}{h} + \frac{1}{hk} \right) + \frac{1}{2} - \frac{1}{2h} - \frac{1}{2k}, \end{aligned}$$

which is equivalent to the desired result. ■

Lemma B gives us an explicit function $f(h, k, c)$ such that

$$\sigma(h, k, c) = f(h, k, c) - \sigma(k, h, c) \quad (30)$$

whenever $0 < h \leq k$, $0 \leq c < k$, and h is relatively prime to k . From the definition (16) it is clear that

$$\sigma(k, h, c) = \sigma(k \bmod h, h, c \bmod h). \quad (31)$$

Therefore we can use (30) iteratively to evaluate $\sigma(h, k, c)$, using a process that reduces the parameters as in Euclid's algorithm.

Further simplifications occur when we examine this iterative procedure more closely. Let us set $m_1 = k$, $m_2 = h$, $c_1 = c$, and form the following tableau:

$$\begin{array}{ll} m_1 = a_1 m_2 + m_3 & c_1 = b_1 m_2 + c_2 \\ m_2 = a_2 m_3 + m_4 & c_2 = b_2 m_3 + c_3 \\ m_3 = a_3 m_4 + m_5 & c_3 = b_3 m_4 + c_4 \\ m_4 = a_4 m_5 & c_4 = b_4 m_5 + c_5 \end{array} \quad (32)$$

Here

$$\begin{aligned} a_j &= \lfloor m_j/m_{j+1} \rfloor, & b_j &= \lfloor c_j/m_{j+1} \rfloor, \\ m_{j+2} &= m_j \bmod m_{j+1}, & c_{j+1} &= c_j \bmod m_{j+1}, \end{aligned} \quad (33)$$

and it follows that

$$0 \leq m_{j+1} < m_j, \quad 0 \leq c_j < m_j. \quad (34)$$

We have assumed for convenience that Euclid's algorithm terminates in (32) after four iterations; this assumption will reveal the pattern that holds in the general case. Since h and k were relatively prime to start with, we must have $m_5 = 1$ and $c_5 = 0$ in (32).

Let us further assume that $c_3 \neq 0$ but $c_4 = 0$, in order to get a feeling for the effect this has on the recurrence. Equations (30) and (31) yield

$$\begin{aligned} \sigma(h, k, c) &= \sigma(m_2, m_1, c_1) \\ &= f(m_2, m_1, c_1) - \sigma(m_3, m_2, c_2) \\ &= \dots \\ &= f(m_2, m_1, c_1) - f(m_3, m_2, c_2) + f(m_4, m_3, c_3) - f(m_5, m_4, c_4). \end{aligned}$$

The first part " $h/k + k/h$ " of the formula for $f(h, k, c)$ in (19) contributes

$$\frac{m_2}{m_1} + \frac{m_1}{m_2} - \frac{m_3}{m_2} - \frac{m_2}{m_3} + \frac{m_4}{m_3} + \frac{m_3}{m_4} - \frac{m_5}{m_4} - \frac{m_4}{m_5}$$

to the total, and this simplifies to

$$\begin{aligned} \frac{h}{k} + \left(a_1 + \frac{m_3}{m_2} \right) - \frac{m_3}{m_2} - \left(a_2 + \frac{m_4}{m_3} \right) + \frac{m_4}{m_3} + \left(a_3 + \frac{m_5}{m_4} \right) - \frac{m_5}{m_4} - a_4 \\ = h/k + a_1 - a_2 + a_3 - a_4. \end{aligned}$$

The next part " $1/hk$ " of (19) also leads to a simple contribution; according to Eq. 4.5.3-9 and other formulas in Section 4.5.3, we have

$$1/m_1m_2 - 1/m_2m_3 + 1/m_3m_4 - 1/m_4m_5 = h'/k - 1, \quad (35)$$

where h' is the unique integer satisfying

$$h'h \equiv 1 \pmod{k}, \quad 0 < h' \leq k. \quad (36)$$

Adding up all the contributions, and remembering our assumption that $c_4 = 0$ (so that $e(m_4, c_3) = 0$, cf. (20)), we find that

$$\begin{aligned} \sigma(h, k, c) &= \frac{h + h'}{k} + (a_1 - a_2 + a_3 - a_4) - 6(b_1 - b_2 + b_3 - b_4) \\ &\quad + 6\left(\frac{c_1^2}{m_1m_2} - \frac{c_2^2}{m_2m_3} + \frac{c_3^2}{m_3m_4} - \frac{c_4^2}{m_4m_5}\right) + 2, \end{aligned}$$

in terms of the assumed tableau (32). Similar results hold in general:

Theorem D. Let h, k, c be integers with $0 < h \leq k$, $0 \leq c < k$, and h relatively prime to k . Form the “Euclidean tableau” as defined in (33) above, and assume that the process stops after t steps with $m_{t+1} = 1$. Let s be the smallest subscript such that $c_s = 0$, and let h' be defined by (36). Then

$$\begin{aligned}\sigma(h, k, c) = \frac{h + h'}{k} + \sum_{1 \leq j \leq t} (-1)^{j+1} \left(a_j - 6b_j + 6 \frac{c_j^2}{m_j m_{j+1}} \right) \\ + 3((-1)^s + \delta_{s1}) - 2 + (-1)^t.\end{aligned}\blacksquare$$

Euclid’s algorithm is analyzed carefully in Section 4.5.3; the quantities a_1, a_2, \dots, a_t are called the *partial quotients* of h/k . Theorem 4.5.3F tells us that the number of iterations, t , will never exceed $\log_\phi k$; hence Dedekind sums can be evaluated rapidly. The terms $c_j^2/m_j m_{j+1}$ can be simplified further, and an efficient algorithm for evaluating $\sigma(h, k, c)$ appears in exercise 17.

Now that we have analyzed generalized Dedekind sums, let us apply our knowledge to the determination of serial correlation coefficients.

Example 1. Find the serial correlation when $m = 2^{35}$, $a = 2^{34} + 1$, $c = 1$.

Solution. We have

$$C = (2^{35}\sigma(2^{34} + 1, 2^{35}, 1) - 3 + 6(2^{35} - (2^{34} - 1) - 1))/(2^{70} - 1)$$

by Eq. (17). To evaluate $\sigma(2^{34} + 1, 2^{35}, 1)$, we can form the tableau

$$\begin{array}{llll} m_1 = 2^{35} & & c_1 = 1 \\ m_2 = 2^{34} + 1 & a_1 = 1 & c_2 = 1 & b_1 = 0 \\ m_3 = 2^{34} - 1 & a_2 = 1 & c_3 = 1 & b_2 = 0 \\ m_4 = 2 & a_3 = 2^{33} - 1 & c_4 = 1 & b_3 = 0 \\ m_5 = 1 & a_4 = 2 & c_5 = 0 & b_4 = 1 \end{array}$$

Since $h' = 2^{34} + 1$, the value according to Theorem D comes to $2^{33} - 3 + 2^{-32}$. Thus

$$C = (2^{68} + 5)/(2^{70} - 1) = \frac{1}{4} + \epsilon, \quad |\epsilon| < 2^{-67}. \quad (37)$$

Such a correlation is much, much too high for randomness. Of course, this generator has very low potency, and we have already rejected it as nonrandom.

Example 2. Find the approximate serial correlation when $m = 10^{10}$, $a = 10001$, $c = 2113248653$.

Solution. We have $C \approx \sigma(a, m, c)/m$, and the computation proceeds as follows:

$$\begin{array}{llllll} m_1 = 10000000000 & & & c_1 = 2113248653 \\ m_2 = 10001 & a_1 = 999900 & c_2 = 7350 & b_1 = 211303 \\ m_3 = 100 & a_2 = 100 & c_3 = 50 & b_2 = 73 \\ m_4 = 1 & a_3 = 100 & c_4 = 0 & b_3 = 50 \end{array}$$

$$\begin{aligned}\sigma(m_2, m_1, c_1) &= -31.6926653544; \\ C &\approx -3 \cdot 10^{-9}.\end{aligned}\quad (38)$$

This is a very respectable value of C indeed. But the generator has a potency of only 3, so it is not really a very good source of random numbers in spite of the fact that it has low serial correlation. It is necessary to have a low serial correlation, but not sufficient.

Example 3. Estimate the serial correlation for general a , m , and c .

Solution. If we consider just one application of (30), we have

$$\sigma(a, m, c) \approx \frac{m}{a} + 6 \frac{c^2}{am} - 6 \frac{c}{a} - \sigma(m, a, c).$$

Now $|\sigma(m, a, c)| < a$ by exercise 12, and therefore

$$C \approx \frac{\sigma(a, m, c)}{m} \approx \frac{1}{a} \left(1 - 6 \frac{c}{m} + 6 \left(\frac{c}{m} \right)^2 \right). \quad (39)$$

The error in this approximation is less than $(a + 6)/m$ in absolute value.

The estimate in (39) was the first theoretical result known about the randomness of congruential generators. R. R. Coveyou [JACM 7 (1960), 72–74] obtained it by averaging over all real numbers x between 0 and m instead of considering only the integer values (cf. exercise 21); then Martin Greenberger [Math. Comp. 15 (1961), 383–389] gave a rigorous derivation including an estimate of the error term.

So began one of the saddest chapters in the history of computer science! Although the above approximation is quite correct, it has been grievously misapplied in practice; people abandoned the perfectly good generators they had been using and replaced them by terrible generators that looked good from the standpoint of (39). For more than a decade, the most common random number generators in daily use were seriously deficient, solely because of a theoretical advance. A little knowledge is a dangerous thing.

If we are to learn by past mistakes, we had better look carefully at how (39) has been misused. In the first place people assumed uncritically that a small serial correlation over the whole period would be a pretty good guarantee of randomness; but in fact it doesn't even ensure a small serial correlation for 1000 consecutive elements of the sequence (see exercise 14).

Secondly, (39) and its error term will ensure a relatively small value of C only when $a \approx \sqrt{m}$; therefore people suggested choosing multipliers near \sqrt{m} . In fact, we shall see that nearly all multipliers give a value of C that is substantially less than $1/\sqrt{m}$, hence (39) is not a very good approximation to the true behavior. Minimizing a crude upper bound for C does not minimize C .

In the third place, people observed that (39) yields its best estimate when $c/m \approx \frac{1}{2} \pm \frac{1}{6}\sqrt{3}$, since these values are the roots of $1 - 6x + 6x^2 = 0$. "In the absence of any other criterion for choosing c , we might as well use this one." The latter statement is not incorrect, but it is misleading at best, since experience has shown that the value of c has hardly any influence on the true value of the serial

correlation when a is a good multiplier; the choice $c/m \approx \frac{1}{2} \pm \frac{1}{6}\sqrt{3}$ reduces C substantially only in cases like Example 2 above. And we are fooling ourselves in such cases, since the bad multiplier will reveal its deficiencies in other ways.

Clearly we need a better estimate than (39); and such an estimate is now available thanks to Theorem D, which stems principally from the work of U. Dieter [Math. Comp. 25 (1971), 855–883]. Theorem D implies that $\sigma(a, m, c)$ will be small if the partial quotients of a/m are small. Indeed, by analyzing generalized Dedekind sums still more closely, it is possible to obtain quite a sharp estimate:

Theorem K. Under the assumptions of Theorem D, we always have

$$-\frac{1}{2} \sum_{\substack{1 \leq j \leq t \\ j \text{ odd}}} a_j - \sum_{\substack{1 \leq j \leq t \\ j \text{ even}}} a_j + \frac{1}{2} \leq \sigma(h, k, c) \leq \sum_{\substack{1 \leq j \leq t \\ j \text{ odd}}} a_j + \frac{1}{2} \sum_{\substack{1 \leq j \leq t \\ j \text{ even}}} a_j - \frac{1}{2}. \quad (40)$$

Proof. See D. E. Knuth, *Acta Arithmetica* 33 (1978), 297–325, where it is shown further that these bounds are essentially the best possible when there are large partial quotients. ■

Example 4. Estimate the serial correlation for $a = 3141592621$, $m = 2^{35}$, c odd.

Solution. The partial quotients of a/m are 10, 1, 14, 1, 7, 1, 1, 1, 3, 3, 3, 5, 2, 1, 8, 7, 1, 4, 1, 2, 4, 2; hence by Theorem K

$$-45 \leq \sigma(a, m, c) \leq 68,$$

and the serial correlation is guaranteed to be extremely low for all c .

Note that this bound is considerably better than we could obtain from (39), since the error in (39) is of order a/m ; our “random” multiplier has turned out to be much better than one specifically chosen to look good on the basis of (39). In fact, it is possible to show that the average value of $\sum_{1 \leq j \leq t} a_j$, taken over all multipliers a relatively prime to m , is

$$\frac{6}{\pi^2} (\ln m)^2 + O((\log m)(\log \log m)^4)$$

(see exercise 4.5.3–35). Therefore the probability that a random multiplier has large $\sum_{1 \leq j \leq t} a_j$, say larger than $(\log m)^{2+\epsilon}$ for some fixed $\epsilon > 0$, approaches zero as $m \rightarrow \infty$. This substantiates the empirical evidence that almost all linear congruential sequences have extremely low serial correlation over the entire period.

The exercises below show that other *a priori* tests, such as the serial test over the entire period, can also be expressed in terms of a few generalized Dedekind sums. It follows from Theorem K that linear congruential sequences will pass these tests provided that certain specified fractions (depending on a and m but

not on c) have small partial quotients. In particular, the result of exercise 19 implies that the serial test on pairs will be satisfactorily passed if and only if a/m has no large partial quotients.

The book *Dedekind Sums* by Hans Rademacher and Emil Grosswald (Math. Assoc. of America, Carus Monograph No. 16, 1972) discusses the history and properties of Dedekind sums and their generalizations. Further theoretical tests, including the serial test in higher dimensions, are discussed in Section 3.3.4.

EXERCISES—First Set

1. [M10] Express " $x \bmod y$ " in terms of the sawtooth and δ functions.
2. [M20] Prove the "replicative law," Eq. (10).
3. [HM22] What is the Fourier series expansion (in terms of sines and cosines) of the function $f(x) = ((x))$?
- 4. [M19] If $m = 10^{10}$, what is the highest possible value of d (in the notation of Theorem P), given that the potency of the generator is 10?
5. [M21] Carry out the derivation of Eq. (17).
6. [M27] Let $hh' + kk' = 1$. (a) Show, without using Lemma B, that

$$\sigma(h, k, c) = \sigma(h, k, 0) + 12 \sum_{0 < j < c} \left(\left(\frac{h'j}{k} \right) \right) + 6 \left(\left(\frac{h'c}{k} \right) \right)$$

for all integers $c \geq 0$. (b) Show that if $0 < j < k$,

$$\left(\left(\frac{h'j}{k} \right) \right) + \left(\left(\frac{k'j}{h} \right) \right) = \frac{j}{hk} - \frac{1}{2} \delta \left(\frac{j}{h} \right).$$

- (c) Under the assumptions of Lemma B, prove Eq. (21).
- 7. [M24] Give a proof of the reciprocity law (19), when $c = 0$, by using the general reciprocity law of exercise 1.2.4–45.
 - 8. [M34] (L. Carlitz.) Let

$$\rho(p, q, r) = 12 \sum_{0 \leq j < r} \left(\left(\frac{jp}{r} \right) \right) \left(\left(\frac{jq}{r} \right) \right).$$

By generalizing the method of proof used in Lemma B, prove the following beautiful identity due to H. Rademacher: If each of p, q, r is relatively prime to the other two,

$$\rho(p, q, r) + \rho(q, r, p) + \rho(r, p, q) = \frac{p}{qr} + \frac{q}{rp} + \frac{r}{pq} - 3.$$

(The reciprocity law for Dedekind sums, with $c = 0$, is the special case $r = 1$.)

9. [M40] Is there a simple proof of Rademacher's identity (exercise 8) along the lines of the proof in exercise 7 of a special case?

10. [M20] Show that when $0 < h < k$ it is possible to express $\sigma(k - h, k, c)$ and $\sigma(h, k, -c)$ easily in terms of $\sigma(h, k, c)$.

11. [M30] The formulas given in the text show us how to evaluate $\sigma(h, k, c)$ when h and k are relatively prime and c is an integer. For the general case, prove that

a) $\sigma(dh, dk, dc) = \sigma(h, k, c)$, integer $d > 0$;

b) $\sigma(h, k, c + \theta) = \sigma(h, k, c) + 6((h'c/k))$, integer c , real $0 < \theta < 1$, when h and k are relatively prime and $hh' \equiv 1$ (modulo k).

12. [M24] Show that if h is relatively prime to k and c is an integer, $|\sigma(h, k, c)| \leq (k-1)(k-2)/k$.

13. [M24] Generalize Eq. (26) so that it gives an expression for $\sigma(h, k, c)$.

► **14.** [M20] The linear congruential generator that has $m = 2^{35}$, $a = 2^{18} + 1$, $c = 1$, was given the serial correlation test on three batches of 1000 consecutive numbers, and the result was a very high correlation, between 0.2 and 0.3, in each case. What is the serial correlation of this generator, taken over all 2^{35} numbers of the period?

15. [M21] Generalize Lemma B so that it applies to all *real* values of c , $0 \leq c < k$.

16. [M24] Given the Euclidean tableau defined in (33), let $p_0 = 1$, $p_1 = a_1$, and $p_j = a_j p_{j-1} + p_{j-2}$ for $1 < j \leq t$. Show that the complicated portion of the sum in Theorem D can be rewritten as follows, making it possible to avoid noninteger computations:

$$\sum_{1 \leq j \leq t} (-1)^{j+1} \frac{c_j^2}{m_j m_{j+1}} = \frac{1}{m_1} \sum_{1 \leq j \leq t} (-1)^{j+1} b_j (c_j + c_{j+1}) p_{j-1}.$$

[Hint: Prove that we have $\sum_{1 \leq j \leq r} (-1)^{j+1}/m_j m_{j+1} = (-1)^{r+1} p_{r-1}/m_1 m_{r+1}$ for $1 \leq r \leq t$.]

17. [M22] Design an algorithm that evaluates $\sigma(h, k, c)$ for integers h, k, c satisfying the hypotheses of Theorem D. Your algorithm should use only integer arithmetic (of unlimited precision), and it should produce the answer in the form $A + B/k$ where A and B are integers. (Cf. exercise 16.) If possible, use only a finite number of variables for temporary storage, instead of maintaining arrays such as a_1, a_2, \dots, a_t .

► **18.** [M23] (U. Dieter.) Given positive integers h, k, z , let

$$S(h, k, c, z) = \sum_{0 \leq j < z} \left(\left(\frac{hj + c}{k} \right) \right).$$

Show that this sum can be evaluated in “closed form,” in terms of generalized Dedekind sums and the sawtooth function. [Hint: When $z \leq k$, the quantity $\lfloor j/k \rfloor - \lfloor (j-z)/k \rfloor$ equals 1 for $0 \leq j < z$, and it equals 0 for $z \leq j < k$, so we can introduce this factor and sum over $0 \leq j < k$.]

► **19.** [M23] Show that the serial test can be analyzed over the full period, in terms of generalized Dedekind sums, by finding a formula for the probability that $\alpha \leq X_n < \beta$ and $\alpha' \leq X_{n+1} < \beta'$ when $\alpha, \beta, \alpha', \beta'$ are given integers with $0 \leq \alpha < \beta \leq m$, $0 \leq \alpha' < \beta' \leq m$. [Hint: Consider the quantity $\lfloor (x-\alpha)/m \rfloor - \lfloor (x-\beta)/m \rfloor$.]

20. [M29] (U. Dieter.) Extend Theorem P by obtaining a formula for the probability that $X_n > X_{n+1} > X_{n+2}$, in terms of generalized Dedekind sums.

EXERCISES—Second Set

In many cases, exact computations with integers are quite difficult to carry out, but we can attempt to study the probabilities that arise when we take the average over all real values of x instead of restricting the calculation to integer values. Although these results are only approximate, they shed some light on the subject.

It is convenient to deal with numbers U_n between zero and one; for linear congruential sequences, $U_n = X_n/m$, and we have $U_{n+1} = \{ax + \theta\}$, where $\theta = c/m$ and $\{x\}$ denotes $x \bmod 1$. For example, the formula for serial correlation now becomes

$$C = \left(\int_0^1 x\{ax + \theta\} dx - \left(\int_0^1 x dx \right)^2 \right) / \left(\int_0^1 x^2 dx - \left(\int_0^1 x dx \right)^2 \right).$$

- 21. [HM28] (R. R. Coveyou.) What is the value of C in the formula just given?
- 22. [M22] Let a be an integer, and let $0 \leq \theta < 1$. If x is a real number between 0 and 1, and if $s(x) = \{ax + \theta\}$, what is the probability that $s(x) < x$? (This is the “real number” analog of Theorem P.)
- 23. [28] The previous exercise gives the probability that $U_{n+1} < U_n$. What is the probability that $U_{n+2} < U_{n+1} < U_n$, assuming that U_n is a random real number between zero and one?
- 24. [M29] Under the assumptions of the preceding problem, except with $\theta = 0$, show that $U_n > U_{n+1} > \dots > U_{n+t-1}$ occurs with probability

$$\frac{1}{t!} \left(1 + \frac{1}{a}\right) \dots \left(1 + \frac{t-2}{a}\right).$$

What is the average length of a descending run starting at U_n , assuming that U_n is selected at random between zero and one?

- 25. [M25] Let $\alpha, \beta, \alpha', \beta'$ be real numbers with $0 \leq \alpha < \beta \leq 1, 0 \leq \alpha' < \beta' \leq 1$. Under the assumptions of exercise 22, what is the probability that $\alpha \leq x < \beta$ and $\alpha' \leq s(x) < \beta'$? (This is the “real number” analog of exercise 19.)
- 26. [M21] Consider a “Fibonacci” generator, where $U_{n+1} = \{U_n + U_{n-1}\}$. Assuming that U_1 and U_2 are independently chosen at random between 0 and 1, find the probability that $U_1 < U_2 < U_3, U_1 < U_3 < U_2, U_2 < U_1 < U_3$, etc. [Hint: Divide the “unit square,” i.e., the points of the plane $\{(x, y) \mid 0 \leq x, y < 1\}$, into six parts, depending on the relative order of x, y , and $\{x + y\}$, and determine the area of each part.]
- 27. [M32] In the Fibonacci generator of the preceding exercise, let U_0 and U_1 be chosen independently in the unit square except that $U_0 > U_1$. Determine the probability that U_1 is the beginning of an upward run of length k , so that $U_0 > U_1 < \dots < U_k > U_{k+1}$. Compare this with the corresponding probabilities for a random sequence.
- 28. [M35] According to Eq. 3.2.1.3–5, a linear congruential generator with potency 2 satisfies the condition $X_{n-1} - 2X_n + X_{n+1} \equiv (a-1)c \pmod{m}$. Consider a generator that abstracts this situation: let $U_{n+1} = \{\alpha + 2U_n - U_{n-1}\}$. As in exercise 26, divide the unit square into parts that show the relative order of U_1, U_2 , and U_3 for each pair (U_1, U_2) . Are there any values of α for which all six possible orders are achieved with probability $\frac{1}{6}$, assuming that U_1 and U_2 are chosen at random in the unit square?

3.3.4. The Spectral Test

In this section we shall study an especially important way to check the quality of linear congruential random number generators; not only do all good generators pass this test, all generators now known to be bad actually *fail* it. Thus it is by far the most powerful test known, and it deserves particular attention. Our discussion will also bring out some fundamental limitations on the degree of randomness we can expect from linear congruential sequences and their generalizations.

The spectral test embodies aspects of both the empirical and theoretical tests studied in previous sections: it is like the theoretical tests because it deals with properties of the full period of the sequence, and it is like the empirical tests because it requires a computer program to determine the results.

A. Ideas underlying the test. The most important randomness criteria seem to rely on properties of the joint distribution of t consecutive elements of the sequence, and the spectral test deals directly with this distribution. If we have a sequence $\langle U_n \rangle$ of period m , the idea is to analyze the set of all m points

$$\{(U_n, U_{n+1}, \dots, U_{n+t-1})\} \quad (1)$$

in t -dimensional space.

For simplicity we shall assume that we have a linear congruential sequence (X_0, a, c, m) of maximum period length m (so that $c \neq 0$), or that m is prime and $c = 0$ and the period length is $m - 1$. In the latter case we shall add the point $(0, 0, \dots, 0)$ to the set (1), so that there are always m points in all; this extra point has a negligible effect when m is large, and it makes the theory much simpler. Under these assumptions, (1) can be rewritten as

$$\left\{ \frac{1}{m}(x, s(x), s(s(x)), \dots, s^{t-1}(x)) \mid 0 \leq x < m \right\}, \quad (2)$$

where

$$s(x) = (ax + c) \bmod m \quad (3)$$

is the “successor” of x . Note that we are considering only the set of all such points in t dimensions, not the order in which those points are actually generated. But the order of generation is reflected in the dependence between components of the vectors; and the spectral test studies such dependence for various dimensions t by dealing with the totality of all points (2).

For example, Fig. 8 shows a typical small case in 2 and 3 dimensions, for the generator with

$$s(x) = (137x + 187) \bmod 256. \quad (4)$$

Of course a generator with period length 256 will hardly be random, but 256 is small enough that we can draw the diagram and gain some understanding before we turn to the larger m 's that are of practical interest.

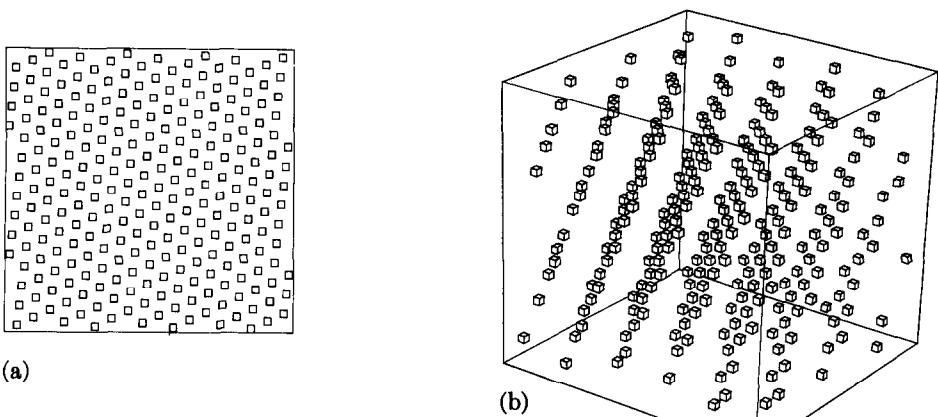


Fig. 8. (a) The two-dimensional grid formed by all pairs of successive points (X_n, X_{n+1}) , when $X_{n+1} = (137X_n + 187) \bmod 256$.

(b) The three-dimensional grid of triplets (X_n, X_{n+1}, X_{n+2}) . [Illustrations courtesy of Bruce G. Baumgart.]

Perhaps the most striking thing about the pattern of boxes in Fig. 8 is that we can cover them all by a fairly small number of parallel lines; indeed, there are many different families of parallel lines that will hit all the points. For example, a set of 20 nearly vertical lines will do the job, as will a set of 21 lines that tilt upward at roughly a 30° angle. We commonly observe similar patterns when driving past farmlands that have been planted in a systematic manner.

If the same generator is considered in three dimensions, we obtain 256 points in a cube, obtained by appending a “height” component $s(s(x))$ to each of the 256 points $(x, s(x))$ in the plane of Fig. 8(a), as shown in Fig. 8(b). Let’s imagine that this 3-D crystal structure has been made into a physical model, a cube that we can turn in our hands; as we rotate it, we will notice various families of parallel planes that encompass all of the points. In the words of Wallace Givens, the random numbers stay “mainly in the planes.”

At first glance we might think that such systematic behavior is so nonrandom as to make congruential generators quite worthless; but more careful reflection, remembering that m is quite large in practice, provides a better insight. The regular structure in Fig. 8 is essentially the “grain” we see when examining our random numbers under a high-power microscope. If we take truly random numbers between 0 and 1, and round or truncate them to finite accuracy so that each is an integer multiple of $1/\nu$ for some given number ν , then the t -dimensional points (1) we obtain will have an extremely regular character when viewed through a microscope.

Let $1/\nu_2$ be the maximum distance between lines, taken over all families of parallel straight lines that cover the points $\{(x/m, s(x)/m)\}$ in two dimensions. We shall call ν_2 the two-dimensional accuracy of the random number generator, since the pairs of successive numbers have a fine structure that is

essentially good to one part in ν_2 . Similarly, let $1/\nu_3$ be the maximum distance between planes, taken over all families of parallel planes that cover all points $\{(x/m, s(x)/m, s(s(x))/m)\}$; we shall call ν_3 the accuracy in three dimensions. The t -dimensional accuracy ν_t is the reciprocal of the maximum distance between hyperplanes, taken over all families of parallel $(t - 1)$ -dimensional hyperplanes that cover all points $\{(x/m, s(x)/m, \dots, s^{t-1}(x)/m)\}$.

The essential difference between periodic sequences and truly random sequences that have been truncated to multiples of $1/\nu$ is that the “accuracy” of truly random sequences is the same in all dimensions, while the “accuracy” of periodic sequences decreases as t increases. Indeed, since there are only m points in the t -dimensional cube when m is the period length, we can’t achieve a t -dimensional accuracy of more than about $m^{1/t}$.

When the independence of t consecutive values is considered, computer-generated random numbers will behave essentially as if we took truly random numbers and truncated them to $\lg \nu_t$ bits, where ν_t decreases with increasing t . In practice, such varying accuracy is usually all we need. We don’t insist that the 10-dimensional accuracy be 2^{35} , in the sense that all $(2^{35})^{10}$ possible 10-tuples $(U_n, U_{n+1}, \dots, U_{n+9})$ should be equally likely on a 35-bit machine; for such large values of t we want only a few of the leading bits of $(U_n, U_{n+1}, \dots, U_{n+t-1})$ to behave as if they were independently random.

On the other hand when an application demands high resolution of the random number sequence, simple linear congruential sequences will necessarily be inadequate; a generator with larger period should be used instead, even though only a small fraction of the period will actually be generated. Squaring the period will essentially square the accuracy in higher dimensions, i.e., it will double the effective number of bits of precision.

The spectral test is based on the values of ν_t for small t , say $2 \leq t \leq 6$. Dimensions 2, 3, and 4 seem to be adequate to detect important deficiencies in a sequence, but since we are considering the entire period it seems best to be somewhat cautious and go up into another dimension or two; on the other hand the values of ν_t for $t \geq 10$ seem to be of no practical significance whatever. (This is fortunate, because it appears to be rather difficult to calculate ν_t when $t \geq 10$.)

Note that there is a vague relation between the spectral test and the serial test; for example, a special case of the serial test, taken over the entire period as in exercise 3.3.3–19, counts the number of boxes in each of 64 subsquares of Fig. 8(a). The main difference is that the spectral test rotates the dots so as to discover the least favorable orientation. We shall return to a consideration of the serial test later in this section.

It may appear at first that we should apply the spectral test only for one suitably high value of t ; if a generator passes the test in three dimensions, it seems plausible that it should also pass the 2-D test, hence we might as well omit the latter. The fallacy in this reasoning occurs because we apply more stringent conditions in lower dimensions. A similar situation occurs with the serial test: Consider a generator that (quite properly) has almost the same number of points

in each subcube of the unit cube, when the unit cube has been divided into 64 subcubes of size $\frac{1}{4} \times \frac{1}{4} \times \frac{1}{4}$; this same generator might yield completely empty subsquares of the unit square, when the unit square has been divided into 64 subsquares of size $\frac{1}{8} \times \frac{1}{8}$. Since we increase our expectations in lower dimensions, a separate test for each dimension is required.

It is not always true that $\nu_t \leq m^{1/t}$, although this upper bound is valid when the points form a rectangular grid. For example, it turns out that $\nu_2 = \sqrt{274} > \sqrt{256}$ in Fig. 8, because a nearly hexagonal structure brings the m points closer together than would be possible in a strictly rectangular arrangement.

In order to develop an algorithm that computes ν_t efficiently, we must look more deeply at the associated mathematical theory. Therefore a reader who is not mathematically inclined is advised to skip to part D of this section, where the spectral test is presented as a “plug-in” method accompanied by several examples. On the other hand, we shall see that the mathematics behind the spectral test requires only some elementary manipulations of vectors.

Some authors have suggested using the minimum number N_t of parallel covering lines or hyperplanes as the criterion, instead of the maximum distance $1/\nu_t$ between them. However, this number does not appear to be as important as the concept of accuracy defined above, because it is biased by how nearly the slope of the lines or hyperplanes matches the coordinate axes of the cube. For example, the 20 nearly vertical lines that cover all the points of Fig. 8 are actually $1/\sqrt{328}$ units apart, and this might falsely imply an accuracy of one part in $\sqrt{328}$, or perhaps even of one part in 20. The true accuracy of only one part in $\sqrt{274}$ is realized only for the larger family of 21 lines with a slope of $7/15$; another family of 24 lines, with a slope of $-11/13$, also has a greater inter-line distance than the 20-line family, since $1/\sqrt{290} > 1/\sqrt{328}$. The precise way in which families of lines act at the boundaries of the unit hypercube does not seem to be an especially “clean” or significant criterion; however, for those people who prefer to count hyperplanes, it is possible to compute N_t using a method quite similar to the way in which we shall calculate ν_t (see exercise 16).

***B. Theory behind the test.** In order to analyze the basic set (2), we start with the observation that

$$\frac{1}{m} s^j(x) = \left(\frac{a^j x + (1 + a + \cdots + a^{j-1})c}{m} \right) \bmod 1. \quad (5)$$

We can get rid of the “mod 1” operation by extending the set periodically, making infinitely many copies of the original t -dimensional hypercube, proceeding in all directions. This gives us the set

$$\begin{aligned} L &= \left\{ \left(\frac{x}{m} + k_1, \frac{s(x)}{m} + k_2, \dots, \frac{s^{t-1}(x)}{m} + k_t \right) \mid \text{integer } x, k_1, k_2, \dots, k_t \right\} \\ &= \left\{ V_0 + \left(\frac{x}{m} + k_1, \frac{ax}{m} + k_2, \dots, \frac{a^{t-1}x}{m} + k_t \right) \mid \text{integer } x, k_1, k_2, \dots, k_t \right\}, \end{aligned}$$

where

$$V_0 = \frac{1}{m}(0, c, (1+a)c, \dots, (1+a+\dots+a^{t-2})c) \quad (6)$$

is a constant vector. The variable k_1 is redundant in this representation of L , because we can change $(x, k_1, k_2, \dots, k_t)$ to $(x+k_1m, 0, k_2-ak_1, \dots, k_t-a^{t-1}k_1)$, reducing k_1 to zero without loss of generality. Therefore we obtain the comparatively simple formula

$$L = \{ V_0 + y_1 V_1 + y_2 V_2 + \dots + y_t V_t \mid \text{integer } y_1, y_2, \dots, y_t \}, \quad (7)$$

where

$$V_1 = \frac{1}{m}(1, a, a^2, \dots, a^{t-1}); \quad (8)$$

$$V_2 = (0, 1, 0, \dots, 0), \quad V_3 = (0, 0, 1, \dots, 0), \quad \dots, \quad V_t = (0, 0, 0, \dots, 1). \quad (9)$$

The points (x_1, x_2, \dots, x_t) of L that satisfy $0 \leq x_j < 1$ for all j are precisely the m points of our original set (2).

Note that the increment c appears only in V_0 , and the effect of V_0 is merely to shift all elements of L without changing their relative distances; hence c does not affect the spectral test in any way, and we might as well assume that $V_0 = (0, 0, \dots, 0)$ when we are calculating ν_t . When V_0 is the zero vector we have a so-called *lattice* of points

$$L_0 = \{ y_1 V_1 + y_2 V_2 + \dots + y_t V_t \mid \text{integer } y_1, y_2, \dots, y_t \}, \quad (10)$$

and our goal is to study the distances between adjacent $(t-1)$ -dimensional hyperplanes, in families of parallel hyperplanes that cover all the points of L_0 .

A family of parallel $(t-1)$ -dimensional hyperplanes can be defined by a nonzero vector $U = (u_1, \dots, u_t)$ that is perpendicular to all of them; and the set of points on a particular hyperplane is then

$$\{ (x_1, \dots, x_t) \mid x_1 u_1 + \dots + x_t u_t = q \}, \quad (11)$$

where q is a different constant for each hyperplane in the family. In other words, each hyperplane is the set of all X for which the *dot product* $X \cdot U$ has a given value q . In our case the hyperplanes are all separated by a fixed distance, and one of them contains $(0, 0, \dots, 0)$; hence we can adjust the magnitude of U so that the set of all integer values q gives all the hyperplanes in the family. Then the distance between neighboring hyperplanes is the minimum distance from $(0, 0, \dots, 0)$ to the hyperplane for $q = 1$, namely

$$\min_{\text{real } x_1, \dots, x_t} \left\{ \sqrt{x_1^2 + \dots + x_t^2} \mid x_1 u_1 + \dots + x_t u_t = 1 \right\}. \quad (12)$$

Cauchy's inequality (cf. exercise 1.2.3–30) tells us that

$$(x_1 u_1 + \dots + x_t u_t)^2 \leq (x_1^2 + \dots + x_t^2)(u_1^2 + \dots + u_t^2), \quad (13)$$

hence the minimum in (12) occurs when each $x_j = u_j/(u_1^2 + \dots + u_t^2)$; the distance between neighboring hyperplanes is

$$1/\sqrt{u_1^2 + \dots + u_t^2} = 1/\text{length}(U). \quad (14)$$

In other words, the quantity ν_t we seek is precisely the length of the shortest vector U that defines a family of hyperplanes $\{X \cdot U = q \mid \text{integer } q\}$ containing all the elements of L_0 .

Such a vector $U = (u_1, \dots, u_t)$ must be nonzero, and it must satisfy $V \cdot U = \text{integer}$ for all V in L_0 . In particular, since the points $(1, 0, \dots, 0)$, $(0, 1, \dots, 0)$, \dots , $(0, 0, \dots, 1)$ are all in L_0 , all of the u_j must be integers. Furthermore since V_1 is in L_0 , we must have $\frac{1}{m}(u_1 + au_2 + \dots + a^{t-1}u_t) = \text{integer}$, i.e.,

$$u_1 + au_2 + \dots + a^{t-1}u_t \equiv 0 \pmod{m}. \quad (15)$$

Conversely, any nonzero integer vector $U = (u_1, \dots, u_t)$ satisfying (15) defines a family of hyperplanes with the required properties, since all of L_0 will be covered: $(y_1 V_1 + \dots + y_t V_t) \cdot U$ will be an integer for all integers y_1, \dots, y_t . We have proved that

$$\begin{aligned} \nu_t^2 &= \min_{(u_1, \dots, u_t) \neq (0, \dots, 0)} \{u_1^2 + \dots + u_t^2 \mid u_1 + au_2 + \dots + a^{t-1}u_t \equiv 0 \pmod{m}\} \\ &= \min_{(x_1, \dots, x_t) \neq (0, \dots, 0)} ((mx_1 - ax_2 - a^2x_3 - \dots - a^{t-1}x_t)^2 + x_2^2 + x_3^2 + \dots + x_t^2). \end{aligned} \quad (16)$$

C. Deriving a computational method. We have now reduced the spectral test to the problem of finding the minimum value (16); but how on earth can we determine that minimum value in a reasonable amount of time? A brute-force search is out of the question, since m is very large in cases of practical interest.

It will be interesting and probably more useful if we develop a computational method for solving an even more general problem: *Find the minimum value of the quantity*

$$f(x_1, \dots, x_t) = (u_{11}x_1 + \dots + u_{1t}x_t)^2 + \dots + (u_{tt}x_1 + \dots + u_{tt}x_t)^2 \quad (17)$$

over all nonzero integer vectors (x_1, \dots, x_t) , given any nonsingular matrix of coefficients $U = (u_{ij})$. The expression (17) is called a “positive definite quadratic form” in t variables. Since U is nonsingular, (17) cannot be zero unless the x_j are all zero.

Let us write U_1, \dots, U_t for the rows of U . Then (17) may be written

$$f(x_1, \dots, x_t) = (x_1 U_1 + \dots + x_t U_t) \cdot (x_1 U_1 + \dots + x_t U_t), \quad (18)$$

the square of the length of the vector $x_1 U_1 + \dots + x_t U_t$. The nonsingular matrix U has an inverse, which means that we can find uniquely determined vectors

V_1, \dots, V_t such that

$$U_i \cdot V_j = \delta_{ij}, \quad 1 \leq i, j \leq t. \quad (19)$$

For example, in the special form (16) that arises in the spectral test, we have

$$\begin{aligned} U_1 &= (m, 0, 0, \dots, 0), & V_1 &= \frac{1}{m}(1, a, a^2, \dots, a^{t-1}), \\ U_2 &= (-a, 1, 0, \dots, 0), & V_2 &= (0, 1, 0, \dots, 0), \\ U_3 &= (-a^2, 0, 1, \dots, 0), & V_3 &= (0, 0, 1, \dots, 0), \\ &\vdots & &\vdots \\ U_t &= (-a^{t-1}, 0, 0, \dots, 1), & V_t &= (0, 0, 0, \dots, 1). \end{aligned} \quad (20)$$

These V_j are precisely the vectors (8), (9) that we used to define our original lattice L_0 . As the reader may well suspect, this is not a coincidence—indeed, if we had begun with an arbitrary lattice L_0 , defined by any set of linearly independent vectors V_1, \dots, V_t , the argument we have used above can be generalized to show that the maximum separation between hyperplanes in a covering family is equivalent to minimizing (17), where the coefficients u_{ij} are defined by (19). (See exercise 2.)

Our first step in minimizing (18) is to reduce it to a finite problem, i.e., to show that we won't need to test infinitely many vectors (x_1, \dots, x_t) to find the minimum. This is where the vectors V_1, \dots, V_t come in handy; we have

$$x_k = (x_1 U_1 + \dots + x_t U_t) \cdot V_k,$$

and Cauchy's inequality tells us that

$$((x_1 U_1 + \dots + x_t U_t) \cdot V_k)^2 \leq f(x_1, \dots, x_t)(V_k \cdot V_k).$$

Hence we have derived a useful upper bound on each coordinate x_k :

Lemma A. *Let (x_1, \dots, x_t) be a nonzero vector that minimizes (18) and let (y_1, \dots, y_t) be any nonzero integer vector. Then*

$$x_k^2 \leq (V_k \cdot V_k)f(y_1, \dots, y_t), \quad \text{for } 1 \leq k \leq t. \quad (21)$$

In particular, letting $y_i = \delta_{ij}$ for all i ,

$$x_k^2 \leq (V_k \cdot V_k)(U_j \cdot U_j), \quad \text{for } 1 \leq j, k \leq t. \quad \blacksquare \quad (22)$$

Lemma A reduces the problem to a finite search, but the right-hand side of (21) is usually much too large to make an exhaustive search feasible; we need at least one more idea. On such occasions, an old maxim provides sound advice: "If you can't solve a problem as it is stated, change it into a simpler problem that

has the same answer." For example, Euclid's algorithm has this form; if we don't know the gcd of the input numbers, we change them into smaller numbers having the same gcd. (In fact, a slightly more general approach probably underlies the discovery of nearly all algorithms: "If you can't solve a problem directly, change it into one or more simpler problems, from whose solution you can solve the original one.")

In our case, a simpler problem is one that requires less searching because the right-hand side of (22) is smaller. The key idea we shall use is that it is possible to change one quadratic form into another one that is equivalent for all practical purposes. Let j be any fixed subscript, $1 \leq j \leq t$; let $(q_1, \dots, q_{j-1}, q_{j+1}, \dots, q_t)$ be any sequence of $t - 1$ integers; and consider the following transformation of the vectors:

$$\begin{aligned} V'_i &= V_i - q_i V_j, & x'_i &= x_i - q_i x_j, & U'_i &= U_i, & \text{for } i \neq j; \\ V'_j &= V_j, & x'_j &= x_j, & U'_j &= U_j + \sum_{i \neq j} q_i U_i. \end{aligned} \quad (23)$$

It is easy to see that the new vectors U'_1, \dots, U'_t define a quadratic form f' for which $f'(x'_1, \dots, x'_t) = f(x_1, \dots, x_t)$; furthermore the basic orthogonality condition (19) remains valid, because it is easy to check that $U'_i \cdot V'_j = \delta_{ij}$. As (x_1, \dots, x_t) runs through all nonzero integer vectors, so does (x'_1, \dots, x'_t) ; hence the new form f' has the same minimum as f .

Our goal is to use transformation (23), replacing U_i by U'_i and V_i by V'_i for all i , in order to make the right-hand side of (22) small; and the right-hand side of (22) will be small when both $U_j \cdot U_j$ and $V_k \cdot V_k$ are small. Therefore it is natural to ask the following two questions about the transformation (23):

- a) What choice of q_i makes $V'_i \cdot V'_i$ as small as possible?
- b) What choice of $q_1, \dots, q_{j-1}, q_{j+1}, \dots, q_t$ makes $U'_j \cdot U'_j$ as small as possible?

It is easiest to solve these questions first for *real* values of the q_i . Question (a) is quite simple, since

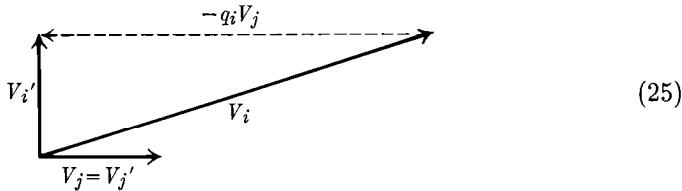
$$\begin{aligned} (V_i - q_i V_j) \cdot (V_i - q_i V_j) &= V_i \cdot V_i - 2q_i V_i \cdot V_j + q_i^2 V_j \cdot V_j \\ &= (V_j \cdot V_j)(q_i - (V_i \cdot V_j / V_j \cdot V_j))^2 \\ &\quad + V_i \cdot V_i - (V_i \cdot V_j)^2 / V_j \cdot V_j, \end{aligned}$$

and the minimum occurs when

$$q_i = V_i \cdot V_j / V_j \cdot V_j. \quad (24)$$

Geometrically, we are asking what multiple of V_j should be subtracted from V_i so that the resulting vector V'_i has minimum length, and the answer is to choose q_i so that V'_i is perpendicular to V_j (i.e., so that $V'_i \cdot V_j = 0$); the following

diagram makes this plain.



Turning to question (b), we want to choose the q_i so that $U_j + \sum_{i \neq j} q_i U_i$ has minimum length; geometrically, we want to start with U_j and add some vector in the $(t - 1)$ -dimensional hyperplane whose points are the sums of multiples of $\{U_i \mid i \neq j\}$. Again the best solution is to choose things so that U_j' is perpendicular to the hyperplane, i.e., so that $U_j' \cdot U_k = 0$ for all $k \neq j$, i.e.,

$$U_j \cdot U_k + \sum_{i \neq j} q_i(U_i \cdot U_k) = 0, \quad 1 \leq k \leq t, \quad k \neq j. \quad (26)$$

(See exercise 12 for a rigorous proof that a solution to question (b) must satisfy these $t - 1$ equations.)

Now that we have answered questions (a) and (b), we are in a bit of a quandary; should we choose the q_i according to (24), so that the $V_i' \cdot V_i'$ are minimized, or according to (26), so that $U_j' \cdot U_j'$ is minimized? Either of these alternatives makes an improvement in the right-hand side of (22), so it is not immediately clear which choice should get priority. Fortunately, there is a very simple answer to this dilemma: Conditions (24) and (26) are exactly the same! (See exercise 7.) Therefore questions (a) and (b) have the same answer; we have a happy state of affairs in which we can reduce the length of both the U 's and the V 's simultaneously. (It may be worthwhile to point out that we have just rediscovered the "Schmidt orthogonalization process.")

Our joy must be tempered with the realization that we have dealt with questions (a) and (b) only for *real* values of the q_i . Our application restricts us to integer values, so we cannot make V_i' exactly perpendicular to V_j . The best we can do for question (a) is to let q_i be the *nearest integer* to $V_i \cdot V_j / V_j \cdot V_j$ (cf. (25)). It turns out that this is not always the best solution to question (b); in fact U_j' may at times be longer than U_j . However, the bound (21) is never increased, since we can remember the smallest value of $f(y_1, \dots, y_t)$ found so far. Thus a choice of q_i based solely on question (a) is quite satisfactory.

If we apply transformation (23) repeatedly in such a way that none of the vectors V_i gets longer and at least one gets shorter, we can never get into a loop; i.e., we will never be considering the same quadratic form again after a sequence of nontrivial transformations of this kind. But eventually we will get "stuck," in the sense that none of the transformations (23) for $1 \leq j \leq t$ will be able to shorten any of the vectors V_1, \dots, V_t . At that point we can revert to an exhaustive search, using the bounds of Lemma A, which will now

be quite small in most cases. Occasionally these bounds (21) will be poor, and another type of transformation will usually get the algorithm unstuck again and reduce the bounds (see exercise 18). However, transformation (23) by itself has proved to be quite adequate for the spectral test; in fact, it has proved to be amazingly powerful when the computations are arranged as in the algorithm discussed below.

***D. How to perform the spectral test.** Here now is an efficient computational procedure that follows from our considerations. R. W. Gosper and U. Dieter have observed that it is possible to use the results of lower dimensions to make the spectral test significantly faster in higher dimensions. This refinement has been incorporated into the following algorithm, together with a significant simplification of the two-dimensional case.

Algorithm S (The spectral test). This algorithm determines the value of

$$\nu_t = \min \left\{ \sqrt{x_1^2 + \cdots + x_t^2} \mid x_1 + ax_2 + \cdots + a^{t-1}x_t \equiv 0 \pmod{m} \right\} \quad (27)$$

for $2 \leq t \leq T$, given a , m , and T , where $0 < a < m$ and a is relatively prime to m . (The number ν_t measures the t -dimensional accuracy of random number generators, as discussed in the text above.) All arithmetic within this algorithm is done on integers whose magnitudes rarely if ever exceed m^2 , except in step S8; in fact, nearly all of the integer variables will be less than m in absolute value during the computation.

When ν_t is being calculated for $t \geq 3$, the algorithm works with two $t \times t$ matrices U and V , whose row vectors are denoted by $U_i = (u_{i1}, \dots, u_{it})$ and $V_i = (v_{i1}, \dots, v_{it})$ for $1 \leq i \leq t$. These vectors satisfy the conditions

$$u_{i1} + au_{i2} + \cdots + a^{t-1}u_{it} \equiv 0 \pmod{m}, \quad 1 \leq i \leq t; \quad (28)$$

$$U_i \cdot V_j = \delta_{ij} m, \quad 1 \leq i, j \leq t. \quad (29)$$

(Thus the V_j of our previous discussion have been multiplied by m , to ensure that their components are integers.) There are three other auxiliary vectors, $X = (x_1, \dots, x_t)$, $Y = (y_1, \dots, y_t)$, and $Z = (z_1, \dots, z_t)$. During the entire algorithm, r will denote $a^{t-1} \pmod{m}$ and s will denote the smallest upper bound for ν_t^2 that has been discovered so far.

S1. [Initialize.] Set $h \leftarrow a$, $h' \leftarrow m$, $p \leftarrow 1$, $p' \leftarrow 0$, $r \leftarrow a$, $s \leftarrow 1 + a^2$. (The first steps of this algorithm handle the case $t = 2$ by a special method, very much like Euclid's algorithm; we will have

$$h - ap \equiv h' - ap' \equiv 0 \pmod{m} \quad \text{and} \quad hp' - h'p = \pm m \quad (30)$$

during this phase of the calculation.)

S2. [Euclidean step.] Set $q \leftarrow \lfloor h'/h \rfloor$, $u \leftarrow h' - qh$, $v \leftarrow p' - qp$. If $u^2 + v^2 < s$, set $s \leftarrow u^2 + v^2$, $h' \leftarrow h$, $h \leftarrow u$, $p' \leftarrow p$, $p \leftarrow v$, and repeat step S2.

S3. [Compute ν_2 .] Set $u \leftarrow u - h$, $v \leftarrow v - p$; and if $u^2 + v^2 < s$, set $s \leftarrow u^2 + v^2$, $h' \leftarrow u$, $p' \leftarrow v$. Then output $\sqrt{s} = \nu_2$. (The validity of this calculation for the two-dimensional case is proved in exercise 5. Now we will set up the U and V matrices satisfying (28) and (29), in preparation for calculations in higher dimensions.) Set

$$U \leftarrow \begin{pmatrix} -h & p \\ -h' & p' \end{pmatrix}, \quad V \leftarrow \pm \begin{pmatrix} p' & h' \\ -p & -h \end{pmatrix},$$

where the $-$ sign is chosen for V if and only if $p' > 0$.

S4. [Advance t .] If $t = T$, the algorithm terminates. (Otherwise we want to increase t by 1. At this point U and V are $t \times t$ matrices satisfying (28) and (29), and we must enlarge them by adding an appropriate new row and column.) Set $t \leftarrow t + 1$ and $r \leftarrow (ar) \bmod m$. Set U_t to the new row $(-r, 0, 0, \dots, 0, 1)$ of t elements, and set $u_{it} \leftarrow 0$ for $1 \leq i < t$. Set V_t to the new row $(0, 0, 0, \dots, 0, m)$. Finally, for $1 \leq i < t$, set $q \leftarrow \text{round}(v_{i1} r/m)$, $v_{it} \leftarrow v_{i1} r - qm$, and $U_t \leftarrow U_t + qU_i$. (Here “round(x)” denotes the nearest integer to x , e.g., $[x + 1/2]$. We are essentially setting $v_{it} \leftarrow v_{i1} r$ and immediately applying transformation (23) with $j = t$, since the numbers $|v_{i1} r|$ are so large they ought to be reduced at once.) Finally set $s \leftarrow \min(s, U_t \cdot U_t)$, $k \leftarrow t$, and $j \leftarrow 1$. (In the following steps, j denotes the current row index for transformation (23), and k denotes the last such index where the transformation shortened at least one of the V_i .)

S5. [Transform.] For $1 \leq i \leq t$, do the following operations: If $i \neq j$ and $2|V_i \cdot V_j| > V_j \cdot V_j$, set $q \leftarrow \text{round}(V_i \cdot V_j / V_j \cdot V_j)$, $V_i \leftarrow V_i - qV_j$, $U_j \leftarrow U_j + qU_i$, and $k \leftarrow j$. (The fact that we omit this transformation, when $2|V_i \cdot V_j|$ exactly equals $V_j \cdot V_j$, prevents the algorithm from looping endlessly; see exercise 19.)

S6. [Examine new bound.] If $k = j$ (i.e., if the transformation in S5 has just done something useful), set $s \leftarrow \min(s, U_j \cdot U_j)$.

S7. [Advance j .] If $j = t$, set $j \leftarrow 1$; otherwise set $j \leftarrow j + 1$. Now if $j \neq k$, return to step S5. (If $j = k$, we have gone through $t - 1$ consecutive cycles of no transformation, so the transformation process is stuck.)

S8. [Prepare for search.] (Now the absolute minimum will be determined, using an exhaustive search over all (x_1, \dots, x_t) satisfying condition (21) of Lemma A.) Set $X \leftarrow Y \leftarrow (0, \dots, 0)$, set $k \leftarrow t$, and set

$$z_j \leftarrow \left\lfloor \sqrt{\lfloor (V_j \cdot V_j)s/m^2 \rfloor} \right\rfloor, \quad \text{for } 1 \leq j \leq t. \quad (31)$$

(We will examine all $X = (x_1, \dots, x_t)$ with $|x_j| \leq z_j$ for $1 \leq j \leq t$. In hundreds of applications of this algorithm, no z_j has yet turned out to be greater than 1, nor has the exhaustive search in the following steps ever reduced s ; however, such phenomena are probably possible in weird cases,

especially in higher dimensions. During the exhaustive search, the vector Y will always be equal to $x_1 U_1 + \dots + x_t U_t$, so that $f(x_1, \dots, x_t) = Y \cdot Y$. Since $f(-x_1, \dots, -x_t) = f(x_1, \dots, x_t)$, we shall examine only vectors whose first nonzero component is positive. The method is essentially that of counting in steps of one, regarding (x_1, \dots, x_t) as the digits in a balanced number system with mixed radices $(2z_1 + 1, \dots, 2z_t + 1)$; cf. Section 4.1.)

- S9.** [Advance x_k .] If $x_k = z_k$, go to S11. Otherwise increase x_k by 1 and set $Y \leftarrow Y + U_k$.
- S10.** [Advance k .] Set $k \leftarrow k + 1$. Then if $k \leq t$, set $x_k \leftarrow -z_k$, $Y \leftarrow Y - 2z_k U_k$, and repeat step S10. But if $k > t$, set $s \leftarrow \min(s, Y \cdot Y)$.
- S11.** [Decrease k .] Set $k \leftarrow k - 1$. If $k \geq 1$, return to S9. Otherwise output $\nu_t = \sqrt{s}$ (the exhaustive search is completed) and return to S4. ■

In practice Algorithm S is applied for $T = 5$ or 6 , say; it usually works reasonably well when $T = 7$ or 8 , but it can be terribly slow when $T \geq 9$ since the exhaustive search tends to make the running time grow as 3^T . (If the minimum value ν_t occurs at many different points, the exhaustive search will hit them all; hence we typically find that all $z_k = 1$ for large t . As remarked above, the values of ν_t are generally irrelevant for practical purposes when t is large.)

An example will help to make Algorithm S clear. Consider the linear congruential sequence defined by

$$m = 10^{10}, \quad a = 3141592621, \quad c = 1, \quad X_0 = 0. \quad (32)$$

Six cycles of the Euclidean algorithm in steps S2 and S3 suffice to prove that the minimum nonzero value of $x_1^2 + x_2^2$ with

$$x_1 + 3141592621x_2 \equiv 0 \pmod{10^{10}}$$

occurs for $x_1 = 67654$, $x_2 = 226$; hence the two-dimensional accuracy of this generator is

$$\nu_2 = \sqrt{67654^2 + 226^2} \approx 67654.37748.$$

Passing to three dimensions, we seek the minimum nonzero value of $x_1^2 + x_2^2 + x_3^2$ such that

$$x_1 + 3141592621x_2 + 3141592621^2 x_3 \equiv 0 \pmod{10^{10}}. \quad (33)$$

Step S4 sets up the matrices

$$U = \begin{pmatrix} -67654 & -226 & 0 \\ -44190611 & 191 & 0 \\ 5793866 & 33 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} -191 & -44190611 & 2564918569 \\ -226 & 67654 & 1307181134 \\ 0 & 0 & 10000000000 \end{pmatrix}.$$

The first iteration of step S5, with $q = 1$ for $i = 2$ and $q = 4$ for $i = 3$, changes them to

$$U = \begin{pmatrix} -21082801 & 97 & 4 \\ -44190611 & 191 & 0 \\ 5793866 & 33 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} -191 & -44190611 & 2564918569 \\ -35 & 44258265 & -1257737435 \\ 764 & 176762444 & -259674276 \end{pmatrix}.$$

(Note that the first row U_1 has actually gotten longer in this transformation, although eventually the rows of U should get shorter.)

The next fourteen iterations of step S5 have $(j, q_1, q_2, q_3) = (2, -2, *, 0)$, $(3, 0, 3, *)$, $(1, *, -10, -1)$, $(2, -1, *, -6)$, $(3, -1, 0, *)$, $(1, *, 0, 2)$, $(2, 0, *, -1)$, $(3, 3, 4, *)$, $(1, *, 0, 0)$, $(2, -5, *, 0)$, $(3, 1, 0, *)$, $(1, *, -3, -1)$, $(2, 0, *, 0)$, $(3, 0, 0, *)$. Now the transformation process is stuck, but the rows of the matrices have become significantly shorter:

$$U = \begin{pmatrix} -1479 & 616 & -2777 \\ -3022 & 104 & 918 \\ -227 & -983 & -130 \end{pmatrix}, \quad V = \begin{pmatrix} -888874 & 601246 & -2994234 \\ -2809871 & 438109 & 1593689 \\ -854296 & -9749816 & -1707736 \end{pmatrix}. \quad (34)$$

The search limits (z_1, z_2, z_3) in step S8 turn out to be $(0, 0, 1)$, so U_3 is the shortest solution to (33); we have

$$\nu_3 = \sqrt{227^2 + 983^2 + 130^2} \approx 1017.21089.$$

Note that only a few iterations were needed to find this value, although condition (33) looks quite difficult to deal with at first glance. All points (U_n, U_{n+1}, U_{n+2}) produced by this random number generator lie on a family of parallel planes about 0.001 units apart.

E. Ratings for various generators. So far we haven't really given a criterion that tells us whether or not a particular random number generator "passes" or "flunks" the spectral test. In fact, this depends on the application, since some applications demand higher resolution than others. It appears that $\nu_t \geq 2^{30/t}$ for $2 \leq t \leq 6$ will be quite adequate in most applications (although the author must admit choosing this criterion partly because 30 is conveniently divisible by 2, 3, 5, and 6).

For some purposes we would like a criterion that is relatively independent of m , so we can say that a particular multiplier is good or bad with respect to the set of all other multipliers for the given m , without examining any others. A reasonable figure of merit for rating the goodness of a particular multiplier seems to be the volume of the ellipsoid in t -space defined by the relation $(x_1 m - x_2 a - \dots - x_t a^{t-1})^2 + x_2^2 + \dots + x_t^2 \leq \nu_t^2$, since this volume tends to indicate how likely it is that nonzero integer points (x_1, \dots, x_t) —corresponding to solutions of (15)—are in the ellipsoid. We therefore propose to calculate this volume, namely

$$\mu_t = \frac{\pi^{t/2} \nu_t^t}{(t/2)! m}, \quad (35)$$

as an indication of the effectiveness of the multiplier a for the given m . In this formula,

$$\left(\frac{t}{2}\right)! = \left(\frac{t}{2}\right)\left(\frac{t}{2}-1\right)\dots\left(\frac{1}{2}\right)\sqrt{\pi}, \quad \text{for } t \text{ odd.} \quad (36)$$

Table 1
SAMPLE RESULTS OF THE SPECTRAL TEST

Line	a	m	ν_2^2	ν_3^2	ν_4^2	ν_5^2	ν_6^2
1	23	$10^8 + 1$	530	530	530	530	447
2	$2^7 + 1$	2^{35}	16642	16642	16642	15602	252
3	$2^{18} + 1$	2^{35}	34359738368	6	4	4	4
4	3141592653	2^{35}	2997222016	1026050	27822	1118	1118
5	137	256	274	30	14	6	4
6	3141592621	10^{10}	4577114792	1034718	62454	1776	542
7	3141592221	10^{10}	4293881050	276266	97450	3366	2382
8	4219755981	10^{10}	10721093248	2595578	49362	5868	820
9	4160984121	10^{10}	9183801602	4615650	16686	6840	1344
10	3141592221	2^{35}	13539813818	5795090	88134	12716	2938
11	2718281829	2^{35}	22939188896	2723830	146116	10782	2914
12	5^{13}	2^{35}	33161885770	2925242	113374	13070	2256
13	5^{15}	2^{35}	22078865098	10274746	167558	5844	2592
14	$2^{23} + 2^{12} + 5$	2^{35}	167510120	8052254	21476	16802	1630
15	$2^{23} + 2^{13} + 5$	2^{35}	168231328	5335322	21476	2008	1134
16	$2^{23} + 2^{14} + 5$	2^{35}	12256151168	5733878	21476	13316	2032
17	$2^{22} + 2^{13} + 5$	2^{35}	8201443840	1830230	21476	7786	3080
18	$2^{24} + 2^{13} + 5$	2^{35}	8364058	8364058	21476	16712	1496
19	19935388837	2^{35}	32300850938	705518	22270	9558	2660
20	1175245817	2^{35}	36436418002	7362242	95306	3006	2860
21	17059465	2^{35}	39341117000	9476606	202796	18758	2382
22	$2^{16} + 3$	2^{29}	536805386	118	116	116	116
23	1812433253	2^{32}	4326934538	1462856	15082	4866	906
24	1566083941	2^{32}	4659748970	2079590	44902	4652	662
25	69069	2^{32}	4243209856	2072544	52804	6990	242
26	1664525	2^{32}	4938916874	2322494	63712	4092	1038
27	314159269	$2^{31} - 1$	1432232969	899290	36985	3427	1144
28	see (39)		$(2^{31} - 1)^2$	1.4×10^{12}	643578623	12930027	837632
29	31167285	2^{48}	3.2×10^{14}	4111841446	17341510	306326	59278
30	see the text	2^{64}	8.8×10^{18}	6.4×10^{12}	4.1×10^9	45662836	1846368

Thus, in six or fewer dimensions the merit is computed as follows:

$$\begin{aligned}\mu_2 &= \pi \nu_2^2 / m, & \mu_3 &= \frac{4}{3} \pi \nu_3^2 / m, & \mu_4 &= \frac{1}{2} \pi^2 \nu_4^4 / m, \\ \mu_5 &= \frac{8}{15} \pi^2 \nu_5^5 / m, & \mu_6 &= \frac{1}{6} \pi^3 \nu_6^6 / m.\end{aligned}\quad (37)$$

We might say that the multiplier a passes the spectral test if μ_t is 0.1 or more for $2 \leq t \leq 6$, and it "passes with flying colors" if $\mu_t \geq 1$ for all these t . A low value of μ_t means that we have probably picked a very unfortunate multiplier, since very few lattices will have integer points so close to the origin. Conversely, a high value of μ_t means that we have found an unusually good multiplier for the given m ; but it does not mean that the random numbers are necessarily very good, since m might be too small. Only the values ν_t truly indicate the degree of randomness.

$$(\epsilon = \frac{1}{10})$$

$\lg \nu_2$	$\lg \nu_3$	$\lg \nu_4$	$\lg \nu_5$	$\lg \nu_6$	μ_2	μ_3	μ_4	μ_5	μ_6	Line
4.5	4.5	4.5	4.5	4.4	$2\epsilon^5$	$5\epsilon^4$	0.01	0.34	4.62	1
7.0	7.0	7.0	7.0	4.0	$2\epsilon^6$	$3\epsilon^4$	0.04	4.66	$2\epsilon^3$	2
17.5	1.3	1.0	1.0	1.0	3.14	$2\epsilon^9$	$2\epsilon^9$	$5\epsilon^9$	ϵ^8	3
15.7	10.0	7.4	5.0	5.0	0.27	0.13	0.11	0.01	0.21	4
4.0	2.5	1.9	1.3	1.0	3.36	2.69	3.78	1.81	1.29	5
16.0	10.0	8.0	5.4	4.5	1.44	0.44	1.92	0.07	0.08	6
16.0	9.0	8.3	5.9	5.6	1.35	0.06	4.69	0.35	6.98	7
16.7	10.7	7.8	6.3	4.8	3.39	1.75	1.20	1.39	0.28	8
16.5	11.1	7.0	6.4	5.2	2.89	4.15	0.14	2.04	1.25	9
16.8	11.2	8.2	6.8	5.8	1.24	1.70	1.12	2.79	3.81	10
17.2	10.7	8.6	6.7	5.8	2.10	0.55	3.15	1.85	3.72	11
17.5	10.7	8.4	6.8	5.6	3.03	0.61	1.85	2.99	1.73	12
17.2	11.6	8.7	6.3	5.7	2.02	4.02	4.03	0.40	2.62	13
13.7	11.5	7.2	7.0	5.3	0.02	2.79	0.07	5.61	0.65	14
13.7	11.2	7.2	5.5	5.1	0.02	1.50	0.07	0.03	0.22	15
16.8	11.2	7.2	6.9	5.5	1.12	1.67	0.07	3.13	1.26	16
16.5	10.4	7.2	6.5	5.8	0.75	0.30	0.07	0.82	4.39	17
11.5	11.5	7.2	7.0	5.3	$8\epsilon^4$	2.95	0.07	5.53	0.50	18
17.5	9.7	7.2	6.6	5.7	2.95	0.07	0.07	1.37	2.83	19
17.5	11.4	8.3	5.8	5.7	3.33	2.44	1.30	0.08	3.52	20
17.6	11.6	8.8	7.1	5.6	3.60	3.56	5.91	7.38	2.03	21
14.5	3.4	3.4	3.4	3.4	3.14	ϵ^5	ϵ^4	ϵ^3	0.02	22
16.0	10.2	6.9	6.1	4.9	3.16	1.73	0.26	2.02	0.89	23
16.1	10.5	7.7	6.1	4.7	3.41	2.92	2.32	1.81	0.35	24
16.0	10.5	7.8	6.4	4.0	3.10	2.91	3.20	5.01	0.02	25
16.1	10.6	8.0	6.0	5.0	3.61	3.45	4.66	1.31	1.35	26
15.2	9.9	7.6	5.9	5.1	2.10	1.66	3.14	1.69	3.60	27
31.0	20.2	15.6	11.8	9.8	3.14	1.49	0.44	0.69	0.66	28
24.1	16.0	12.0	9.1	7.9	3.60	3.92	5.27	0.97	3.82	29
31.5	21.3	16.0	12.7	10.4	1.50	3.68	4.52	4.02	1.76	30

upper bounds from (40): 3.63 5.92 9.87 14.89 23.87

Table 1 shows what sorts of values occur in typical sequences. Each line of the table considers a particular generator, and lists ν_t , μ_t , and the “number of bits of accuracy” $\lg \nu_t$. Lines 1 through 4 show the generators that were the subject of Figs. 2 and 5 in Section 3.3.1. The generators in lines 1 and 2 suffer from too small a multiplier; a diagram like Fig. 8 will have a nearly vertical “stripes” when a is small. The terrible generator in line 3 has a good μ_2 but very poor μ_3 and μ_4 ; like nearly all generators of potency 2, it has $\nu_3 = \sqrt{6}$ and $\nu_4 = 2$ (see exercise 3). Line 4 shows a “random” multiplier; this generator has satisfactorily passed numerous empirical tests for randomness, but it does not have especially high values of μ_2, \dots, μ_6 . In fact, the value of μ_5 flunks our criterion.

Line 5 shows the generator of Fig. 8. It passes the spectral test with very high-flying colors, when μ_2 through μ_6 are considered, but of course m is so

small that the numbers can hardly be called random; the ν_t values are terribly low.

Line 6 is the generator discussed above; line 7 is a similar example, having an abnormally low value of μ_3 . Line 8 shows a nonrandom multiplier for the same modulus m ; all of its partial quotients are 1, 2, or 3. Such multipliers have been suggested by I. Borosh and H. Niederreiter because the Dedekind sums are likely to be especially small and because they produce best results in the two-dimensional serial test (cf. Section 3.3.3 and exercise 30). The particular example in line 8 has only one ‘3’ as a partial quotient; there is no multiplier congruent to 1 modulo 20 whose partial quotients with respect to 10^{10} are only 1’s and 2’s. The generator in line 9 shows another multiplier chosen with malice aforethought, following a suggestion by A. G. Waterman that guarantees a reasonably high value of μ_2 (see exercise 11).

Lines 10 through 21 of Table 1 show further examples with $m = 2^{35}$, beginning with some random multipliers. The generators in lines 12 and 13 are reminders of the good old days—they were once used extensively since O. Taussky first suggested them in the early 1950s. Lines 14 through 18 show various multipliers of maximum potency having only four 1’s in their binary representation. The point of having few 1’s is to replace multiplication by a few additions, but only line 16 comes near to being passable. Since these multipliers satisfy $(a - 5)^3 \bmod 2^{35} = 0$, all five of them achieve ν_4 at the same point $(x_1, x_2, x_3, x_4) = (-125, 75, -15, 1)$. Another curiosity is the high value of μ_3 following a very low μ_2 in line 18 (see exercise 8). Lines 19 and 20 are respectively the Borosh–Niederreiter and Waterman multipliers for modulus 2^{35} ; and line 21 was found by M. Lavaux and F. Janssens, in a computer search for spectrally good multipliers having a very high μ_2 .

Lines 22 through 28 apply to System/370 and other machines with 32-bit words; in this case the comparatively small word size calls for comparatively greater care. Line 22 is, regrettably, the generator that has actually been used on such machines in most of the world’s scientific computing centers for about a decade; its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists! The actual generator is defined by

$$X_0 \text{ odd}, \quad X_{n+1} = (65539X_n) \bmod 2^{31}, \quad (38)$$

and exercise 20 indicates that 2^{29} is the appropriate modulus for the spectral test. Since $9X_n + 6X_{n+2} + X_{n+4} \equiv 0$ (modulo 2^{31}), the generator fails most three-dimensional criteria for randomness, and it should never have been used. Almost any multiplier $\equiv 5$ (modulo 8) would be better. (A curious fact about RANDU, noticed by R. W. Gosper, is that $\nu_4 = \nu_5 = \nu_6 = \nu_7 = \nu_8 = \nu_9 = \sqrt{116}$, hence μ_9 is a spectacular 11.98.) Lines 23 and 24 are the Borosh–Niederreiter and Waterman multipliers for modulus 2^{32} , lines 26 and 29 were found by Lavaux and Janssens, and line 30 (whose excellent multiplier 6364136223846793005 is too big to fit in the column) is due to C. E. Haynes. Line 25 was nominated by George Marsaglia as “a candidate for the best of all multipliers,” after a computer search

in dimensions 2 through 5, partly because it is easy to remember. Line 27 uses a random primitive root, modulo the prime $2^{31} - 1$, as multiplier. Line 28 is for the sequence

$$X_n = (271828183X_{n-1} - 314159269X_{n-2}) \bmod (2^{31} - 1), \quad (39)$$

which can be shown to have period length $(2^{31} - 1)^2 - 1$; it has been analyzed with the generalized spectral test of exercise 24.

Theoretical upper bounds on μ_t , which can never be transcended for any m , are shown just below Table 1; it is known that every lattice with m points per unit volume has

$$\nu_t \leq \gamma_t m^{1/t}, \quad (40)$$

where γ_t takes the respective values

$$(4/3)^{1/4}, \quad 2^{1/6}, \quad 2^{1/4}, \quad 2^{3/10}, \quad (64/3)^{1/12}, \quad 2^{3/7}, \quad 2^{1/2} \quad (41)$$

for $t = 2, \dots, 8$. (See exercise 9 and J. W. S. Cassels, *Introduction to the Geometry of Numbers* (Berlin: Springer, 1959), p. 332.) These bounds hold for lattices generated by vectors with arbitrary real coordinates. For example, the optimum lattice for $t = 2$ is hexagonal, and it is generated by vectors of length $2/\sqrt{3m}$ that form two sides of an equilateral triangle. In three dimensions the optimum lattice is generated by vectors V_1, V_2, V_3 that can be rotated into the form $(v, v, -v), (v, -v, v), (-v, v, v)$, where $v = 1/\sqrt[3]{4m}$.

***F. Relation to the serial test.** In a series of important papers published during the 1970s, Harald Niederreiter has shown how to analyze the distribution of the t -dimensional vectors (1) by means of exponential sums. One of the main consequences of his theory is that the serial test in several dimensions will be passed by any generator that passes the spectral test, even when we consider only a sufficiently large part of the period instead of the whole period. We shall now turn briefly to a study of his interesting methods, in the case of linear congruential sequences (X_0, a, c, m) of period length m .

The first idea we need is the notion of *discrepancy* in t dimensions, a quantity that we shall define as the difference between the expected number and the actual number of t -dimensional vectors $(x_n, x_{n+1}, \dots, x_{n+t-1})$ falling into a hyper-rectangular region, maximized over all such regions. To be precise, let $\langle x_n \rangle$ be a sequence of integers in the range $0 \leq x_n < m$. We define

$$D_N^{(t)} = \max_R \left| \frac{\text{number of } (x_n, \dots, x_{n+t-1}) \text{ in } R \text{ for } 0 \leq n < N}{N} - \frac{\text{volume of } R}{m^t} \right| \quad (42)$$

where R ranges over all sets of points of the form

$$R = \{(y_1, \dots, y_t) \mid \alpha_1 \leq y_1 < \beta_1, \dots, \alpha_t \leq y_t < \beta_t\}; \quad (43)$$

here α_j and β_j are integers in the range $0 \leq \alpha_j < \beta_j \leq m$, for $1 \leq j \leq t$. The volume of R is clearly $(\beta_1 - \alpha_1) \dots (\beta_t - \alpha_t)$. To get the discrepancy $D_N^{(t)}$, we imagine looking at all these sets R and finding the one with the greatest excess or deficiency of points (x_n, \dots, x_{n+t-1}) .

An upper bound for the discrepancy can be found by using exponential sums. Let $\omega = e^{2\pi i/m}$ be a primitive m th root of unity. If (x_1, \dots, x_t) and (y_1, \dots, y_t) are two vectors with all components in the range $0 \leq x_j, y_j < m$, we have

$$\sum_{0 \leq u_1, \dots, u_t < m} \omega^{(x_1 - y_1)u_1 + \dots + (x_t - y_t)u_t} = \begin{cases} m^t, & \text{if } (x_1, \dots, x_t) = (y_1, \dots, y_t); \\ 0, & \text{if } (x_1, \dots, x_t) \neq (y_1, \dots, y_t). \end{cases}$$

Therefore the number of vectors (x_n, \dots, x_{n+t-1}) in R for $0 \leq n < N$, when R is defined by (43), can be expressed as

$$\frac{1}{m^t} \sum_{0 \leq n < N} \sum_{0 \leq u_1, \dots, u_t < m} \omega^{x_n u_1 + \dots + x_{n+t-1} u_t} \times \sum_{\alpha_1 \leq y_1 < \beta_1} \dots \sum_{\alpha_t \leq y_t < \beta_t} \omega^{-(y_1 u_1 + \dots + y_t u_t)}.$$

When $u_1 = \dots = u_t = 0$ in this sum, we get N/m^t times the volume of R ; hence we can express $D_N^{(t)}$ as

$$\max_R \left| \frac{1}{Nm^t} \sum_{0 \leq n < N} \sum_{\substack{0 \leq u_1, \dots, u_t < m \\ (u_1, \dots, u_t) \neq (0, \dots, 0)}} \omega^{x_n u_1 + \dots + x_{n+t-1} u_t} \times \sum_{\alpha_1 \leq y_1 < \beta_1} \dots \sum_{\alpha_t \leq y_t < \beta_t} \omega^{-(y_1 u_1 + \dots + y_t u_t)} \right|.$$

Since complex numbers satisfy $|w + z| \leq |w| + |z|$ and $|wz| = |w||z|$, it follows that

$$\begin{aligned} D_N^{(t)} &\leq \max_R \frac{1}{m^t} \sum_{\substack{0 \leq u_1, \dots, u_t < m \\ (u_1, \dots, u_t) \neq (0, \dots, 0)}} \left| \sum_{\alpha_1 \leq y_1 < \beta_1} \dots \sum_{\alpha_t \leq y_t < \beta_t} \omega^{-(y_1 u_1 + \dots + y_t u_t)} \right| g(u_1, \dots, u_t) \\ &\leq \frac{1}{m^t} \sum_{\substack{0 \leq u_1, \dots, u_t < m \\ (u_1, \dots, u_t) \neq (0, \dots, 0)}} \max_R \left| \sum_{\alpha_1 \leq y_1 < \beta_1} \dots \sum_{\alpha_t \leq y_t < \beta_t} \omega^{-(y_1 u_1 + \dots + y_t u_t)} \right| g(u_1, \dots, u_t) \\ &= \sum_{\substack{0 \leq u_1, \dots, u_t < m \\ (u_1, \dots, u_t) \neq (0, \dots, 0)}} f(u_1, \dots, u_t) g(u_1, \dots, u_t), \end{aligned} \tag{44}$$

where

$$\begin{aligned} g(u_1, \dots, u_t) &= \left| \frac{1}{N} \sum_{0 \leq n < N} \omega^{x_n u_1 + \dots + x_{n+t-1} u_t} \right|; \\ f(u_1, \dots, u_t) &= \max_R \frac{1}{m^t} \left| \sum_{\alpha_1 \leq y_1 < \beta_1} \dots \sum_{\alpha_t \leq y_t < \beta_t} \omega^{-(y_1 u_1 + \dots + y_t u_t)} \right| \\ &= \max_R \left| \frac{1}{m} \sum_{\alpha_1 \leq y_1 < \beta_1} \omega^{-u_1 y_1} \right| \dots \left| \frac{1}{m} \sum_{\alpha_t \leq y_t < \beta_t} \omega^{-u_t y_t} \right|. \end{aligned}$$

Both f and g can be further simplified in order to get a good upper bound on $D_N^{(t)}$. We have

$$\left| \frac{1}{m} \sum_{\alpha \leq y < \beta} \omega^{-uy} \right| = \left| \frac{1}{m} \frac{\omega^{-\beta u} - \omega^{-\alpha u}}{\omega^{-u} - 1} \right| \leq \frac{2}{m|\omega^u - 1|} = \frac{1}{m \sin(\pi u/m)}$$

when $u \neq 0$, and the sum is ≤ 1 when $u = 0$; hence

$$f(u_1, \dots, u_t) \leq r(u_1, \dots, u_t), \quad (45)$$

where

$$r(u_1, \dots, u_t) = \prod_{\substack{1 \leq k \leq t \\ u_k \neq 0}} \frac{1}{m \sin(\pi u_k/m)}. \quad (46)$$

Furthermore, when $\langle x_n \rangle$ is generated modulo m by a linear congruential sequence, we have

$$\begin{aligned} x_n u_1 + \dots + x_{n+t-1} u_t &= x_n u_1 + (ax_n + c)u_2 + \dots \\ &\quad + (a^{t-1}x_n + c(a^{t-2} + \dots + 1))u_t \\ &= (u_1 + au_2 + \dots + a^{t-1}u_t)x_n + h(u_1, \dots, u_t) \end{aligned}$$

where $h(u_1, \dots, u_t)$ is independent of n ; hence

$$g(u_1, \dots, u_t) = \left| \frac{1}{N} \sum_{0 \leq n < N} \omega^{q(u_1, \dots, u_t)x_n} \right|, \quad (47)$$

where

$$q(u_1, \dots, u_t) = u_1 + au_2 + \dots + a^{t-1}u_t. \quad (48)$$

Now here is where the connection to the spectral test comes in: We will show that the sum $g(u_1, \dots, u_t)$ is rather small unless $q(u_1, \dots, u_t) \equiv 0$ (modulo m), i.e., unless (u_1, \dots, u_t) is a solution to (15). Furthermore exercise 27 shows that $r(u_1, \dots, u_t)$ is rather small when (u_1, \dots, u_t) is a “large” solution to (15). Hence the discrepancy $D_N^{(t)}$ will be rather small when (15) has only “large” solutions, i.e., when the spectral test is passed. All that remains is to quantify these qualitative statements by making careful calculations.

In the first place, let's consider the size of $g(u_1, \dots, u_t)$. When $N = m$, so that the sum (47) is over an entire period, we have $g(u_1, \dots, u_t) = 0$ except when (u_1, \dots, u_t) satisfies (15), so the discrepancy is bounded above in this case by the sum of $r(u_1, \dots, u_t)$ taken over all the nonzero solutions of (15). But let's consider also what happens in a sum like (47) when N is less than m and $q(u_1, \dots, u_t)$ is not a multiple of m . We have

$$\begin{aligned} \frac{1}{N} \sum_{0 \leq n < N} \omega^{x_n} &= \frac{1}{N} \sum_{0 \leq n < N} \frac{1}{m} \sum_{0 \leq k < m} \omega^{-nk} \sum_{0 \leq j < m} \omega^{x_j + jk} \\ &= \frac{1}{N} \sum_{0 \leq k < m} \left(\frac{1}{m} \sum_{0 \leq n < N} \omega^{-nk} \right) S_{k0}, \end{aligned} \quad (49)$$

where

$$S_{kl} = \sum_{0 \leq j < m} \omega^{x_j + l + jk}. \quad (50)$$

Now $S_{kl} = \omega^{-lk} S_{k0}$, so $|S_{kl}| = |S_{k0}|$ for all l , and we can calculate this common value by further exponential-summary:

$$\begin{aligned} |S_{k0}|^2 &= \frac{1}{m} \sum_{0 \leq l < m} |S_{kl}|^2 \\ &= \frac{1}{m} \sum_{0 \leq l < m} \sum_{0 \leq j < m} \omega^{x_j + l + jk} \sum_{0 \leq i < m} \omega^{-x_i + l - ik} \\ &= \frac{1}{m} \sum_{0 \leq i, j < m} \omega^{(j-i)k} \sum_{0 \leq l < m} \omega^{x_j + l - x_i + l} \\ &= \frac{1}{m} \sum_{0 \leq i < m} \sum_{i \leq j < m+i} \omega^{(j-i)k} \sum_{0 \leq l < m} \omega^{(a^{j-i}-1)x_i + l + (a^{j-i}-1)c/(a-1)}. \end{aligned}$$

Let s be minimum such that $a^s \equiv 1$ (modulo m), and let

$$s' = (a^s - 1)c/(a - 1) \bmod m.$$

Then s is a divisor of m , and $x_n + js \equiv x_n + js'$ (modulo m). The sum on l vanishes unless $j - i$ is a multiple of s , so we find that

$$|S_{k0}|^2 = m \sum_{0 \leq j < m/s} \omega^{jsk + js'}.$$

We have $s' = q's$ where q' is relatively prime to m (cf. exercise 3.2.1–21), so it turns out that

$$|S_{k0}| = \begin{cases} 0, & \text{if } k + q' \not\equiv 0 \pmod{m/s}; \\ m/\sqrt{s}, & \text{if } k + q' \equiv 0 \pmod{m/s}. \end{cases} \quad (51)$$

Putting this information back into (49), and recalling the derivation of (45), shows that

$$\left| \frac{1}{N} \sum_{0 \leq n < N} \omega^{x_n} \right| \leq \frac{m}{N\sqrt{s}} \sum_k r(k), \quad (52)$$

where the sum is over $0 < k < m$ such that $k + q' \equiv 0$ (modulo m/s). Exercise 25 now can be used to estimate the remaining sum, and we find that

$$\left| \frac{1}{N} \sum_{0 \leq n < N} \omega^{x_n} \right| \leq \frac{2\sqrt{s}}{\pi N} \ln s + O\left(\frac{m}{N\sqrt{s}}\right). \quad (53)$$

The same bound can be used to estimate $|N^{-1} \sum_{0 \leq n < N} \omega^{qx_n}|$ for any $q \not\equiv 0$ (modulo m), since the effect is to replace m in this derivation by a divisor of m .

In fact, the upper bound gets even smaller when q has a factor in common with m , since s and m/\sqrt{s} generally become smaller. (See exercise 26.)

We have now proved that the $g(u_1, \dots, u_t)$ part of our upper bound (44) on the discrepancy is small, if N is large enough and if (u_1, \dots, u_t) does not satisfy the spectral test congruence (15). Exercise 27 proves that the $f(u_1, \dots, u_t)$ part of our upper bound is small, when summed over all the nonzero vectors (u_1, \dots, u_t) satisfying (15), provided that all such vectors are far away from $(0, \dots, 0)$. Putting these results together leads to the following theorem of Niederreiter:

Theorem N. Let $\langle X_n \rangle$ be a linear congruential sequence (X_0, a, c, m) of period length m , and let s be the least positive integer such that $a^s \equiv 1$ (modulo m). Let ν_t be the t -dimensional accuracy of $\langle X_n \rangle$, as determined by the spectral test. Then the t -dimensional discrepancy $D_N^{(t)}$ determined by the first N values of $\langle X_n \rangle$, as defined in (42), satisfies

$$D_N^{(t)} = O\left(\frac{\sqrt{s} \log s (\log m)^t}{N}\right) + O\left(\frac{m(\log m)^t}{N\sqrt{s}}\right) + O((\log m)^t r_{\max}); \quad (54)$$

$$D_m^{(t)} = O((\log m)^t r_{\max}). \quad (55)$$

Here r_{\max} is the maximum value of the quantity $r(u_1, \dots, u_t)$ defined in (46), taken over all nonzero integer vectors (u_1, \dots, u_t) satisfying (15).

Proof. The first two O terms in (54) come from vectors (u_1, \dots, u_t) in (44) that do not satisfy (15), since exercise 25 proves that $f(u_1, \dots, u_t)$ summed over all (u_1, \dots, u_t) is $O(((2/\pi)\ln m)^t)$ and exercise 26 bounds each $g(u_1, \dots, u_t)$. (These terms are missing from (55) since $g(u_1, \dots, u_t) = 0$ in that case.) The remaining O term in (54) and (55) comes from nonzero vectors (u_1, \dots, u_t) that do satisfy (15), using the bound derived in exercise 27. (By examining this proof carefully, we could replace each O in these formulas by an explicit function of t .) ■

Eq. (55) relates to the serial test in t dimensions over the entire period, while Eq. (54) gives us useful information about the distribution of the first N generated values when N is less than m , provided that N is not too small. Note that (54) will guarantee low discrepancy only when s is sufficiently large, otherwise the m/\sqrt{s} term will dominate. If $m = p_1^{e_1} \dots p_r^{e_r}$ and $\gcd(a-1, m) = p_1^{f_1} \dots p_r^{f_r}$, then s equals $p_1^{e_1-f_1} \dots p_r^{e_r-f_r}$ (cf. Lemma 3.2.1.2P); thus, the largest values of s correspond to high potency. In the common case $m = 2^e$ and $a \equiv 5$ (modulo 8), we have $s = \frac{1}{4}m$, so $D_N^{(t)}$ is $O(\sqrt{m}(\log m)^{t+1}/N) + O((\log m)^t r_{\max})$. It is not difficult to prove that $r_{\max} \leq \sqrt{2}/\nu_t$ unless ν_t is very small (see exercise 29); therefore Eq. (54) says in particular that the discrepancy will be low in t dimensions if the spectral test is passed and if N is somewhat larger than $\sqrt{m}(\log m)^{t+1}$.

In a sense Theorem N is almost too strong, for the result in exercise 30 shows that linear congruential sequences like those in lines 8, 19, and 23 of Table 1 have a discrepancy of order $(\log m)^2/m$ in two dimensions. The discrepancy

in this case is extremely small in spite of the fact that there are parallelogram-shaped regions of area $\approx 1/\sqrt{m}$ containing no points (U_n, U_{n+1}) . The fact that discrepancy can change so drastically when the points are rotated warns us that the serial test may not be as meaningful a measure of randomness as the rotation-invariant spectral test.

G. Historical remarks. In 1959, while deriving upper bounds for the error in the evaluation of t -dimensional integrals by the Monte Carlo method, N. M. Korobov devised a way to rate the multiplier of a linear congruential sequence. His formula (which is rather complicated) is related to the spectral test since it is strongly influenced by “small” solutions to (15); but it is not quite the same. Korobov’s test has been the subject of an extensive literature, surveyed by Kuipers and Niederreiter in *Uniform Distribution of Sequences* (New York: Wiley, 1974), §2.5.

The spectral test was originally formulated by R. R. Coveyou and R. D. MacPherson [JACM 14 (1967), 100–119], who introduced it in an interesting indirect way. Instead of working with the grid structure of successive points, they considered random number generators as sources of t -dimensional “waves.” The numbers $\sqrt{x_1^2 + \cdots + x_t^2}$ such that $x_1 + \cdots + a^{t-1}x_t \equiv 0$ (modulo m) in their original treatment were the wave “frequencies,” or points in the “spectrum” defined by the random number generator, with low-frequency waves being the most damaging to randomness; hence the name *spectral test*. Coveyou and MacPherson introduced a procedure analogous to Algorithm S for performing their test, based on the principle of Lemma A. However, the original procedure (which used matrices UU^T and VV^T instead of U and V) dealt with extremely large numbers; the idea of working directly with U and V was independently suggested by F. Janssens and by U. Dieter.

Several other authors pointed out that the spectral test could be understood in far more concrete terms; by introducing the study of the grid and lattice structures corresponding to linear congruential sequences, the fundamental limitations on randomness became graphically clear. See G. Marsaglia, Proc. Nat. Acad. Sci. 61 (1968), 25–28; W. W. Wood, J. Chem. Phys. 48 (1968), 427; R. R. Coveyou, Studies in Applied Math. 3 (1969), 70–112; W. A. Beyer, R. B. Roof, and D. Williamson, Math. Comp. 25 (1971), 345–360; G. Marsaglia and W. A. Beyer, Applications of Number Theory to Numerical Analysis, ed. by S. K. Zaremba (New York: Academic Press, 1972), 249–285, 361–370.

Harald Niederreiter’s papers concerning the use of exponential sums to study the distribution of linear congruential sequences have appeared in Math. Comp. 26 (1972), 793–795; 28 (1974), 1117–1132; 30 (1976), 571–597; Advances in Math. 26 (1977), 99–181 [this is the most important paper of the series]; and Bull. Amer. Math. Soc. 84 (1978), 273–274, 957–1041 [this one summarizes the others and contains an extensive bibliography].

EXERCISES

- [M10] To what does the spectral test reduce in one dimension? (In other words,

what happens when $t = 1$?)

2. [HM20] Let V_1, \dots, V_t be linearly independent vectors in t -space, let L_0 be the lattice of points defined by (10), and let U_1, \dots, U_t be defined by (19). Prove that the maximum distance between $(t - 1)$ -dimensional hyperplanes, over all families of parallel hyperplanes that cover L_0 , is $1/\min\{f(x_1, \dots, x_t) \mid (x_1, \dots, x_t) \neq (0, \dots, 0)\}$, where f is defined in (17).

3. [M24] Determine ν_3 and ν_4 for all linear congruential generators of potency 2 and period length m .

► **4.** [M23] Let $u_{11}, u_{12}, u_{21}, u_{22}$ be elements of a 2×2 integer matrix such that $u_{11} + au_{12} \equiv u_{21} + au_{22} \equiv 0$ (modulo m) and $u_{11}u_{22} - u_{21}u_{12} = m$. (a) Prove that all integer solutions (y_1, y_2) to the congruence $y_1 + ay_2 \equiv 0$ (modulo m) have the form $(y_1, y_2) = (x_1u_{11} + x_2u_{21}, x_1u_{12} + x_2u_{22})$ for integer x_1, x_2 . (b) If, in addition, $2|u_{11}u_{21} + u_{12}u_{22}| \leq u_{11}^2 + u_{12}^2 \leq u_{21}^2 + u_{22}^2$, prove that $(y_1, y_2) = (u_{11}, u_{12})$ minimizes $y_1^2 + y_2^2$ over all nonzero solutions to the congruence.

5. [M30] Prove that steps S1 through S3 of Algorithm S correctly perform the spectral test in two dimensions. [Hint: See exercise 4, and prove that $(h' + h)^2 + (p' + p)^2 \geq h^2 + p^2$ at the beginning of step S2.]

6. [M30] Let a_0, a_1, \dots, a_{t-1} be the partial quotients of a/m as defined in Section 3.3.3, and let $A = \max_{0 \leq j < t} a_j$. Prove that $\mu_2 > 2\pi/(A + 1 + 1/A)$.

7. [HM22] Prove that “question (a)” and “question (b)” of the text have the same solution for real values of $q_1, \dots, q_{j-1}, q_{j+1}, \dots, q_t$ (cf. (24), (26)).

8. [M16] Line 18 of Table 1 has a very low value of μ_2 , yet μ_3 is quite satisfactory. What is the highest possible value of μ_3 when $\mu_2 = 10^{-6}$ and $m = 10^{10}$?

9. [HM32] (C. Hermite, 1846.) Let $f(x_1, \dots, x_t)$ be a positive definite quadratic form, defined by the matrix U as in (17), and let θ be the minimum value of f at nonzero integer points. Prove that $\theta \leq (\frac{1}{3})^{(t-1)/2} |\det U|^{2/t}$. [Hints: If W is any integer matrix of determinant 1, the matrix WU defines a form equivalent to f ; and if S is any orthogonal matrix (i.e., $S^{-1} = S^T$), the matrix US defines a form identically equal to f . Show that there is an equivalent form g whose minimum θ occurs at $(1, 0, \dots, 0)$. Then prove the general result by induction on t , writing $g(x_1, \dots, x_t) = \theta(x_1 + \beta_2 x_2 + \dots + \beta_t x_t)^2 + h(x_2, \dots, x_t)$ where h is a positive definite quadratic form in $t - 1$ variables.]

10. [M28] Let (y_1, y_2) be relatively prime integers such that $y_1 + ay_2 \equiv 0$ (modulo m) and $y_1^2 + y_2^2 < \sqrt{4/3}m$. Show that there exist integers (u_1, u_2) such that $u_1 + au_2 \equiv 0$ (modulo m), $u_1y_2 - u_2y_1 = m$, $2|u_1y_1 + u_2y_2| \leq \min(u_1^2 + u_2^2, y_1^2 + y_2^2)$, and $(u_1^2 + u_2^2) \times (y_1^2 + y_2^2) \geq m^2$. (Hence by exercise 4, $\nu_2^2 = \min(u_1^2 + u_2^2, y_1^2 + y_2^2)$.)

► **11.** [HM30] (Alan G. Waterman, 1974.) Invent a reasonably efficient procedure that computes multipliers $a \equiv 1$ (modulo 4) for which there exists a relatively prime solution to the congruence $y_1 + ay_2 \equiv 0$ (modulo m) with $y_1^2 + y_2^2 = \sqrt{4/3}m - \epsilon$, where $\epsilon > 0$ is as small as possible, given $m = 2^\epsilon$. (By exercise 10, this choice of a will guarantee that $\nu_2^2 \geq m^2/(y_1^2 + y_2^2) > \sqrt{3/4}m$, and there is a chance that ν_2^2 will be near its optimum value $\sqrt{4/3}m$. In practice we will compute several such multipliers having small ϵ , choosing the one with best spectral values ν_2, ν_3, \dots)

12. [HM23] Prove, without geometrical handwaving, that any solution to the text’s “question (b)” must also satisfy the set of equations (26).

13. [HM22] Lemma A uses the fact that U is nonsingular to prove that a positive definite quadratic form attains a definite, nonzero minimum value at nonzero integer points. Show that this hypothesis is necessary, by exhibiting a quadratic form (19) whose matrix of coefficients is singular, and for which the values of $f(x_1, \dots, x_t)$ get arbitrarily near zero (but never reach it) at nonzero integer points (x_1, \dots, x_t) .

14. [24] Perform Algorithm S by hand, for $m = 100$, $a = 41$, $T = 3$.

► 15. [M20] Let U be an integer vector satisfying (15). How many of the $(t - 1)$ -dimensional hyperplanes defined by U intersect the unit hypercube $\{(x_1, \dots, x_t) \mid 0 \leq x_j < 1 \text{ for } 1 \leq j \leq t\}$? (This is approximately the number of hyperplanes in the family that will suffice to cover L_0 .)

16. [M90] (U. Dieter.) Show how to modify Algorithm S in order to calculate the minimum number N_t of parallel hyperplanes intersecting the unit hypercube as in exercise 15, over all U satisfying (15). [Hint: What are appropriate analogs to positive definite quadratic forms and to Lemma A?]

17. [20] Modify Algorithm S so that, in addition to computing the quantities ν_t , it outputs all integer vectors (u_1, \dots, u_t) satisfying (15) such that $u_1^2 + \dots + u_t^2 = \nu_t^2$, for $2 \leq t \leq T$.

18. [M90] (a) Let $m = 2^\epsilon$, where ϵ is even. By considering “combinatorial matrices,” i.e., matrices whose elements have the form $y + x\delta_{ij}$ (cf. exercise 1.2.3–39), find 3×3 matrices of integers U and V satisfying (29) such that the transformation of step S5 does nothing for any j , but the corresponding values of z_k in (31) are so huge that exhaustive search is out of the question. (The matrix U need not satisfy (28), we are interested here in arbitrary positive definite quadratic forms of determinant m .) (b) Although transformation (23) is of no use for the matrices constructed in (a), find another transformation that does produce a substantial reduction.

► 19. [HM25] Suppose step S5 were changed slightly, so that a transformation with $q = 1$ would be performed when $2V_i \cdot V_j = V_j \cdot V_j$. (Thus, $q = \lfloor (V_i \cdot V_j / V_j \cdot V_j) + \frac{1}{2} \rfloor$ in all cases.) Would it be possible for Algorithm S to get into an infinite loop?

20. [M21] Discuss how to carry out an appropriate spectral test for linear congruential sequences having $c = 0$, X_0 odd, $m = 2^\epsilon$, $a \bmod 8 = 5$.

21. [M20] (R. W. Gosper.) A certain application uses random numbers in batches of four, but “throws away” the second of each set. How can we study the grid structure of $\{\frac{1}{m}(X_{4n}, X_{4n+2}, X_{4n+3})\}$, given a linear congruential generator of period $m = 2^\epsilon$?

22. [M46] What is the best upper bound on μ_3 , given that μ_2 is very near its maximum value $\sqrt{4/3}\pi$? What is the best upper bound on μ_2 , given that μ_3 is very near its maximum value $\frac{4}{3}\pi\sqrt{2}$?

23. [M46] Let U_i, V_j be vectors of real numbers with $U_i \cdot V_j = \delta_{ij}$ for $1 \leq i, j \leq t$, and such that $U_i \cdot U_i = 1$, $2|U_i \cdot U_j| \leq 1$, $2|V_i \cdot V_j| \leq V_j \cdot V_j$ for $i \neq j$. How large can $V_1 \cdot V_1$ be? (This question relates to the bounds in step S8, if both (23) and the transformation of exercise 18(b) fail to make any reductions. The maximum value known to be achievable is $(n + 2)/3$, which occurs when $U_1 = I_1$, $U_j = \frac{1}{2}I_1 + \frac{1}{2}\sqrt{3}I_j$, $V_1 = I_1 - (I_2 + \dots + I_n)/\sqrt{3}$, $V_j = 2I_j/\sqrt{3}$, for $2 \leq j \leq n$, where (I_1, \dots, I_n) is the identity matrix; this construction is due to B. V. Alekseev [Matematicheskie Zametki, to appear].)

► 24. [M28] Generalize the spectral test to second-order sequences of the form $X_n = (aX_{n-1} + bX_{n-2}) \bmod p$, having period length $p^2 - 1$. (Cf. Eq. 3.2.2–8.) How should Algorithm S be modified?

25. [HM24] Let d be a divisor of m and let $0 \leq q < d$. Prove that $\sum r(k)$, summed over all $0 \leq k < m$ such that $k \bmod d = q$, is at most $(2/d\pi) \ln(m/d) + O(1)$. (Here $r(k)$ is defined in Eq. (46) when $t = 1$.)

26. [M22] Explain why the derivation of (53) leads to a similar bound on

$$\left| N^{-1} \sum_{0 \leq n < N} \omega^{qx_n} \right|$$

for $0 < q < m$. Where does the derivation of (53) break down when $m = 1$?

27. [HM39] (E. Hlawka, H. Niederreiter.) Let $r(u_1, \dots, u_m)$ be the function defined in (46). Prove that $\sum r(u_1, \dots, u_t)$, summed over all $0 \leq u_1, \dots, u_t < m$ such that $(u_1, \dots, u_t) \neq (0, \dots, 0)$ and (15) holds, is $O((2\pi \lg m)^t r_{\max})$, where r_{\max} is the maximum term $r(u_1, \dots, u_t)$ in the sum.

► 28. [M28] (H. Niederreiter.) Find an analog of Theorem N for the case $m = \text{prime}$, $c = 0$, $a = \text{primitive root modulo } m$, $X_0 \not\equiv 0 \pmod{m}$. [Hint: Your exponential sums should involve $\zeta = e^{2\pi i/(m-1)}$ as well as ω .] Prove that in this case the “average” primitive root has discrepancy $D_{m-1}^{(t)} = O(t(\log m)^t / \phi(m-1))$, hence good primitive roots exist for all m .

29. [M21] Prove that quantity r_{\max} of exercise 27 is never larger than $\sqrt{2}/\nu_t$, unless $\nu_t^2 \leq 2(t-1)$.

30. [M33] (S. K. Zaremba.) Prove that in two dimensions, $r_{\max} \leq m/\max(a_1, \dots, a_s)$, where a_1, \dots, a_s are the partial quotients obtained when Euclid’s algorithm is applied to m and a . [Hint: We have $a/m = |a_1, \dots, a_s|$ in the notation of Section 4.5.3; apply exercise 4.5.3–42.]

31. [HM47] (I. Borosh.) Prove that for all sufficiently large m there exists a number a relatively prime to m such that all partial quotients of a/m are ≤ 3 . Furthermore the set of all m satisfying this condition but with all partial quotients ≤ 2 has positive density.

3.4. OTHER TYPES OF RANDOM QUANTITIES

WE HAVE NOW SEEN how to make a computer generate a sequence of numbers U_0, U_1, U_2, \dots that behaves as if each number were independently selected at random between zero and one with the uniform distribution. Applications of random numbers often call for other kinds of distributions, however; for example, if we want to make a random choice from among k alternatives, we want a random integer between 1 and k . If some simulation process calls for a random waiting time between occurrences of independent events, a random number with the “exponential distribution” is desired. Sometimes we don’t even want random numbers—we want a random permutation (i.e., a random arrangement of n objects) or a random combination (i.e., a random choice of k objects from a collection of n).

In principle, any of these other random quantities may be obtained from the uniform deviates U_0, U_1, U_2, \dots . People have devised a number of important “random tricks” that may be used to perform these manipulations efficiently on a computer, and a study of these techniques also gives some insight into the proper use of random numbers in any Monte Carlo application.

It is conceivable that someday somebody will invent a random number generator that produces one of these other random quantities *directly*, instead of getting it indirectly via the uniform distribution. But except for the “random bit” generator described in Section 3.2.2, no direct methods have so far proved to be practical.

The discussion in the following section assumes the existence of a random sequence of uniformly distributed real numbers between zero and one. A new uniform deviate U is generated whenever we need it. These numbers are usually represented in a computer word with the decimal point assumed at the left.

3.4.1. Numerical Distributions

This section summarizes the best techniques known for producing numbers from various important distributions. Many of the methods were originally suggested by John von Neumann in the early 1950s, and they have gradually been improved upon by other people, notably George Marsaglia, J. H. Ahrens, and U. Dieter.

A. Random choices from a finite set. The simplest and most common type of distribution required in practice is a random integer. An integer between 0 and 7 can be extracted from three bits of U on a binary computer; in such a case, these bits should be extracted from the *most significant* (left-hand) part of the computer word, since the least significant bits produced by many random number generators are not sufficiently random. (See the discussion in Section 3.2.1.1.)

In general, to get a random integer X between 0 and $k - 1$, we can multiply by k , and let $X = \lfloor kU \rfloor$. On MIX, we would write

```
LDA U
MUL K
```

(1)

and after these two instructions have been executed the desired integer will appear in register A. If a random integer between 1 and k is desired, we add one to this result. (The instruction "INCA 1" would follow (1).)

This method gives each integer with nearly equal probability. There is a slight error because the computer word size is finite (see exercise 2); but the error is quite negligible if k is small, for example if $k/m < 1/10000$.

In a more general situation we might want to give different weights to different integers. Suppose that the value $X = x_1$ is to be obtained with probability p_1 , and $X = x_2$ with probability $p_2, \dots, X = x_k$ with probability p_k . We can generate a uniform number U and let

$$X = \begin{cases} x_1, & \text{if } 0 \leq U < p_1; \\ x_2, & \text{if } p_1 \leq U < p_1 + p_2; \\ \vdots & \\ x_k, & \text{if } p_1 + p_2 + \dots + p_{k-1} \leq U < 1. \end{cases} \quad (2)$$

(Note that $p_1 + p_2 + \dots + p_k = 1$.)

There is a "best possible" way to do the comparisons of U against various values of $p_1 + p_2 + \dots + p_s$, as implied in (2); this situation is discussed in Section 2.3.4.5. Special cases can be handled by more efficient methods; for example, to obtain one of the eleven values 2, 3, ..., 12 with the respective "dice" probabilities $\frac{1}{36}, \frac{2}{36}, \dots, \frac{6}{36}, \dots, \frac{2}{36}, \frac{1}{36}$, we could compute two independent random integers between 1 and 6 and add them together.

However, none of the above techniques is really the fastest general method for selecting x_1, \dots, x_k with the correct probabilities. An ingenious way to do the trick has been discovered by A. J. Walker [*Electronics Letters* 10,8 (1974), 127–128; *ACM Trans. Math. Software* 3 (1976), 253–256]. Suppose we form kU and consider the integer part $K = \lfloor kU \rfloor$ and fraction part $V = (kU) \bmod 1$ separately; for example, after the code (1) we will have K in register A and V in register X. Then we can always obtain the desired distribution by doing the operations

$$\text{if } V < P_K \text{ then } X \leftarrow x_{K+1} \text{ otherwise } X \leftarrow Y_K, \quad (3)$$

for some appropriate tables (P_0, \dots, P_{k-1}) and (Y_0, \dots, Y_{k-1}) . Exercise 7 shows how such tables can be computed in general. Walker's method is sometimes called the method of "aliases."

On a binary computer it is usually helpful to assume that k is a power of 2, so that multiplication can be replaced by shifting; this can be done without loss of generality by introducing additional x 's that occur with probability zero. For example, let's consider dice again; suppose we want $X = j$ to occur with the following 16 probabilities:

$$\begin{aligned} j &= 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \\ p_j &= 0 \ 0 \ \frac{1}{36} \ \frac{2}{36} \ \frac{3}{36} \ \frac{4}{36} \ \frac{5}{36} \ \frac{6}{36} \ \frac{5}{36} \ \frac{4}{36} \ \frac{3}{36} \ \frac{2}{36} \ \frac{1}{36} \ 0 \ 0 \ 0 \end{aligned}$$

We can do this using (3), if $k = 16$ and $x_j = j$ for $0 \leq j < 16$, and if the P and Y tables are set up as follows:

$$\begin{array}{ccccccccccccccccc} P_j & = & 0 & 0 & \frac{4}{9} & \frac{8}{9} & 1 & \frac{7}{9} & 1 & 1 & 1 & \frac{7}{9} & \frac{7}{9} & \frac{8}{9} & \frac{4}{9} & 0 & 0 & 0 \\ Y_j & = & 5 & 9 & 7 & 4 & * & 6 & * & * & * & 8 & 4 & 7 & 10 & 6 & 7 & 8 \end{array}$$

(When $P_j = 1$, Y_j is not used.) For example, the value 7 occurs with probability $\frac{1}{16} \cdot ((1 - P_2) + P_7 + (1 - P_{11}) + (1 - P_{14})) = \frac{6}{36}$ as required. It is a peculiar way to throw dice, but the results are indistinguishable from the real thing.

B. General methods for continuous distributions. The most general real-valued distribution may be expressed in terms of its “distribution function” $F(x)$, which specifies the probability that a random quantity X will not exceed x :

$$F(x) = \text{probability that } (X \leq x). \quad (4)$$

This function always increases monotonically from zero to one; i.e.,

$$F(x_1) \leq F(x_2), \quad \text{if } x_1 \leq x_2; \quad F(-\infty) = 0, \quad F(+\infty) = 1. \quad (5)$$

Examples of distribution functions are given in Section 3.3.1, Fig. 3. If $F(x)$ is continuous and strictly increasing (so that $F(x_1) < F(x_2)$ when $x_1 < x_2$), it takes on all values between zero and one, and there is an *inverse function* $F^{-1}(y)$ such that, for $0 < y < 1$,

$$y = F(x) \quad \text{if and only if} \quad x = F^{-1}(y). \quad (6)$$

In general we can compute a random quantity X with the continuous, strictly increasing distribution $F(x)$ by setting

$$X = F^{-1}(U), \quad (7)$$

where U is uniform; this works because the probability that $X \leq x$ is the probability that $F^{-1}(U) \leq x$, i.e., the probability that $U \leq F(x)$, and this is $F(x)$.

The problem now reduces to one of numerical analysis, namely to find good methods for evaluating $F^{-1}(U)$ to the desired accuracy. Numerical analysis lies outside the scope of this seminumerical book; yet there are a number of important shortcuts available to speed up this general approach, and we will consider them here.

In the first place, if X_1 is a random variable having the distribution $F_1(x)$ and if X_2 is an independent random variable with the distribution $F_2(x)$, then

$$\begin{aligned} \max(X_1, X_2) & \quad \text{has the distribution} & F_1(x)F_2(x), \\ \min(X_1, X_2) & \quad \text{has the distribution} & F_1(x) + F_2(x) - F_1(x)F_2(x). \end{aligned} \quad (8)$$

(See exercise 4.) For example, a uniform deviate U has the distribution $F(x) = x$, for $0 \leq x \leq 1$; if U_1, U_2, \dots, U_t are independent uniform deviates, then

$\max(U_1, U_2, \dots, U_t)$ has the distribution function $F(x) = x^t$, for $0 < x \leq 1$. This is the basis of the “maximum-of- t test” given in Section 3.3.2. Note that the inverse function in this case is $F^{-1}(y) = \sqrt[t]{y}$. In the special case $t = 2$, we see therefore that the two formulas

$$X = \sqrt{U} \quad \text{and} \quad X = \max(U_1, U_2) \quad (9)$$

will give equivalent distributions to the random variable X , although this is not obvious at first glance. We need not take the square root of a uniform deviate.

The number of tricks like this is endless: any algorithm that employs random numbers as input will give a random quantity with some distribution as output. The problem is to find general methods for constructing the algorithm, given the distribution function of the output. Instead of discussing such methods in purely abstract terms, we shall study how they can be applied in important cases.

C. The normal distribution. Perhaps the most important nonuniform, continuous distribution is the so-called *normal distribution with mean zero and standard deviation one*:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt. \quad (10)$$

The significance of this distribution was indicated in Section 1.2.10. Note that the inverse function F^{-1} is not especially easy to compute; but we shall see that several other techniques are available.

(1) *The polar method*, due to G. E. P. Box, M. E. Muller, and G. Marsaglia. (See *Annals Math. Stat.* **28** (1958), 610; and Boeing Scientific Res. Lab. report D1-82-0203 (1962).)

Algorithm P (Polar method for normal deviates). This algorithm calculates two independent normally distributed variables, X_1 and X_2 .

P1. [Get uniform variables.] Generate two independent random variables, U_1 , U_2 , uniformly distributed between zero and one. Set $V_1 \leftarrow 2U_1 - 1$, $V_2 \leftarrow 2U_2 - 1$. (Now V_1 and V_2 are uniformly distributed between -1 and $+1$. On most computers it will be preferable to have V_1 and V_2 represented in floating point form at this point.)

P2. [Compute S .] Set $S \leftarrow V_1^2 + V_2^2$.

P3. [Is $S \geq 1$?] If $S \geq 1$, return to step P1. (Steps P1 through P3 are executed 1.27 times on the average, with a standard deviation of 0.587; see exercise 6.)

P4. [Compute X_1, X_2 .] Set X_1, X_2 according to the following two equations:

$$X_1 = V_1 \sqrt{\frac{-2 \ln S}{S}}, \quad X_2 = V_2 \sqrt{\frac{-2 \ln S}{S}}. \quad (11)$$

These are the normally distributed variables desired. ■

To prove the validity of this method, we use elementary analytic geometry and calculus: If $S < 1$ in step P3, the point in the plane with Cartesian coordinates (V_1, V_2) is a *random point uniformly distributed inside the unit circle*. Transforming to polar coordinates $V_1 = R \cos \Theta$, $V_2 = R \sin \Theta$, we find $S = R^2$, $X_1 = \sqrt{-2 \ln S} \cos \Theta$, $X_2 = \sqrt{-2 \ln S} \sin \Theta$. Using also the polar coordinates $X_1 = R' \cos \Theta'$, $X_2 = R' \sin \Theta'$, we find that $\Theta' = \Theta$ and $R' = \sqrt{-2 \ln S}$. It is clear that R' and Θ' are independent, since R and Θ are independent inside the unit circle. Also, Θ' is uniformly distributed between 0 and 2π ; and the probability that $R' \leq r$ is the probability that $-2 \ln S \leq r^2$, i.e., the probability that $S \geq e^{-r^2/2}$. This equals $1 - e^{-r^2/2}$, since $S = R^2$ is uniformly distributed between zero and one. The probability that R' lies between r and $r + dr$ is therefore the derivative of $1 - e^{-r^2/2}$, namely, $re^{-r^2/2} dr$. Similarly, the probability that Θ' lies between θ and $\theta + d\theta$ is $(1/2\pi) d\theta$. The joint probability that $X_1 \leq x_1$ and that $X_2 \leq x_2$ now can be computed, it is

$$\begin{aligned} & \int_{\{(r,\theta) | r \cos \theta \leq x_1, r \sin \theta \leq x_2\}} \frac{1}{2\pi} e^{-r^2/2} r dr d\theta \\ &= \frac{1}{2\pi} \int_{\{(x,y) | x \leq x_1, y \leq x_2\}} e^{-(x^2+y^2)/2} dx dy \\ &= \left(\sqrt{\frac{1}{2\pi}} \int_{-\infty}^{x_1} e^{-x^2/2} dx \right) \left(\sqrt{\frac{1}{2\pi}} \int_{-\infty}^{x_2} e^{-y^2/2} dy \right). \end{aligned}$$

This proves that X_1 and X_2 are independent and normally distributed, as desired.

(2) *The rectangle-wedge-tail method*, introduced by G. Marsaglia. In this method we use the distribution

$$F(x) = \sqrt{\frac{2}{\pi}} \int_0^x e^{-t^2/2} dt, \quad x \geq 0, \quad (12)$$

so that $F(x)$ gives the distribution of the *absolute value* of a normal deviate. After X has been computed according to distribution (12), we will attach a random sign to its value, and this will make it a true normal deviate.

The rectangle-wedge-tail approach is based on several important general techniques that we shall explore as we develop the algorithm. The first key idea is to regard $F(x)$ as a mixture of several other functions, namely to write

$$F(x) = p_1 F_1(x) + p_2 F_2(x) + \cdots + p_n F_n(x), \quad (13)$$

where F_1, F_2, \dots, F_n are appropriate distributions and p_1, p_2, \dots, p_n are nonnegative probabilities that sum to 1. If we generate a random variable X by choosing distribution F_j with probability p_j , it is easy to see that X will have distribution F overall. Some of the distributions $F_j(x)$ may be rather difficult to handle, even harder than F itself, but we can usually arrange things so that the

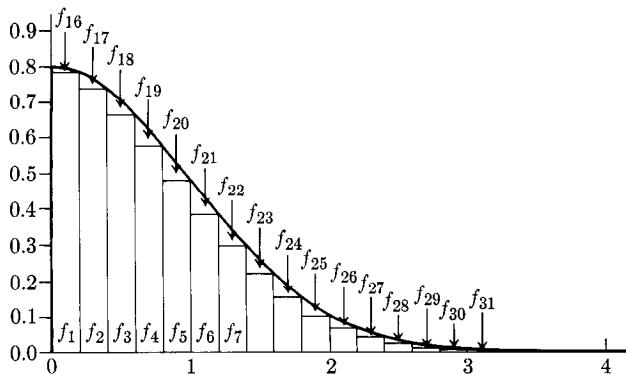


Fig. 9. The density function divided into 31 parts. The area of each part represents the average number of times a random number with that density is to be computed.

probability p_j is very small in this case. Most of the distributions $F_j(x)$ will be quite easy to accommodate, since they will be trivial modifications of the uniform distribution. The resulting method yields an extremely efficient program, since its *average* running time is very small.

It is easier to understand the method if we work with the *derivatives* of the distributions instead of the distributions themselves. Let

$$f(x) = F'(x), \quad f_j(x) = F'_j(x);$$

these are called the *density functions* of the probability distributions. Equation (13) becomes

$$f(x) = p_1 f_1(x) + p_2 f_2(x) + \cdots + p_n f_n(x). \quad (14)$$

Each $f_j(x)$ is ≥ 0 , and the total area under the graph of $f_j(x)$ is 1; so there is a convenient graphical way to display the relation (14): The area under $f(x)$ is divided into n parts, with the part corresponding to $f_j(x)$ having area p_j . See Fig. 9, which illustrates the situation in the case of interest to us here, with $f(x) = F'(x) = \sqrt{2/\pi} e^{-x^2/2}$; the area under this curve has been divided into $n = 31$ parts. There are 15 rectangles, which represent $p_1 f_1(x), \dots, p_{15} f_{15}(x)$; there are 15 wedge-shaped pieces, which represent $p_{16} f_{16}(x), \dots, p_{30} f_{30}(x)$; and the remaining part $p_{31} f_{31}(x)$ is the “tail,” namely the entire graph of $f(x)$ for $x \geq 3$.

The rectangular parts $f_1(x), \dots, f_{15}(x)$ represent *uniform distributions*. For example, $f_3(x)$ represents a random variable uniformly distributed between $\frac{2}{5}$ and $\frac{3}{5}$. The altitude of $p_j f_j(x)$ is $f(j/5)$, hence the area of the j th rectangle is

$$p_j = \frac{1}{5} f(j/5) = \sqrt{\frac{2}{25\pi}} e^{-j^2/50}, \quad \text{for } 1 \leq j \leq 15. \quad (15)$$

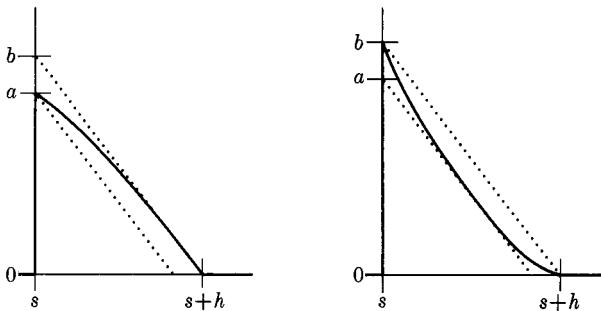


Fig. 10. Density functions for which Algorithm L may be used to generate random numbers.

In order to generate such rectangular portions of the distribution, we simply compute

$$X = \frac{1}{5}U + S, \quad (16)$$

where U is uniform and S takes the value $(j - 1)/5$ with probability p_j . Since $p_1 + \dots + p_{15} = .9183$, we can use simple uniform deviates like this about 92 percent of the time.

In the remaining 8 percent, we will usually have to generate one of the wedge-shaped distributions F_{16}, \dots, F_{30} . Typical examples of what we need to do are shown in Fig. 10. When $x < 1$, the curved part is concave downward, and when $x > 1$ it is concave upward, but in each case the curved part is reasonably close to a straight line, and it can be enclosed in two parallel lines as shown.

To handle these wedge-shaped distributions, we will rely on yet another general technique, von Neumann's *rejection method* for obtaining a complicated density from another one that "encloses" it. The polar method described above is a simple example of such an approach: Steps P1-P3 obtain a random point inside the unit circle by first generating a random point in a larger square, rejecting it and starting over again if the point was outside the circle.

The general rejection method is even more powerful than this. To generate a random variable X with density f , let g be another probability density function such that

$$f(t) \leq cg(t) \quad (17)$$

for all t , where c is a constant. Now generate X according to density g , and also generate an independent uniform deviate U . If $U \geq f(X)/cg(X)$, reject X and start again with another X and U . When the condition $U < f(X)/cg(X)$ finally occurs, the resulting X will have density f as desired. [Proof: $X \leq x$ will occur with probability $p(x) = \int_{-\infty}^x (g(t) dt \cdot f(t)/cg(t)) + qp(x)$, where the quantity $q = \int_{-\infty}^{\infty} (g(t) dt \cdot (1 - f(t)/cg(t))) = 1 - 1/c$ is the probability of rejection; hence $p(x) = \int_{-\infty}^x f(t) dt$.]

The rejection technique is most efficient when c is small, since there will be c iterations on the average before a value is accepted. (See exercise 6.) In some

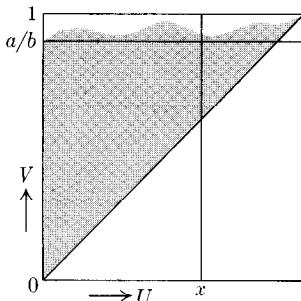


Fig. 11. Region of “acceptance” in Algorithm L.

cases $f(x)/cg(x)$ is always 0 or 1; then U need not be generated. In other cases if $f(x)/cg(x)$ is hard to compute, we may be able to “squeeze” it between two bounding functions

$$r(x) \leq f(x)/cg(x) \leq s(x) \quad (18)$$

that are much simpler, and the exact value of $f(x)/cg(x)$ need not be calculated unless $r(x) \leq U < s(x)$. The following algorithm solves the wedge problem by developing the rejection method still further.

Algorithm L (*Nearly linear densities*). This algorithm may be used to generate a random variable X for any distribution whose density $f(x)$ satisfies the following conditions (cf. Fig. 10):

$$\begin{aligned} f(x) &= 0, && \text{for } x < s \text{ and for } x > s + h; \\ a - b(x-s)/h &\leq f(x) \leq b - b(x-s)/h, && \text{for } s \leq x \leq s + h. \end{aligned} \quad (19)$$

- L1. [Get $U \leq V$.] Generate two independent random variables U, V , uniformly distributed between zero and one. If $U > V$, exchange $U \leftrightarrow V$.
- L2. [Easy case?] If $V \leq a/b$, go to L4.
- L3. [Try again?] If $V > U + (1/b)f(s + hU)$, go back to step L1. (If a/b is close to 1, this step of the algorithm will not be necessary very often.)
- L4. [Compute X .] Set $X \leftarrow s + hU$. ■

When step L4 is reached, the point (U, V) is a random point in the area shaded in Fig. 11, namely, $0 \leq U \leq V \leq U + (1/b)f(s + hU)$. Conditions (19) ensure that

$$\frac{a}{b} \leq U + \frac{1}{b}f(s + hU) \leq 1.$$

Now the probability that $X \leq s + hx$, for $0 \leq x \leq 1$, is the ratio of area to the left of the vertical line $U = x$ in Fig. 11 to the total area, namely,

$$\int_0^x \frac{1}{b}f(s + hu) du / \int_0^1 \frac{1}{b}f(s + hu) du = \int_s^{s+hx} f(v) dv;$$

therefore X has the correct distribution.

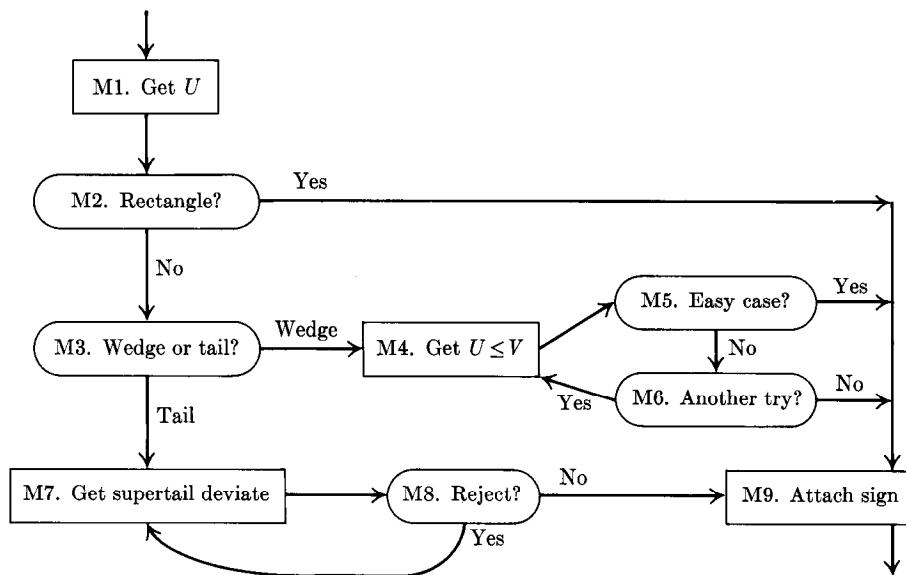


Fig. 12. The “rectangle-wedge-tail” algorithm for generating normal deviates.

With appropriate constants a_j , b_j , s_j , Algorithm L will take care of the wedge-shaped densities f_{j+15} of Fig. 9, for $1 \leq j \leq 15$. The final distribution, F_{31} , needs to be treated only about one time in 370; it is used whenever a result $X \geq 3$ is to be computed. Exercise 11 shows that a standard rejection scheme can be used for this “tail”; hence we are ready to consider the procedure in its entirety:

Algorithm M (Rectangle-wedge-tail method for normal deviates). This algorithm uses a number of auxiliary tables (P_0, \dots, P_{31}) , (Q_1, \dots, Q_{15}) , (Y_0, \dots, Y_{31}) , (Z_0, \dots, Z_{31}) , (S_1, \dots, S_{16}) , (D_{16}, \dots, D_{30}) , (E_{16}, \dots, E_{30}) , constructed as explained in exercise 10; examples appear in Table 1. We assume that a binary computer is being used; a similar procedure could be worked out for decimal machines.

M1. [Get U .] Generate a uniform random number $U = (.b_0 b_1 b_2 \dots b_t)_2$. (Here the b 's are the bits in the binary representation of U . For reasonable accuracy, t should be at least 24.) Set $\psi \leftarrow b_0$. (Later, ψ will be used to determine the sign of the result.)

M2. [Rectangle?] Set $j \leftarrow (b_1 b_2 b_3 b_4 b_5)_2$, a binary number determined by the leading bits of U , and set $f \leftarrow (.b_6 b_7 \dots b_t)_2$, the fraction determined by the remaining bits. If $f \geq P_j$, set $X \leftarrow Y_j + fZ_j$ and go to M9. Otherwise if $j \leq 15$ (i.e., $b_1 = 0$), set $X \leftarrow S_j + fQ_j$ and go to M9. (This is an adaptation of Walker's alias method (3).)

Table 1
EXAMPLE OF TABLES USED WITH ALGORITHM M*

j	P_j	P_{j+16}	Q_j	Y_j	Y_{j+16}	Z_j	Z_{j+16}	S_{j+1}	D_{j+15}	E_{j+15}
0	.000	.067		0.00	0.59	0.20	0.21	0.0		
1	.849	.161	.236	-0.92	0.96	1.32	0.24	0.2	.505	25.00
2	.970	.236	.206	-5.86	-0.06	6.66	0.26	0.4	.773	12.50
3	.855	.285	.234	-0.58	0.12	1.38	0.28	0.6	.876	8.33
4	.994	.308	.201	-33.13	1.31	34.93	0.29	0.8	.939	6.25
5	.995	.304	.201	-39.55	0.31	41.35	0.29	1.0	.986	5.00
6	.933	.280	.214	-2.57	1.12	2.97	0.28	1.2	.995	4.06
7	.923	.241	.217	-1.61	0.54	2.61	0.26	1.4	.987	3.37
8	.727	.197	.275	0.67	0.75	0.73	0.25	1.6	.979	2.86
9	1.000	.152	.200	0.00	0.56	0.00	0.24	1.8	.972	2.47
10	.691	.112	.289	0.35	0.17	0.65	0.23	2.0	.966	2.16
11	.454	.079	.440	-0.17	0.38	0.37	0.22	2.2	.960	1.92
12	.287	.052	.698	0.92	-0.01	0.28	0.21	2.4	.954	1.71
13	.174	.033	1.150	0.36	0.39	0.24	0.21	2.6	.948	1.54
14	.101	.020	1.974	-0.02	0.20	0.22	0.20	2.8	.942	1.40
15	.057	.086	3.526	0.19	0.78	0.21	0.22	3.0	.936	1.27

*In practice, this data would be given with much greater precision; the table shows only enough figures so that interested readers may test their own algorithms for computing the values more accurately.

M3. [Wedge or tail?] (Now $15 \leq j \leq 31$, and each particular value j occurs with probability p_j .) If $j = 31$, go to M7.

M4. [Get $U \leq V$.] Generate two new uniform deviates, U, V ; if $U > V$, exchange $U \leftrightarrow V$. (We are now performing Algorithm L.) Set $X \leftarrow S_{j-15} + \frac{1}{5}U$.

M5. [Easy case?] If $V \leq D_j$, go to M9.

M6. [Another try?] If $V > U + E_j(e^{(S_{j-14}^2 - X^2)/2} - 1)$, go back to step M4; otherwise go to M9. (This step is executed with low probability.)

M7. [Get supertail deviate.] Generate two new independent uniform deviates, U, V , and set $X \leftarrow \sqrt{9 - 2 \ln V}$.

M8. [Reject?] If $UX \geq 3$, go back to step M7. (This will occur only about one-twelfth as often as we reach step M8.)

M9. [Attach sign.] If $\psi = 1$, set $X \leftarrow -X$. ■

This algorithm is a very pretty example of mathematical theory intimately interwoven with programming ingenuity—a fine illustration of the art of computer programming! Only steps M1, M2, and M9 need to be performed most of the time, and the other steps aren't terribly slow either. The first publications of the rectangle-wedge-tail method were by G. Marsaglia, *Annals Math. Stat.* **32** (1961), 894–899; G. Marsaglia, M. D. MacLaren, and T. A. Bray, *CACM* **7** (1964), 4–10. Further refinements of Algorithm M have been developed by G. Marsaglia, K. Ananthanarayanan, and N. J. Paul, *Inf. Proc. Letters* **5** (1976), 27–30.

(3) *The odd-even method*, due to G. E. Forsythe. An amazingly simple technique for generating random deviates with a density of the general exponential form

$$f(x) = Ce^{-h(x)}, \quad \text{for } a \leq x < b, \quad f(x) = 0 \quad \text{otherwise}, \quad (20)$$

when

$$0 \leq h(x) \leq 1 \quad \text{for } a \leq x < b, \quad (21)$$

was discovered by John von Neumann and G. E. Forsythe about 1950. The idea is based on the rejection method described earlier, letting $g(x)$ be the uniform distribution on $[a, b]$: We set $X \leftarrow a + (b - a)U$, where U is a uniform deviate, and then we want to accept X with probability $e^{-h(X)}$. The latter operation could be done by testing $e^{-h(X)}$ vs. V , or $h(X)$ vs. $-\ln V$, when V is another uniform deviate, but the job can be done without applying any transcendental functions in the following interesting way. Set $V_0 \leftarrow h(X)$, then generate uniform deviates V_1, V_2, \dots until finding some $K \geq 1$ with $V_{K-1} < V_K$. For fixed X and k , the probability that $h(X) \geq V_1 \geq \dots \geq V_k$ is $1/k!$ times the probability that $\max(V_1, \dots, V_k) \leq h(X)$, namely $h(X)^k/k!$; hence the probability that $K = k$ is $h(X)^{k-1}/(k-1)! - h(X)^k/k!$, and the probability that K is odd is

$$\sum_{k \text{ odd}, k \geq 1} \left(\frac{h(X)^{k-1}}{(k-1)!} - \frac{h(X)^k}{k!} \right) = e^{-h(X)}. \quad (22)$$

Therefore we reject X and try again if K is even; we accept X as a random variable with density (20) if K is odd. Note that we usually won't have to generate many V 's in order to determine K , since the average value of K (given X) is $\sum_{k \geq 0}$ probability that($K > k$) = $\sum_{k \geq 0} h(X)^k/k! = e^{h(X)} \leq e$.

Forsythe realized some years later that this approach leads to an efficient method for calculating normal deviates, without the need for any auxiliary routines to calculate square roots or logarithms as in Algorithms P and M. His procedure, with an improved choice of intervals $[a, b]$ due to J. H. Ahrens and U. Dieter, can be summarized as follows.

Algorithm F (Odd-even method for normal deviates). This algorithm generates normal deviates on a binary computer, assuming approximately $t + 1$ bits of accuracy. The algorithm requires a table of values $d_j = a_j - a_{j-1}$, for $1 \leq j \leq t + 1$, where a_j is defined by the relation

$$\sqrt{\frac{2}{\pi}} \int_{a_j}^{\infty} e^{-x^2/2} dx = \frac{1}{2^j}. \quad (23)$$

- F1.** [Get U .] Generate a uniform random number $U = (.b_0 b_1 \dots b_t)_2$, where b_0, \dots, b_t denote the bits in binary notation. Set $\psi \leftarrow b_0$, $j \leftarrow 1$, and $a \leftarrow 0$.
- F2.** [Find first zero b_j .] If $b_j = 1$, set $a \leftarrow a + d_j$, $j \leftarrow j + 1$, and repeat this step. (If $j = t + 1$, treat b_j as zero.)
- F3.** [Generate candidate.] (Now $a = a_{j-1}$, and the current value of j occurs with probability $\approx 2^{-j}$. We will generate X in the range $[a_{j-1}, a_j]$, using

the rejection method described above, with $h(x) = x^2/2 - a^2/2 = y^2/2 + ay$ where $y = x - a$. Exercise 12 proves that $h(x) \leq 1$ as required in (21). Set $Y \leftarrow d_j$ times $(.b_{j+1} \dots b_t)_2$ and $V \leftarrow (\frac{1}{2}Y + a)Y$. (Since the average value of j is 2, there will usually be enough significant bits in $(.b_{j+1} \dots b_t)_2$ to provide decent accuracy. The calculations are readily done in fixed point arithmetic.)

F4. [Reject?] Generate a uniform deviate U . If $V < U$, go on to step F5. Otherwise set V to a new uniform deviate; and if now $U < V$ (i.e., if K is even, in the discussion above), go back to F3, otherwise repeat step F4.

F5. [Return X .] Set $X \leftarrow a + Y$. If $\psi = 1$, set $X \leftarrow -X$. ■

Values of d_j for $1 \leq j \leq 47$ appear in a paper by Ahrens and Dieter, *Math. Comp.* 27 (1973), 927–937; their paper discusses refinements of the algorithm that improve its speed at the expense of more tables. Algorithm F is attractive since it is almost as fast as Algorithm M and it is easier to implement. The average number of uniform deviates per normal deviate is 2.53947; R. P. Brent [*CACM* 17 (1974), 704–705] has shown how to reduce this number to 1.37446 at the expense of two subtractions and one division per uniform deviate saved.

(4) *Ratios of uniform deviates.* There is yet another good way to generate normal deviates, discovered by A. J. Kinderman and J. F. Monahan in 1976. Their idea is to generate a random point (U, V) in the region defined by

$$0 < u \leq 1, \quad -2u\sqrt{\ln(1/u)} \leq v \leq 2u\sqrt{\ln(1/u)}, \quad (24)$$

and then to output the ratio $X \leftarrow V/U$. The shaded area of Fig. 13 is the magic region (24) that makes this all work. Before we study the associated theory, let us first state the algorithm so that its efficiency and simplicity are manifest:

Algorithm R (*Ratio method for normal deviates*). This algorithm generates normal deviates X .

R1. [Get U, V .] Generate two independent uniform deviates U and V , where U is nonzero, and set $X \leftarrow \sqrt{8/e}(V - \frac{1}{2})/U$. (Now X is the ratio of the coordinates $(U, \sqrt{8/e}(V - \frac{1}{2}))$ of a random point in the rectangle that encloses the shaded region in Fig. 13. We will accept X if the corresponding point actually lies “in the shade,” otherwise we will try again.)

R2. [Optional upper bound test.] If $X^2 \leq 5 - 4e^{1/4}U$, output X and terminate the algorithm. (This step can be omitted if desired; it tests whether or not the selected point is in the interior region of Fig. 13, making it unnecessary to calculate a logarithm.)

R3. [Optional lower bound test.] If $X^2 \geq 4e^{-1.35}/U + 1.4$, go back to R1. (This step could also be omitted; it tests whether or not the selected point is outside the exterior region of Fig. 13, making it unnecessary to calculate a logarithm.)

R4. [Final test.] If $X^2 \leq -4/\ln U$, output X and terminate the algorithm. Otherwise go back to R1. ■

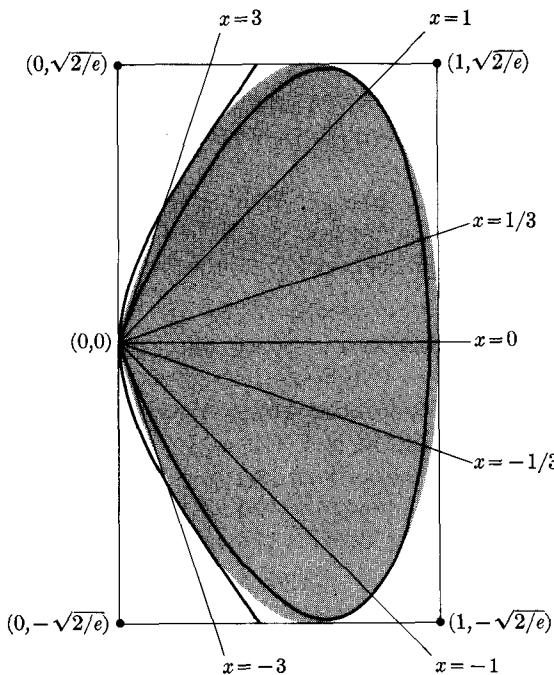


Fig. 13. Region of “acceptance” in the ratio - of - uniforms method for normal deviates. Lengths of lines with coordinate ratio x have the normal distribution.

Exercises 20 and 21 work out the timing analysis; four different algorithms are analyzed, since steps R2 and R3 can be included or omitted depending on one's preference. The following table shows how many times each step will be performed, on the average, depending on which of the optional tests is applied:

Step	Neither	R2 only	R3 only	Both
R1	1.369	1.369	1.369	1.369
R2	0	1.369	0	1.369
R3	0	0	1.369	0.467
R4	1.369	0.467	1.134	0.232

(25)

Thus it pays to omit the optional tests if there is a very fast logarithm operation, but if the log routine is rather slow it pays to include them.

But why does it work? One reason is that we can calculate the probability that $X \leq x$, and it turns out to be the correct value (10). But such a calculation isn't very easy unless one happens to hit on the right “trick,” and anyway it is better to understand how the algorithm might have been discovered in the first place. Kinderman and Monahan derived it by working out the following theory that can be used with any well-behaved density function $f(x)$ [cf. ACM Trans. Math. Software 3 (1977), 257–260].

In general, suppose that a point (U, V) has been generated uniformly over the region of the (u, v) -plane defined by

$$u > 0, \quad u^2 \leq g(v/u) \quad (26)$$

for some nonnegative integrable function g . If we set $X \leftarrow V/U$, the probability that $X \leq x$ can be calculated by integrating $du dv$ over the region defined by the two relations in (26) plus the auxiliary condition $v/u \leq x$, then dividing by the same integral without this extra condition. Letting $v = tu$ so that $dv = u dt$, the integral becomes

$$\int_{-\infty}^x dt \int_0^{\sqrt{g(t)}} u du = \frac{1}{2} \int_{-\infty}^x g(t) dt.$$

Hence the probability that $X \leq x$ is

$$\int_{-\infty}^x g(t) dt / \int_{-\infty}^{+\infty} g(t) dt. \quad (27)$$

The normal distribution comes out when $g(t) = e^{-t^2/2}$; and the condition $u^2 \leq g(v/u)$ simplifies in this case to $(v/u)^2 \leq -4 \ln u$. It is easy to see that the set of all (u, v) satisfying this relation is entirely contained in the rectangle of Fig. 13.

The bounds in steps R2 and R3 define interior and exterior regions with simpler boundary equations. The well-known inequality

$$e^x \geq 1 + x,$$

which holds for all real numbers x , can be used to show that

$$1 + \ln c - cu \leq -\ln u \leq 1/cu - 1 + \ln c \quad (28)$$

for any constant $c > 0$. Exercise 21 proves that $c = e^{1/4}$ is the best possible constant to use in step R2. The situation is more complicated in step R3, and there doesn't seem to be a simple expression for the optimum c in that case, but computational experiments show that the best value for R3 is approximately $e^{1.35}$. The approximating curves (28) are tangent to the true boundary when $u = 1/c$.

It is possible to obtain a faster method by partitioning the region into subregions, most of which can be handled more quickly. Of course, this means that auxiliary tables will be needed, as in Algorithms M and F.

(5) *Variations of the normal distribution.* So far we have considered the normal distribution with mean zero and standard deviation one. If X has this distribution, then

$$Y = \mu + \sigma X \quad (29)$$

has the normal distribution with mean μ and standard deviation σ . Furthermore, if X_1 and X_2 are independent normal deviates with mean zero and standard deviation one, and if

$$Y_1 = \mu_1 + \sigma_1 X_1, \quad Y_2 = \mu_2 + \sigma_2 (\rho X_1 + \sqrt{1 - \rho^2} X_2), \quad (30)$$

then Y_1 and Y_2 are dependent random variables, normally distributed with means μ_1 , μ_2 and standard deviations σ_1 , σ_2 , and with correlation coefficient ρ . (For a generalization to n variables, see exercise 13.)

D. The exponential distribution. After uniform deviates and normal deviates, the next most important random quantity is an *exponential deviate*. Such numbers occur in “arrival time” situations; for example, if a radioactive substance emits alpha particles at a rate so that one particle is emitted every μ seconds on the average, then the time between two successive emissions has the exponential distribution with mean μ . This distribution is defined by the formula

$$F(x) = 1 - e^{-x/\mu}, \quad x \geq 0. \quad (31)$$

(1) *Logarithm method.* Clearly, if $y = F(x) = 1 - e^{-x/\mu}$, then $x = F^{-1}(y) = -\mu \ln(1 - y)$. Therefore by Eq. (7), $-\mu \ln(1 - U)$ has the exponential distribution. Since $1 - U$ is uniformly distributed when U is, we conclude that

$$X = -\mu \ln U \quad (32)$$

is exponentially distributed with mean μ . (The case $U = 0$ must be avoided.)

(2) *Random minimization method.* We saw in Algorithm F that there are simple and fast alternatives to calculating the logarithm of a uniform deviate. The following especially efficient approach has been developed by G. Marsaglia, M. Sibuya, and J. H. Ahrens.

Algorithm S (Exponential distribution with mean μ). This algorithm produces exponential deviates on a binary computer, using uniform deviates with t -bit accuracy. The constants

$$Q[k] = \frac{\ln 2}{1!} + \frac{(\ln 2)^2}{2!} + \cdots + \frac{(\ln 2)^k}{k!}, \quad k \geq 1, \quad (33)$$

should be precomputed, extending until $Q[k] > 1 - 2^{1-t}$.

- S1. [Get U and shift.] Generate a t -bit uniform random binary fraction $U = (.b_1 b_2 \dots b_t)_2$; locate the first zero bit b_j , and shift off the leading j bits, setting $U \leftarrow (.b_{j+1} \dots b_t)_2$. (As in Algorithm F, the average value of j is 2.)
- S2. [Immediate acceptance?] If $U < \ln 2$, set $X \leftarrow \mu(j \ln 2 + U)$ and terminate the algorithm. (Note that $Q[1] = \ln 2$.)
- S3. [Minimize.] Find the least $k \geq 2$ such that $U < Q[k]$. Generate k new uniform deviates U_1, \dots, U_k and set $V \leftarrow \min(U_1, \dots, U_k)$.
- S4. [Deliver the answer.] Set $X \leftarrow \mu(j + V) \ln 2$. ■

Alternative ways to generate exponential deviates (e.g., a ratio of uniforms as in Algorithm R) might also be used.

E. Other continuous distributions. Let us now consider briefly how to handle some other distributions that arise reasonably often in practice.

(1) *The gamma distribution* of order $a > 0$ is defined by

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt, \quad x \geq 0. \quad (34)$$

When $a = 1$, this is the exponential distribution with mean 1; when $a = \frac{1}{2}$, it is the distribution of $\frac{1}{2}Z^2$, where Z has the normal distribution (mean 0, variance 1). If X and Y are independent gamma-distributed random variables, of order a and b , respectively, then $X + Y$ has the gamma distribution of order $a + b$. Thus, for example, the sum of k independent exponential deviates with mean 1 has the gamma distribution of order k . If the logarithm method (32) is being used to generate these exponential deviates, we need compute only one logarithm: $X \leftarrow -\ln(U_1 \dots U_k)$, where U_1, \dots, U_k are nonzero uniform deviates. This technique handles all integer orders a ; to complete the picture, a suitable method for $0 < a < 1$ appears in exercise 16.

The simple logarithm method is much too slow when a is large, since it requires $\lfloor a \rfloor$ uniform deviates. For large a , the following algorithm due to J. H. Ahrens is reasonably efficient, and it is easy to write in terms of standard subroutines.

Algorithm A (*Gamma distribution of order $a > 1$*).

- A1. [Generate candidate.] Set $Y \leftarrow \tan(\pi U)$, where U is a uniform deviate, and set $X \leftarrow \sqrt{2a-1}Y + a - 1$. (In place of $\tan(\pi U)$ we could use a polar method, e.g., V_2/V_1 in step P4 of Algorithm P.)
- A2. [Accept?] If $X \leq 0$, return to A1. Otherwise generate a uniform deviate V , and return to A1 if $V > (1+Y^2)\exp((a-1)\ln(X/(a-1)) - \sqrt{2a-1}Y)$. Otherwise accept X . ■

The average number of times step A1 is performed is < 1.902 when $a \geq 3$. For further discussion, proof, and a slightly more complex method that is two to three times faster, see J. H. Ahrens and U. Dieter, *Computing* 12 (1974), 223–246.

There is also an attractive approach for large a based on the remarkable fact that gamma deviates are approximately equal to aX^3 , where X is normally distributed with mean $1 - 1/(9a)$ and standard deviation $1/\sqrt{9a}$; see G. Marsaglia, *Computers and Math.* 3 (1977), 321–325.*

(2) *The beta distribution* with positive parameters a and b is defined by

$$F(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1}(1-t)^{b-1} dt, \quad 0 \leq x \leq 1. \quad (35)$$

Let X_1 and X_2 be independent gamma deviates of order a and b , respectively, and set $X \leftarrow X_1/(X_1 + X_2)$. Another method, useful for small a and b , is to set

*Change “ $+(3a-1)$ ” to “ $-(3a-1)$ ” in Step 3 of the algorithm on page 323.

$Y_1 \leftarrow U_1^{1/a}$ and $Y_2 \leftarrow U_2^{1/b}$ repeatedly until $Y_1 + Y_2 \leq 1$; then $X \leftarrow Y_1/(Y_1 + Y_2)$. [See M. D. Jöhnk, *Metrika* 8 (1964), 5–15.] Still another approach, if a and b are integers (not too large), is to set X to the b th largest of $a + b - 1$ independent uniform deviates (cf. exercise 7 at the beginning of Chapter 5). See also the direct method described by R. C. H. Cheng, *CACM* 21 (1978), 317–322.

(3) The chi-square distribution with ν degrees of freedom (Eq. 3.3.1–22) is obtained by setting $X \leftarrow 2Y$, where Y is a random variable having the gamma distribution of order $\nu/2$.

(4) The F -distribution (variance-ratio distribution) with ν_1 and ν_2 degrees of freedom is defined by

$$F(x) = \frac{\nu_1^{\nu_1/2} \nu_2^{\nu_2/2} \Gamma((\nu_1 + \nu_2)/2)}{\Gamma(\nu_1/2) \Gamma(\nu_2/2)} \int_0^x t^{\nu_1/2-1} (\nu_2 + \nu_1 t)^{-\nu_1/2-\nu_2/2} dt, \quad (36)$$

where $x \geq 0$. Let Y_1 and Y_2 be independent, having the chi-square distribution with ν_1 and ν_2 degrees of freedom, respectively; set $X \leftarrow Y_1 \nu_2 / Y_2 \nu_1$. Or set $X \leftarrow \nu_2 Y / \nu_1 (1 - Y)$, where Y is a beta variate with parameters $\nu_1/2$, $\nu_2/2$.

(5) The t -distribution with ν degrees of freedom is defined by

$$F(x) = \frac{\Gamma((\nu + 1)/2)}{\sqrt{\pi \nu} \Gamma(\nu/2)} \int_{-\infty}^x (1 + t^2/\nu)^{-(\nu+1)/2} dt. \quad (37)$$

Let Y_1 be a normal deviate (mean 0, variance 1) and let Y_2 be independent of Y_1 , having the chi-square distribution with ν degrees of freedom; set $X \leftarrow Y_1 / \sqrt{Y_2 / \nu}$. Alternatively, when $\nu > 2$, let Y_1 be a normal deviate and let Y_2 independently have the exponential distribution with mean $2/(\nu - 2)$; set $Z \leftarrow Y_1^2 / (\nu - 2)$ and reject (Y_1, Y_2) if $e^{-Y_2 - Z} \geq 1 - Z$, otherwise set $X \leftarrow Y_1 / \sqrt{(1 - 2\nu)(1 - z)}$. The latter method is due to George Marsaglia, *Math. Comp.* 34 (1980), 235–236. [See also A. J. Kinderman, J. F. Monahan, and J. G. Ramage, *Math. Comp.* 31 (1977), 1009–1018.]

(6) Random point on n -dimensional sphere with radius one. Let X_1, X_2, \dots, X_n be independent normal deviates (mean 0, variance 1); the desired point on the unit sphere is

$$(X_1/r, X_2/r, \dots, X_n/r), \quad \text{where } r = \sqrt{X_1^2 + X_2^2 + \dots + X_n^2}. \quad (38)$$

Note that if the X 's are calculated using the polar method, Algorithm P, we compute two independent X 's each time, and $X_1^2 + X_2^2 = -2 \ln S$ (in the notation of that algorithm); this saves a little of the time needed to evaluate r . The validity of this method comes from the fact that the distribution function for the point (X_1, \dots, X_n) has a density that depends only on its distance from the origin, so when it is projected onto the unit sphere it has the uniform distribution. This method was first suggested by G. W. Brown, in *Modern Mathematics for the Engineer*, First series, ed. by E. F. Beckenbach (New York: McGraw-Hill,

1956), p. 302. To get a random point *inside* the n -sphere, R. P. Brent suggests taking a point on the surface and multiplying it by $U^{1/n}$.

In three dimensions a significantly simpler method can be used, since each individual coordinate is uniformly distributed between -1 and 1 : Find V_1 , V_2 , and S by steps P1–P3 of Algorithm P; then the desired random point on the surface of a globe is $(\alpha V_1, \alpha V_2, 2S - 1)$, where $\alpha = 2\sqrt{1 - S}$. [Robert E. Knop, CACM 13 (1970), 326.]

F. Important integer-valued distributions. A probability distribution that consists only of integer values can essentially be handled by the techniques described at the beginning of this section; but some of these distributions are so important in practice, they deserve special mention here.

(1) *The geometric distribution.* If some event occurs with probability p , the number N of independent trials needed until the event first occurs (or between occurrences of the event) has the geometric distribution. We have $N = 1$ with probability p , $N = 2$ with probability $(1 - p)p$, ..., $N = n$ with probability $(1 - p)^{n-1}p$. This is essentially the situation we have already considered in the gap test of Section 3.3.2; it is also directly related to the number of times certain loops in the algorithms of this section are executed, e.g., steps P1–P3 of the polar method.

A convenient way to generate a variable with this distribution is to set

$$N \leftarrow \lceil \ln U / \ln(1 - p) \rceil. \quad (39)$$

To check this formula, we observe that $\lceil \ln U / \ln(1 - p) \rceil = n$ if and only if $n - 1 < \ln U / \ln(1 - p) \leq n$, that is, $(1 - p)^{n-1} > U \geq (1 - p)^n$, and this happens with probability $p(1 - p)^{n-1}$ as required. Note that $\ln U$ can be replaced by $-Y$, where Y has the exponential distribution with mean 1.

The special case $p = \frac{1}{2}$ can be handled more easily on a binary computer, since formula (39) becomes $N \leftarrow \lceil -\log_2 U \rceil$; that is, N is one more than the number of leading zero bits in the binary representation of U .

(2) *The binomial distribution* (t, p). If some event occurs with probability p , and if we carry out t independent trials, the total number N of occurrences equals n with probability $\binom{t}{n} p^n (1 - p)^{t-n}$. (See Section 1.2.10.) In other words if we generate U_1, \dots, U_t , we want to count how many of these are $< p$. For small t we can obtain N in exactly this way.

For large t , we can generate a beta variate X with integer parameters a and b where $a + b - 1 = t$; this effectively gives us the b th largest of t elements, without bothering to generate the other elements. Now if $X \geq p$, we set $N \leftarrow N_1$ where N_1 has the binomial distribution $(a - 1, p/X)$, since this tells us how many of $a - 1$ random numbers in the range $[0, X)$ are $< p$; and if $X < p$, we set $N \leftarrow a + N_1$ where N_1 has the binomial distribution $(b - 1, (p - X)/(1 - X))$, since N_1 tells us how many of $b - 1$ random numbers in the range $[X, 1)$ are $< p$. By choosing $a = 1 + \lfloor t/2 \rfloor$, the parameter t will be reduced to a reasonable size after about $\lg t$ reductions of this kind. (This approach is due to J. H. Ahrens, who has also suggested an alternative for medium-sized t ; see exercise 27.)

(3) *The Poisson distribution* with mean μ . This distribution is related to the exponential distribution as the binomial distribution is related to the geometric: It represents the number of occurrences, per unit time, of an event that can occur at any instant of time. For example, the number of alpha particles emitted by a radioactive substance in a single second has a Poisson distribution.

According to this principle, we can produce a Poisson deviate N by generating independent exponential deviates X_1, X_2, \dots with mean $1/\mu$, stopping as soon as $X_1 + \dots + X_m \geq 1$; then $N \leftarrow m - 1$. The probability that $X_1 + \dots + X_m \geq 1$ is the probability that a gamma deviate of order m is $\geq \mu$, and this comes to $\int_{\mu}^{\infty} t^{m-1} e^{-t} dt / (m-1)!$; hence the probability that $N = n$ is

$$\frac{1}{n!} \int_{\mu}^{\infty} t^n e^{-t} dt = \frac{1}{(n-1)!} \int_{\mu}^{\infty} t^{n-1} e^{-t} dt = e^{-\mu} \frac{\mu^n}{n!}, \quad n \geq 0. \quad (40)$$

If we generate exponential deviates by the logarithm method, the above recipe tells us to stop when $-(\ln U_1 + \dots + \ln U_m)/\mu \geq 1$. Simplifying this expression, we see that the desired Poisson deviate can be obtained by calculating $e^{-\mu}$, converting it to a fixed point representation, then generating one or more uniform deviates U_1, U_2, \dots until the product satisfies $U_1 \dots U_m \leq e^{-\mu}$, finally setting $N \leftarrow m - 1$. On the average this requires the generation of $\mu + 1$ uniform deviates, so it is a very useful approach when μ is not too large.

When μ is large, we can obtain a method of order $\log \mu$ by using the fact that we know how to handle the gamma and binomial distributions for large orders: First generate X with the gamma distribution of order $m = \lfloor \alpha \mu \rfloor$, where α is a suitable constant. (Since X is equivalent to $-\ln(U_1 \dots U_m)$, we are essentially bypassing m steps of the previous method.) If $X < \mu$, set $N \leftarrow m + N_1$, where N_1 is a Poisson deviate with mean $\mu - X$; and if $X \geq \mu$, set $N \leftarrow N_1$, where N_1 has the binomial distribution $(m-1, \mu/X)$. This method is due to J. H. Ahrens and U. Dieter, whose experiments suggest that $\frac{7}{8}$ is a good choice for α .

The validity of the above reduction when $X \geq \mu$ is a consequence of the following important principle: "Let X_1, \dots, X_m be independent exponential deviates with the same mean; let $S_j = X_1 + \dots + X_j$ and let $V_j = S_j/S_m$ for $1 \leq j \leq m$. Then the distribution of V_1, V_2, \dots, V_{m-1} is the same as the distribution of $m-1$ independent uniform deviates sorted into increasing order." To establish this principle formally, we compute the probability that $V_1 \leq v_1, \dots, V_{m-1} \leq v_{m-1}$, given the value of $S_m = s$, for arbitrary values $0 \leq v_1 \leq \dots \leq v_{m-1} \leq 1$: Let $f(v_1, v_2, \dots, v_{m-1})$ be the $(m-1)$ -fold integral

$$\int_0^{v_1 s} \mu e^{-t_1/\mu} dt_1 \int_0^{v_2 s - t_1} \mu e^{-t_2/\mu} dt_2 \dots \\ \times \int_0^{v_{m-1} s - t_1 - \dots - t_{m-2}} \mu e^{-t_{m-1}/\mu} dt_{m-1} \cdot \mu e^{-(s-t_1-\dots-t_{m-1})/\mu};$$

then

$$\frac{f(v_1, v_2, \dots, v_{m-1})}{f(1, 1, \dots, 1)} = \frac{\int_0^{v_1} du_1 \int_{u_1}^{v_2} du_2 \dots \int_{u_{m-2}}^{v_{m-1}} du_{m-1}}{\int_0^1 du_1 \int_{u_1}^1 du_2 \dots \int_{u_{m-2}}^1 du_{m-1}},$$

by making the substitution $t_1 = su_1$, $t_1 + t_2 = su_2$, ..., $t_1 + \dots + t_{m-1} = su_{m-1}$. The latter ratio is the corresponding probability that uniform deviates U_1, \dots, U_{m-1} satisfy $U_1 \leq v_1, \dots, U_{m-1} \leq v_{m-1}$, given that they also satisfy $U_1 \leq \dots \leq U_{m-1}$.

A more efficient but somewhat more complicated technique for binomial and Poisson deviates is sketched in exercise 22.

G. For further reading. The forthcoming book *Non-Uniform Random Numbers* by J. H. Ahrens and U. Dieter discusses many more algorithms for the generation of random variables with nonuniform distributions, together with a careful consideration of the efficiency of each technique on typical computers.

From a theoretical point of view it is interesting to consider *optimal* methods for generating random variables with a given distribution, in the sense that the method produces the desired result from the minimum possible number of random bits. For the beginnings of a theory dealing with such questions, see D. E. Knuth and A. C. Yao, *Algorithms and Complexity*, ed. by J. F. Traub (New York: Academic Press, 1976), 357–428.

Exercise 16 is recommended as a review of many of the techniques in this section.

EXERCISES

1. [10] If α and β are real numbers with $\alpha < \beta$, how would you generate a random real number uniformly distributed between α and β ?
2. [M16] Assuming that mU is a random integer between 0 and $m - 1$, what is the exact probability that $\lfloor kU \rfloor = r$, if $0 \leq r < k$? Compare this with the desired probability $1/k$.
3. [14] Discuss treating U as an integer and dividing by k to get a random integer between 0 and $k - 1$, instead of multiplying as suggested in the text. Thus (1) would be changed to

```
ENTA 0
LDX  U
DIV  K
```

with the result appearing in register X. Is this a good method?

4. [M20] Prove the two relations in (8).
5. [21] Suggest an efficient method to compute a random variable with the distribution $px + qx^2 + rx^3$, where $p \geq 0$, $q \geq 0$, $r \geq 0$, and $p + q + r = 1$.
6. [HM21] A quantity X is computed by the following method:

“Step 1. Generate two independent uniform deviates U, V .

Step 2. If $U^2 + V^2 \geq 1$, return to step 1; otherwise set $X \leftarrow U$.”

What is the distribution function of X ? How many times will step 1 be performed? (Give the mean and standard deviation.)

- 7. [20] (A. J. Walker.) Suppose we have a bunch of cubes of k different colors, say n_j cubes of color C_j for $1 \leq j \leq k$, and we also have k boxes $\{B_1, \dots, B_k\}$ each of which can hold exactly n cubes. Furthermore $n_1 + \dots + n_k = kn$, so the cubes will just fit in the boxes. Prove (constructively) that there is always a way to put the cubes into the boxes so that each box contains at most two different colors of cubes; in fact, there is a way to do it so that, whenever box B_j contains two colors, one of those colors is C_j . Show how to use this principle to compute the P and Y tables required in (3), given a probability distribution (p_1, \dots, p_k) .

8. [M15] Show that operation (3) could be changed to

$$\text{if } U < P_K \text{ then } X \leftarrow x_{K+1} \text{ otherwise } X \leftarrow Y_K$$

(i.e., using the original value of U instead of V) if this were more convenient, by suitably modifying P_0, P_1, \dots, P_{k-1} .

9. [HM10] Why is the curve $f(x)$ of Fig. 9 concave downward for $x < 1$, concave upward for $x > 1$?
- 10. [HM24] Explain how to calculate auxiliary constants $P_j, Q_j, Y_j, Z_j, S_j, D_j, E_j$ so that Algorithm M delivers answers with the correct distribution.
- 11. [HM27] Prove that steps M7–M8 of Algorithm M generate a random variable with the appropriate tail of the normal distribution; i.e., the probability that $X \leq x$ should be

$$\int_3^x e^{-t^2/2} dt / \int_3^\infty e^{-t^2/2} dt, \quad x \geq 3.$$

[Hint: Show that it is a special case of the rejection method, with $g(t) = Cte^{-t^2/2}$ for some C .]

12. [HM23] (R. P. Brent.) Prove that the numbers a_j defined in (23) satisfy the relation $a_j^2/2 - a_{j-1}^2/2 < \ln 2$ for all $j \geq 1$. [Hint: If $f(x) = e^{x^2/2} \int_x^\infty e^{-t^2/2} dt$, show that $f(x) < f(y)$ for $0 \leq x < y$.]

13. [HM25] Given a set of n independent normal deviates, X_1, X_2, \dots, X_n , with mean 0 and variance 1, show how to find constants b_j and a_{ij} , $1 \leq j \leq i \leq n$, so that if

$$\begin{aligned} Y_1 &= b_1 + a_{11}X_1, & Y_2 &= b_2 + a_{21}X_1 + a_{22}X_2, & \dots, \\ && Y_n &= b_n + a_{n1}X_1 + a_{n2}X_2 + \dots + a_{nn}X_n, \end{aligned}$$

then Y_1, Y_2, \dots, Y_n are dependent normally distributed variables, Y_j has mean μ_j , and the Y 's have a given covariance matrix (c_{ij}) . (The covariance, c_{ij} , of Y_i and Y_j is defined to be the average value of $(Y_i - \mu_i)(Y_j - \mu_j)$. In particular, c_{jj} is the variance of Y_j , the square of its standard deviation. Not all matrices (c_{ij}) can be covariance matrices, and your construction is, of course, only supposed to work whenever a solution to the given conditions is possible.)

14. [M21] If X is a random variable with continuous distribution $F(x)$, and if c is a constant, what is the distribution of cX ?

15. [HM21] If X_1 and X_2 are independent random variables with the respective distributions $F_1(x)$ and $F_2(x)$, and with densities $f_1(x) = F_1'(x)$, $f_2(x) = F_2'(x)$, what are the distribution and density functions of the quantity $X_1 + X_2$?

► 16. [HM22] (J. H. Ahrens.) Develop an algorithm for gamma deviates of order a when $0 < a \leq 1$, using the rejection method with $c g(t) = t^{a-1}/\Gamma(a)$ for $0 < t < 1$, $e^{-t}/\Gamma(a)$ for $t \geq 1$.

► 17. [M24] What is the distribution function $F(x)$ for the geometric distribution with probability p ? What is the generating function $G(z)$? What are the mean and standard deviation of this distribution?

18. [M24] Suggest a method to compute a random integer N for which N takes the value n with probability $np^2(1-p)^{n-1}$, $n \geq 0$. (The case of particular interest is when p is rather small.)

19. [22] The negative binomial distribution (t, p) has integer values $N = n$ with probability $\binom{t-1+n}{n} p^t (1-p)^n$. (Unlike the ordinary binomial distribution, t need not be an integer, since this quantity is nonnegative for all n whenever $t > 0$.) Generalizing exercise 18, explain how to generate integers N with this distribution when t is a small positive integer. What method would you suggest if $t = p = \frac{1}{2}$?

20. [M20] Let A be the area of the shaded region in Fig. 13, and let R be the area of the enclosing rectangle. Let I be the area of the interior region recognized by step R2, and let E be the area between the exterior region rejected in step R3 and the outer rectangle. Determine the number of times each step of Algorithm R is performed, for each of its four variants as in (25), in terms of A , R , I , and E .

21. [HM29] Derive formulas for the quantities A , R , I , and E defined in exercise 20. (For I and especially E you may wish to use an interactive computer algebra system.) Show that $c = e^{1/4}$ is the best possible constant in step R2 for tests of the form " $X^2 \leq 4(1 + \ln c) - 4cU$ ".

22. [HM40] Can the exact Poisson distribution for large μ be obtained by generating an appropriate normal deviate, converting it to an integer in some convenient way, and applying a (possibly complicated) correction a small percent of the time?

23. [HM29] (J. von Neumann.) Are the following two ways to generate a random quantity X equivalent (i.e., does the quantity X have the same distribution)?

Method 1: Set $X \leftarrow \sin((\pi/2)U)$, where U is uniform.

Method 2: Generate two uniform deviates, U , V , and if $U^2 + V^2 \geq 1$, repeat until $U^2 + V^2 < 1$. Then set $X \leftarrow |U^2 - V^2|/(U^2 + V^2)$.

24. [HM40] (S. Ulam, J. von Neumann.) Let V_0 be a randomly selected real number between 0 and 1, and define the sequence $\langle V_n \rangle$ by the rule $V_{n+1} = 4V_n(1 - V_n)$. If this computation is done with perfect accuracy, the result should be a sequence with the distribution $\sin^2 \pi U$, where U is uniform, i.e., with distribution function $F(x) = \int_0^x dx / \sqrt{2\pi x(1-x)}$. For if we write $V_n = \sin^2 \pi U_n$, we find that $U_{n+1} = (2U_n) \bmod 1$; and by the fact that almost all real numbers have a random binary expansion (see Section 3.5), this sequence U_n is equidistributed. But if the computation of V_n is done with only finite accuracy, the above argument breaks down because we soon are dealing with noise from the roundoff error. [Reference: von Neumann's *Collected Works* 5, 768–770.]

Analyze the sequence $\langle V_n \rangle$ defined above when only finite accuracy is present, both empirically (for various different choices of V_0) and theoretically. Does the sequence have a distribution resembling the expected distribution?

25. [M25] Let X_1, X_2, \dots, X_5 be binary words each of whose bits is independently 0 or 1 with probability $\frac{1}{2}$. What is the probability that a given bit position of $X_1 \vee (X_2 \wedge (X_3 \vee (X_4 \wedge X_5)))$ contains a 1? Generalize.

26. [M18] Let N_1 and N_2 be independent Poisson deviates with respective means μ_1 and μ_2 , where $\mu_1 > \mu_2 \geq 0$. Prove or disprove: (a) $N_1 + N_2$ has the Poisson distribution with mean $\mu_1 + \mu_2$. (b) $N_1 - N_2$ has the Poisson distribution with mean $\mu_1 - \mu_2$.

27. [22] (J. H. Ahrens.) On most binary computers there is an efficient way to count the number of 1's in a binary word (cf. Section 7.1). Hence there is a nice way to obtain the binomial distribution (t, p) when $p = \frac{1}{2}$, simply by generating t random bits and counting the number of 1's.

Design an algorithm that produces the binomial distribution (t, p) for arbitrary p , using only a subroutine for the special case $p = \frac{1}{2}$ as a source of random data. [Hint: Simulate a process that first looks at the most significant bits of t uniform deviates, then at the second bit of those deviates whose leading bit is not sufficient to determine whether or not their value is $< p$, etc.]

28. [HM35] (R. P. Brent.) Develop a method to generate a random point on the surface of the ellipsoid defined by $\sum a_k x_k^2 = 1$, where $a_1 \geq \dots \geq a_n > 0$.

29. [M20] (J. L. Bentley and J. D. Saxe.) Find a simple way to generate n numbers X_1, \dots, X_n that are uniform between 0 and 1 except for the fact that they are sorted: $X_1 \leq \dots \leq X_n$. Your algorithm should take only $O(n)$ steps.

3.4.2. Random Sampling and Shuffling

Many data processing applications call for an unbiased choice of n records at random from a file containing N records. This problem arises, for example, in quality control or other statistical calculations where sampling is needed. Usually N is very large, so that it is impossible to contain all the data in memory at once; and the individual records themselves are often very large, so that we can't even hold n records in memory. Therefore we seek an efficient procedure for selecting n records by deciding either to accept or to reject each record as it comes along, writing the accepted records onto an output file.

Several methods have been devised for this problem. The most obvious approach is to select each record with probability n/N ; this may sometimes be appropriate, but it gives only an average of n records in the sample. The standard deviation is $\sqrt{n(1 - n/N)}$, and it is possible that the sample will be either too large for the desired application, or too small to give the necessary results.

A simple modification of the "obvious" procedure gives what we want: The $(t+1)$ st record should be selected with probability $(n-m)/(N-t)$, if m items have already been selected. This is the appropriate probability, since of all the possible ways to choose n things from N such that m values occur in the first t , exactly

$$\binom{N-t-1}{n-m-1} / \binom{N-t}{n-m} = \frac{n-m}{N-t} \quad (1)$$

of these select the $(t + 1)$ st element.

The idea developed in the preceding paragraph leads immediately to the following algorithm:

Algorithm S (*Selection sampling technique*). To select n records at random from a set of N , where $0 < n \leq N$.

- S1. [Initialize.] Set $t \leftarrow 0$, $m \leftarrow 0$. (During this algorithm, m represents the number of records selected so far, and t is the total number of input records we have dealt with.)
- S2. [Generate U .] Generate a random number U , uniformly distributed between zero and one.
- S3. [Test.] If $(N - t)U \geq n - m$, go to step S5.
- S4. [Select.] Select the next record for the sample, and increase m and t by 1. If $m < n$, go to step S2; otherwise the sample is complete and the algorithm terminates.
- S5. [Skip.] Skip the next record (do not include it in the sample), increase t by 1, and go to step S2. ■

This algorithm may appear to be unreliable at first glance and, in fact, to be incorrect; but a careful analysis (see the exercises below) shows that it is completely trustworthy. It is not difficult to verify that

- a) At most N records are input (we never run off the end of the file before choosing n items).
- b) The sample is completely unbiased; in particular, the probability that any given element is selected, e.g., the last element of the file, is n/N .

Statement (b) is true in spite of the fact that we are *not* selecting the $(t+1)$ st item with probability n/N ; we select it with the probability in Eq. (1)! This has caused some confusion in the published literature. Can the reader explain this seeming contradiction?

(Note: When using Algorithm S, one should be careful to use a different source of random numbers U each time the program is run, to avoid connections between the samples obtained on different days. This can be done, for example, by choosing a different value of X_0 for the linear congruential method each time; X_0 could be set to the current date, or to the last X value generated on the previous run of the program.)

We will usually not have to pass over all N records; in fact, since (b) above says that the last record is selected with probability n/N , we will terminate the algorithm *before* considering the last record exactly $(1 - n/N)$ of the time. The average number of records considered when $n = 2$ is about $\frac{2}{3}N$, and the general formulas are given in exercises 5 and 6.

Algorithm S and a number of other sampling techniques are discussed in a paper by C. T. Fan, Mervin E. Muller, and Ivan Rezucha, *J. Amer. Stat. Assoc.* 57 (1962), 387–402. The method was independently discovered by T. G. Jones, *CACM* 5 (1962), 343.

A problem arises if we don't know the value of N in advance, since the precise value of N is crucial in Algorithm S. Suppose we want to select n items at random from a file, without knowing exactly how many are present in that file. We could first go through and count the records, then take a second pass to select them; but it is generally better to sample $m \geq n$ of the original items on the first pass, where m is much less than N , so that only m items must be considered on the second pass. The trick, of course, is to do this in such a way that the final result is a truly random sample of the original file.

Since we don't know when the input is going to end, we must keep track of a random sample of the input records seen so far, thus always being prepared for the end. As we read the input we will construct a "reservoir" that contains only those m records that have appeared among the previous samples. The first n records always go into the reservoir. When the $(t+1)$ st record is being input, for $t \geq n$, we will have in memory a table of n indices pointing to those records in the reservoir that belong to the random sample we have chosen from the first t records. The problem is to maintain this situation with t increased by one, namely to find a new random sample from among the $t+1$ records now known to be present. It is not hard to see that we should include the new record in the new sample with probability $n/(t+1)$, and in such a case it should replace a random element of the previous sample.

Thus, the following procedure does the job:

Algorithm R (*Reservoir sampling*). To select n records at random from a file of unknown size $\geq n$, given $n > 0$. An auxiliary file called the "reservoir" contains all records that are candidates for the final sample. The algorithm uses a table of distinct indices $I[j]$ for $1 \leq j \leq n$, each of which points to one of the records in the reservoir.

- R1. [Initialize.] Input the first n records and copy them to the reservoir. Set $I[j] \leftarrow j$ for $1 \leq j \leq n$, and set $t \leftarrow m \leftarrow n$. (If the file being sampled has fewer than n records, it will of course be necessary to abort the algorithm and report failure. During this algorithm, $\{I[1], \dots, I[n]\}$ point to the records in the current sample, m is the size of the reservoir, and t is the number of input records dealt with so far.)
- R2. [End of file?] If there are no more records to be input, go to step R6.
- R3. [Generate and test.] Increase t by 1, then generate a random integer M between 1 and t (inclusive). If $M > n$, go to R5.
- R4. [Add to reservoir.] Copy the next record of the input file to the reservoir, increase m by 1, and set $I[M] \leftarrow m$. (The record previously pointed to by $I[M]$ is being replaced in the sample by the new record.) Go back to R2.
- R5. [Skip.] Skip over the next record of the input file (do not include it in the reservoir), and return to step R2.
- R6. [Second pass.] Sort the I table entries so that $I[1] < \dots < I[n]$; then go through the reservoir, copying the records with these indices into the output file that is to hold the final sample. ■

Algorithm R is due to Alan G. Waterman. The reader may wish to work out the example of its operation that appears in exercise 9.

If the records are sufficiently short, it is of course unnecessary to have a reservoir at all; we can keep the n records of the current sample in memory at all times (see exercise 10).

The natural question to ask about Algorithm R is, "What is the expected size of the reservoir?" Exercise 11 shows that the average value of m is exactly $n(1 + H_N - H_n)$; this is approximately $n(1 + \ln(N/n))$. So if $N/n = 1000$, the reservoir will contain only about $\frac{1}{125}$ as many items as the original file.

Note that Algorithms S and R can be used to obtain samples for several independent categories simultaneously. For example, if we have a large file of names and addresses of U.S. residents, we could pick random samples of exactly 10 people from each of the 50 states without making 50 passes through the file, and without first sorting the file by state.

The sampling problem can be regarded as the computation of a random *combination*, according to the conventional definition of combinations of N things taken n at a time (see Section 1.2.6). Now let us consider the problem of computing a random *permutation* of t objects; we will call this the *shuffling problem*, since shuffling a deck of cards is nothing more than subjecting it to a random permutation.

A moment's reflection is enough to convince oneself that the approaches people traditionally use to shuffle cards are miserably inadequate; there is no hope of obtaining each of the $t!$ permutations with anywhere near equal probability by such methods. It has been said that expert bridge players make use of this fact when deciding whether or not to "finesse."

If t is small, we can obtain a random permutation very quickly by generating a random integer between 1 and $t!$. For example, when $t = 4$, a random number between 1 and 24 suffices to select a random permutation from a table of all possibilities. But for large t , it is necessary to be more careful if we want to claim that each permutation is equally likely, since $t!$ is much larger than the accuracy of individual random numbers.

A suitable shuffling procedure can be obtained by recalling Algorithm 3.3.2P, which gives a simple correspondence between each of the $t!$ possible permutations and a sequence of numbers $(c_1, c_2, \dots, c_{t-1})$, with $0 \leq c_j \leq j$. It is easy to compute such a set of numbers at random, and we can use the correspondence to produce a random permutation.

Algorithm P (Shuffling). Let X_1, X_2, \dots, X_t be a set of t numbers to be shuffled.

P1. [Initialize.] Set $j \leftarrow t$.

P2. [Generate U .] Generate a random number U , uniformly distributed between zero and one.

P3. [Exchange.] Set $k \leftarrow \lfloor jU \rfloor + 1$. (Now k is a random integer, between 1 and j .) Exchange $X_k \leftrightarrow X_j$.

P4. [Decrease j .] Decrease j by 1. If $j > 1$, return to step P2. ■

This algorithm was first published by L. E. Moses and R. V. Oakford, in *Tables of Random Permutations* (Stanford University Press, 1963); and by R. Durstenfeld, *CACM* 7 (1964), 420. It can also be modified to obtain a random permutation of a random combination (see exercise 14).

For a discussion of random combinatorial objects of other kinds (e.g., partitions), see Section 7.2 and/or the book *Combinatorial Algorithms* by Nijenhuis and Wilf (New York: Academic Press, 1975).

EXERCISES

1. [M12] Explain Eq. (1).

2. [20] Prove that Algorithm S never tries to read more than N records of its input file.

► 3. [22] The $(t+1)$ st item in Algorithm S is selected with probability $(n-m)/(N-t)$, not n/N , yet the text claims that the sample is unbiased—so each item should be selected with the same probability. How can both of these statements be true?

4. [M23] Let $p(m, t)$ be the probability that exactly m items are selected from among the first t in the selection sampling technique. Show directly from Algorithm S that

$$p(m, t) = \binom{t}{m} \binom{N-t}{n-m} / \binom{N}{n}, \quad \text{for } 0 \leq t \leq N.$$

5. [M24] What is the average value of t when Algorithm S terminates? (In other words, how many of the N records have been passed, on the average, before the sample is complete?)

6. [M24] What is the standard deviation of the value computed in the previous exercise?

► 7. [M25] Prove that any given choice of n records from the set of N is obtained by Algorithm S with probability $1/\binom{N}{n}$. Therefore the sample is completely unbiased.

8. [M46] Algorithm S computes one uniform deviate for each input record it handles. Find a more efficient way to determine the number of input records to skip before the first is selected, assuming that N/n is rather large. (We could iterate this process to select the remaining $n - 1$ records, thus reducing the number of necessary random deviates from order N to order n .)

9. [12] Let $n = 3$. If Algorithm R is applied to a file containing 20 records numbered 1 thru 20, and if the random numbers generated in step R3 are respectively

$$4, 1, 6, 7, 5, 3, 5, 11, 11, 3, 7, 9, 3, 11, 4, 5, 4,$$

which records go into the reservoir? Which are in the final sample?

10. [15] Modify Algorithm R so that the reservoir is eliminated, assuming that the n records of the current sample can be held in memory.

► 11. [M25] Let p_m be the probability that exactly m elements are put into the reservoir during the first pass of Algorithm R. Determine the generating function $G(z) = \sum_m p_m z^m$, and find the mean and standard deviation. (Use the ideas of Section 1.2.10.)

12. [M26] The gist of Algorithm P is that any permutation π can be uniquely written as a product of transpositions in the form $\pi = (a_1t)\dots(a_33)(a_22)$, where $1 \leq a_j \leq j$ for $t \geq j > 1$. Prove that there is also a unique representation of the form $\pi = (b_22)(b_33)\dots(b_tt)$, where $1 \leq b_j \leq j$ for $1 < j \leq t$, and design an algorithm that computes the b 's from the a 's in $O(t)$ steps.

13. [M23] (S. W. Golomb.) One of the most common ways to shuffle cards is to divide the deck into two parts as equal as possible, and to “riffle” them together. (See the discussion of card-playing etiquette in Hoyle’s rules of card games; we read, “A shuffle of this sort should be made about three times to mix the cards thoroughly.”) Consider a deck of $2n - 1$ cards $X_1, X_2, \dots, X_{2n-1}$; a “perfect shuffle” s divides this deck into X_1, X_2, \dots, X_n and X_{n+1}, \dots, X_{2n-1} , then perfectly interleaves them to obtain $X_1, X_{n+1}, X_2, X_{n+2}, \dots, X_{2n-1}, X_n$. The “cut” operation c^j changes $X_1, X_2, \dots, X_{2n-1}$ into $X_{j+1}, \dots, X_{2n-1}, X_1, \dots, X_j$. Show that by combining perfect shuffles and cuts, at most $(2n - 1)(2n - 2)$ different arrangements of the deck are possible, if $n > 1$.

► **14.** [30] (Ole-Johan Dahl.) If $X_k = k$ for $1 \leq k \leq t$ at the start of Algorithm P, and if we terminate the algorithm when j reaches the value $t - n$, the sequence X_{t-n+1}, \dots, X_t is a random permutation of a random combination of n elements. Show how to simulate the effect of this procedure using only $O(n)$ cells of memory.

► **15.** [M25] Devise a way to compute a random sample of n records from N , given N and n , based on the idea of hashing (Section 6.4). Your method should use $O(n)$ storage locations and an average of $O(n)$ units of time, and it should present the sample as a sorted set of integers $1 \leq X_1 < X_2 < \dots < X_n \leq N$.

16. [25] Discuss the problem of weighted sampling, where each subset of n elements is obtained with probability proportional to the product of the weights of the elements.

3.5. WHAT IS A RANDOM SEQUENCE?

A. Introductory remarks. We have seen in this chapter how to generate sequences

$$\langle U_n \rangle = U_0, U_1, U_2, \dots \quad (1)$$

of real numbers in the range $0 \leq U_n < 1$, and we have called them “random” sequences even though they are completely deterministic in character. To justify this terminology, we claimed that the numbers “behave as if they are truly random.” Such a statement may be satisfactory for practical purposes (at the present time), but it sidesteps a very important philosophical and theoretical question: Precisely what do we mean by “random behavior”? A quantitative definition is needed. It is undesirable to talk about concepts that we do not really understand, especially since many apparently paradoxical statements can be made about random numbers.

The mathematical theory of probability and statistics carefully sidesteps the question; it refrains from making absolute statements, and instead expresses everything in terms of how much *probability* is to be attached to statements involving random sequences of events. The axioms of probability theory are set up so that abstract probabilities can be computed readily, but nothing is said about what probability really signifies, or how this concept can be applied meaningfully to the actual world. In the book *Probability, Statistics, and Truth* (New York: Macmillan, 1957), R. von Mises discusses this situation in detail, and presents the view that a proper definition of probability depends on obtaining a proper definition of a random sequence.

Let us paraphrase here some statements made by two of the many authors who have commented on the subject.

D. H. Lehmer (1951): “A random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests, traditional with statisticians and depending somewhat on the uses to which the sequence is to be put.”

J. N. Franklin (1962): “The sequence (1) is random if it has every property that is shared by all infinite sequences of independent samples of random variables from the uniform distribution.”

Franklin’s statement essentially generalizes Lehmer’s to say that the sequence must satisfy *all* statistical tests. His definition is not completely precise, and we will see later that a reasonable interpretation of his statement leads us to conclude that there is no such thing as a random sequence! So let us begin with Lehmer’s less restrictive statement and attempt to make it precise. What we really want is a relatively short list of mathematical properties, each of which is satisfied by our intuitive notion of a random sequence; furthermore, the list is to be complete enough so that we are willing to agree that any sequence satisfying these properties is “random.” In this section, we will develop what seems to be

an adequate definition of randomness according to these criteria, although many interesting questions remain to be answered.

Let u and v be real numbers, $0 \leq u < v \leq 1$. If U is a random variable that is uniformly distributed between 0 and 1, the probability that $u \leq U < v$ is equal to $v - u$. For example, the probability that $\frac{1}{3} \leq U < \frac{2}{3}$ is $\frac{1}{3}$. How can we translate this property of the single number U into a property of the infinite sequence U_0, U_1, U_2, \dots ? The obvious answer is to count how many times U_n lies between u and v , and the average number of times should equal $v - u$. Our intuitive idea of probability is based in this way on the frequency of occurrence.

More precisely, let $\nu(n)$ be the number of values of j , $0 \leq j < n$, such that $u \leq U_j < v$; we want the ratio $\nu(n)/n$ to approach the value $v - u$ as n approaches infinity:

$$\lim_{n \rightarrow \infty} \nu(n)/n = v - u. \quad (2)$$

If this condition holds for all choices of u and v , the sequence is said to be *equidistributed*.

Let $S(n)$ be a statement about the integer n and the sequence U_1, U_2, \dots ; for example, $S(n)$ might be the statement considered above, " $u \leq U_n < v$." We can generalize the idea used in the preceding paragraph to define "the probability that $S(n)$ is true" with respect to a particular infinite sequence: Let $\nu(n)$ be the number of values of j , $0 \leq j < n$, such that $S(j)$ is true.

Definition A. We say $\Pr(S(n)) = \lambda$, if $\lim_{n \rightarrow \infty} \nu(n)/n = \lambda$. (Read, "The probability that $S(n)$ is true equals λ , if the limit as n tends to infinity of $\nu(n)/n$ equals λ .)

In terms of this notation, the sequence U_0, U_1, \dots is equidistributed if and only if $\Pr(u \leq U_n < v) = v - u$, for all real numbers u, v with $0 \leq u < v \leq 1$.

A sequence may be equidistributed without being random. For example, if U_0, U_1, \dots and V_0, V_1, \dots are equidistributed sequences, it is not hard to show that the sequence

$$W_0, W_1, W_2, W_3, \dots = \frac{1}{2}U_0, \frac{1}{2} + \frac{1}{2}V_0, \frac{1}{2}U_1, \frac{1}{2} + \frac{1}{2}V_1, \dots \quad (3)$$

is also equidistributed, since the subsequence $\frac{1}{2}U_0, \frac{1}{2}U_1, \dots$ is equidistributed between 0 and $\frac{1}{2}$, while the alternate terms $\frac{1}{2} + \frac{1}{2}V_0, \frac{1}{2} + \frac{1}{2}V_1, \dots$, are equidistributed between $\frac{1}{2}$ and 1. In the sequence of W 's, a value less than $\frac{1}{2}$ is always followed by a value greater than or equal to $\frac{1}{2}$, and conversely; hence the sequence is not random by any reasonable definition. A stronger property than equidistribution is needed.

A natural generalization of the equidistribution property, which removes the objection stated in the preceding paragraph, is to consider adjacent pairs of numbers of our sequence. We can require the sequence to satisfy the condition

$$\Pr(u_1 \leq U_n < v_1 \text{ and } u_2 \leq U_{n+1} < v_2) = (v_1 - u_1)(v_2 - u_2) \quad (4)$$

for any four numbers u_1, v_1, u_2, v_2 with $0 \leq u_1 < v_1 \leq 1, 0 \leq u_2 < v_2 \leq 1$. In

general, for any positive integer k we can require our sequence to be k -distributed in the following sense:

Definition B. *The sequence (1) is said to be k -distributed if*

$$\Pr(u_1 \leq U_n < v_1, \dots, u_k \leq U_{n+k-1} < v_k) = (v_1 - u_1) \dots (v_k - u_k) \quad (5)$$

for all choices of real numbers u_j, v_j , with $0 \leq u_j < v_j \leq 1$, for $1 \leq j \leq k$.

An equidistributed sequence is a 1-distributed sequence. Note that if $k > 1$, a k -distributed sequence is always $(k-1)$ -distributed, since we may set $u_k = 0$ and $v_k = 1$ in Eq. (5). Thus, in particular, any sequence that is known to be 4-distributed must also be 3-distributed, 2-distributed, and equidistributed. We can investigate the largest k for which a given sequence is k -distributed; and this leads us to formulate

Definition C. *A sequence is said to be ∞ -distributed if it is k -distributed for all positive integers k .*

So far we have considered “[0, 1) sequences,” i.e., sequences of real numbers lying between zero and one. The same ideas apply to integer-valued sequences; let us say a sequence $\langle X_n \rangle = X_0, X_1, X_2, \dots$ is a “ b -ary sequence” if each X_n is one of the integers $0, 1, \dots, b-1$. Thus, a 2-ary (binary) sequence is a sequence of zeros and ones.

We also say that a k -digit “ b -ary number” is a string of k integers $x_1 x_2 \dots x_k$, where $0 \leq x_j < b$ for $1 \leq j \leq k$.

Definition D. *A b -ary sequence is said to be k -distributed if*

$$\Pr(X_n X_{n+1} \dots X_{n+k-1} = x_1 x_2 \dots x_k) = 1/b^k \quad (6)$$

for all b -ary numbers $x_1 x_2 \dots x_k$.

It is clear from this definition that if U_0, U_1, \dots is a k -distributed [0, 1) sequence, then the sequence $[bU_0], [bU_1], \dots$ is a k -distributed b -ary sequence. (If we set $u_j = x_j/b$, $v_j = (x_j + 1)/b$, $X_n = [bU_n]$, Eq. (5) becomes Eq. (6).) Furthermore, every k -distributed b -ary sequence is also $(k-1)$ -distributed, if $k > 1$: we add together the probabilities for the b -ary numbers $x_1 \dots x_{k-1} 0$, $x_1 \dots x_{k-1} 1, \dots, x_1 \dots x_{k-1} (b-1)$ to obtain

$$\Pr(X_n \dots X_{n+k-2} = x_1 \dots x_{k-1}) = 1/b^{k-1}.$$

(Probabilities for disjoint events are additive; see exercise 5.) It therefore is natural to speak of an ∞ -distributed b -ary sequence, as in Definition C above.

The representation of a positive real number in the radix- b number system may be regarded as a b -ary sequence; for example, π corresponds to the 10-ary

sequence 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, It has been conjectured that this sequence is ∞ -distributed, but nobody has yet been able to prove that it is even 1-distributed.

Let us analyze these concepts a little more closely in the case when k equals a million. A binary sequence that is 1000000-distributed is going to have runs of a million zeros in a row! Similarly, a $[0, 1]$ sequence that is 1000000-distributed is going to have runs of a million consecutive values each of which is less than $\frac{1}{2}$. It is true that this will happen only $(\frac{1}{2})^{1000000}$ of the time, on the average, but the fact is that it does happen. Indeed, this phenomenon will occur in any truly random sequence, using our intuitive notion of "truly random." One can easily imagine that such a situation will have a drastic effect if this set of a million "truly random" numbers is being used in a computer-simulation experiment; there would be good reason to complain about the random number generator. However, if we have a sequence of numbers that never has runs of a million consecutive U 's less than $\frac{1}{2}$, the sequence is not random, and it will not be a suitable source of numbers for other conceivable applications that use extremely long blocks of U 's as input. In summary, a *truly random sequence will exhibit local nonrandomness*. Local nonrandomness is necessary in some applications, but it is disastrous in others. We are forced to conclude that no sequence of "random" numbers can be adequate for every application.

In a similar vein, one may argue that there is no way to judge whether a finite sequence is random or not; any particular sequence is just as likely as any other one. These facts are definitely stumbling blocks if we are ever to have a useful definition of randomness, but they are not really cause for alarm. It is still possible to give a definition for the randomness of infinite sequences of real numbers in such a way that the corresponding theory (viewed properly) will give us a great deal of insight concerning the ordinary finite sequences of rational numbers that are actually generated on a computer. Furthermore, we shall see later in this section that there are several plausible definitions of randomness for finite sequences.

B. ∞ -distributed sequences. Let us now undertake a brief study of the theory of sequences that are ∞ -distributed. To describe the theory adequately, we will need to use a bit of higher mathematics, so we assume in the remainder of this subsection that the reader knows the material ordinarily taught in an "advanced calculus" course.

First it is convenient to generalize Definition A, since the limit appearing there does not exist for all sequences. Let us define

$$\overline{\Pr}(S(n)) = \limsup_{n \rightarrow \infty} (\nu(n)/n), \quad \underline{\Pr}(S(n)) = \liminf_{n \rightarrow \infty} (\nu(n)/n). \quad (7)$$

Then $\Pr(S(n))$, if it exists, is the common value of $\underline{\Pr}(S(n))$ and $\overline{\Pr}(S(n))$.

We have seen that a k -distributed $[0, 1]$ sequence leads to a k -distributed b -ary sequence, if U is replaced by $[bU]$. Our first theorem shows that a converse result is also true.

Theorem A. Let $\langle U_n \rangle = U_0, U_1, U_2, \dots$ be a $[0, 1)$ sequence. If the sequence

$$\langle \lfloor b_j U_n \rfloor \rangle = \lfloor b_j U_0 \rfloor, \lfloor b_j U_1 \rfloor, \lfloor b_j U_2 \rfloor, \dots$$

is a k -distributed b_j -ary sequence for all b_j in an infinite sequence of integers $1 < b_1 < b_2 < b_3 < \dots$, then the original sequence $\langle U_n \rangle$ is k -distributed.

As an example of this theorem, suppose that $b_j = 2^j$. The sequence $\lfloor 2^j U_0 \rfloor, \lfloor 2^j U_1 \rfloor, \dots$ is essentially the sequence of the first j bits of the binary representations of U_0, U_1, \dots . If all these integer sequences are k -distributed, in the sense of Definition D, then the real-valued sequence U_0, U_1, \dots must also be k -distributed in the sense of Definition B.

Proof of Theorem A. If the sequence $\lfloor b U_0 \rfloor, \lfloor b U_1 \rfloor, \dots$ is k -distributed, it follows by the addition of probabilities that Eq. (5) holds whenever each u_j and v_j is a rational number with denominator b . Now let u_j, v_j be any real numbers, and let u'_j, v'_j be rational numbers with denominator b such that

$$u'_j \leq u_j < u'_j + 1/b, \quad v'_j \leq v_j < v'_j + 1/b.$$

Let $S(n)$ be the statement that $u_1 \leq U_n < v_1, \dots, u_k \leq U_{n+k-1} < v_k$. We have

$$\begin{aligned} \overline{\Pr}(S(n)) &\leq \Pr\left(u'_1 \leq U_n < v'_1 + \frac{1}{b}, \dots, u'_k \leq U_{n+k-1} < v'_k + \frac{1}{b}\right) \\ &= \left(v'_1 - u'_1 + \frac{1}{b}\right) \dots \left(v'_k - u'_k + \frac{1}{b}\right); \end{aligned}$$

$$\begin{aligned} \underline{\Pr}(S(n)) &\geq \Pr\left(u'_1 + \frac{1}{b} \leq U_n < v'_1, \dots, u'_k + \frac{1}{b} \leq U_{n+k-1} < v'_k\right) \\ &= \left(v'_1 - u'_1 - \frac{1}{b}\right) \dots \left(v'_k - u'_k - \frac{1}{b}\right). \end{aligned}$$

Now $|(v'_j - u'_j \pm 1/b) - (v_j - u_j)| \leq 2/b$; since our inequalities hold for all $b = b_j$, and since $b_j \rightarrow \infty$ as $j \rightarrow \infty$, we have

$$(v_1 - u_1) \dots (v_k - u_k) \leq \underline{\Pr}(S(n)) \leq \overline{\Pr}(S(n)) \leq (v_1 - u_1) \dots (v_k - u_k). \blacksquare$$

The next theorem is our main tool for proving things about k -distributed sequences.

Theorem B. Let $\langle U_n \rangle$ be a k -distributed $[0, 1)$ sequence, and let $f(x_1, x_2, \dots, x_k)$ be a Riemann-integrable function of k variables; then

$$\begin{aligned} &\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{0 \leq j < n} f(U_j, U_{j+1}, \dots, U_{j+k-1}) \\ &= \int_0^1 \dots \int_0^1 f(x_1, x_2, \dots, x_k) dx_1 \dots dx_k. \end{aligned} \tag{8}$$

Proof. The definition of a k -distributed sequence states that this result is true in the special case that

$$f(x_1, \dots, x_k) = \begin{cases} 1, & \text{if } u_1 \leq x_1 < v_1, \dots, u_k \leq x_k < v_k; \\ 0, & \text{otherwise.} \end{cases} \quad (9)$$

Therefore Eq. (8) is true whenever $f = a_1 f_1 + a_2 f_2 + \dots + a_m f_m$ and when each f_j is a function of type (9); in other words, Eq. (8) holds whenever f is a “step-function” obtained by (i) partitioning the unit k -dimensional cube into subcells whose faces are parallel to the coordinate axes, and (ii) assigning a constant value to f on each subcell.

Now let f be any Riemann-integrable function. If ϵ is any positive number, we know (by the definition of Riemann-integrability) that there exist step functions \underline{f} and \bar{f} such that $\underline{f}(x_1, \dots, x_k) \leq f(x_1, \dots, x_k) \leq \bar{f}(x_1, \dots, x_k)$, and such that the difference of the integrals of \underline{f} and \bar{f} is less than ϵ . Since Eq. (8) holds for \underline{f} and \bar{f} , and since

$$\begin{aligned} \frac{1}{n} \sum_{0 \leq j < n} \underline{f}(U_j, \dots, U_{j+k-1}) &\leq \frac{1}{n} \sum_{0 \leq j < n} f(U_j, \dots, U_{j+k-1}) \\ &\leq \frac{1}{n} \sum_{0 \leq j < n} \bar{f}(U_j, \dots, U_{j+k-1}), \end{aligned}$$

we conclude that Eq. (8) is true also for f . ■

Theorem B can be applied, for example, to the permutation test of Section 3.3.2. Let (p_1, p_2, \dots, p_k) be any permutation of the numbers $\{1, 2, \dots, k\}$; we want to show that

$$\Pr(U_{n+p_1-1} < U_{n+p_2-1} < \dots < U_{n+p_k-1}) = 1/k!. \quad (10)$$

To prove this, assume that the sequence $\langle U_n \rangle$ is k -distributed, and let

$$f(x_1, \dots, x_k) = \begin{cases} 1, & \text{if } x_{p_1} < x_{p_2} < \dots < x_{p_k}; \\ 0, & \text{otherwise.} \end{cases}$$

We have

$$\begin{aligned} \Pr(U_{n+p_1-1} < U_{n+p_2-1} < \dots < U_{n+p_k-1}) \\ &= \int_0^1 \cdots \int_0^1 f(x_1, \dots, x_k) dx_1 \dots dx_k \\ &= \int_0^1 dx_{p_k} \int_0^{x_{p_k}} \cdots \int_0^{x_{p_3}} dx_{p_2} \int_0^{x_{p_2}} dx_{p_1} = \frac{1}{k!}. \end{aligned}$$

Corollary P. If a $[0, 1]$ sequence is k -distributed, it satisfies the permutation test of order k , in the sense of Eq. (10). ■

We can also show that the *serial correlation test* is satisfied:

Corollary 8. If a $[0, 1]$ sequence is $(k + 1)$ -distributed, the serial correlation coefficient between U_n and U_{n+k} tends to zero:

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{n} \sum U_j U_{j+k} - (\frac{1}{n} \sum U_j)(\frac{1}{n} \sum U_{j+k})}{\sqrt{(\frac{1}{n} \sum U_j^2 - (\frac{1}{n} \sum U_j)^2)(\frac{1}{n} \sum U_{j+k}^2 - (\frac{1}{n} \sum U_{j+k})^2)}} = 0.$$

(All summations here are for $0 \leq j < n$.)

Proof. By Theorem B, the quantities

$$\frac{1}{n} \sum U_j U_{j+k}, \quad \frac{1}{n} \sum U_j^2, \quad \frac{1}{n} \sum U_{j+k}^2, \quad \frac{1}{n} \sum U_j, \quad \frac{1}{n} \sum U_{j+k}$$

tend to the respective limits $\frac{1}{4}, \frac{1}{3}, \frac{1}{3}, \frac{1}{2}, \frac{1}{2}$ as $n \rightarrow \infty$. ■

Let us now consider some slightly more general distribution properties of sequences. We have defined the notion of k -distribution by considering all of the adjacent k -tuples; for example, a sequence is 2-distributed if and only if the points

$$(U_0, U_1), (U_1, U_2), (U_2, U_3), (U_3, U_4), (U_4, U_5), \dots$$

are equidistributed in the unit square. It is quite possible, however, that this can happen while alternate pairs of points $(U_1, U_2), (U_3, U_4), (U_5, U_6), \dots$ are not equidistributed; if the density of points (U_{2n-1}, U_{2n}) is deficient in some area, the other points (U_{2n}, U_{2n+1}) might compensate. For example, the periodic binary sequence

$$\langle X_n \rangle = 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, \dots, \quad (11)$$

with a period of length 16, is seen to be 3-distributed; yet the sequence of even-numbered elements $\langle X_{2n} \rangle = 0, 0, 0, 1, 0, 1, 0, 1, 0, \dots$ has three times as many zeros as ones, while the subsequence of odd-numbered elements $\langle X_{2n+1} \rangle = 0, 1, 0, 1, 1, 1, 1, 1, \dots$ has three times as many ones as zeros.

If a sequence $\langle U_n \rangle$ is ∞ -distributed, example (11) shows that it is not at all obvious that the subsequence of alternate terms $\langle U_{2n} \rangle = U_0, U_2, U_4, U_6, \dots$ will be ∞ -distributed or even 1-distributed. But we shall see that $\langle U_{2n} \rangle$ is, in fact, ∞ -distributed, and much more is true.

Definition E. A $[0, 1]$ sequence $\langle U_n \rangle$ is said to be (m, k) -distributed if

$$\begin{aligned} \Pr(u_1 \leq U_{mn+j} < v_1, u_2 \leq U_{mn+j+1} < v_2, \dots, u_k \leq U_{mn+j+k-1} < v_k) \\ &= (v_1 - u_1) \dots (v_k - u_k) \end{aligned}$$

for all choices of real numbers u_r, v_r with $0 \leq u_r < v_r \leq 1$ for $1 \leq r \leq k$, and for all integers j with $0 \leq j < m$.

Thus a k -distributed sequence is the special case $m = 1$ in Definition E; the case $m = 2$ means that the k -tuples starting in even positions must have the same density as the k -tuples starting in odd positions, etc.

Several properties of Definition E are obvious:

An (m, k) -distributed sequence is (m, κ) -distributed for $1 \leq \kappa \leq k$. (12)

An (m, k) -distributed sequence is (d, k) -distributed for all divisors d of m . (13)

We can also define the concept of an (m, k) -distributed b -ary sequence, as in Definition D; and the proof of Theorem A remains valid for (m, k) -distributed sequences.

The next theorem, which is in many ways rather surprising, shows that the property of being ∞ -distributed is very strong indeed, much stronger than we imagined it to be when we first considered the definition of the concept.

Theorem C (Ivan Niven and H. S. Zuckerman). *An ∞ -distributed sequence is (m, k) -distributed for all positive integers m and k .*

Proof. It suffices to prove the theorem for b -ary sequences, by using the generalization of Theorem A just mentioned. Furthermore, we may assume that $m = k$, because (12) and (13) tell us that the sequence will be (m, k) -distributed if it is (mk, mk) -distributed.

So we will prove that *any ∞ -distributed b -ary sequence X_0, X_1, \dots is (m, m) -distributed for all positive integers m .* Our proof is a simplified version of the original one given by Niven and Zuckerman in *Pacific J. Math.* 1 (1951), 103–109.

The key idea we shall use is an important technique that applies to many situations in mathematics: “If the sum of m quantities and the sum of their squares are both consistent with the hypothesis that the m quantities are equal, then that hypothesis is true.” In a strong form, this principle may be stated as follows:

Lemma E. *Given m sequences of numbers $\langle y_{jn} \rangle = y_{j0}, y_{j1}, \dots$ for $1 \leq j \leq m$, suppose that*

$$\begin{aligned} \lim_{n \rightarrow \infty} (y_{1n} + y_{2n} + \dots + y_{mn}) &= m\alpha, \\ \limsup_{n \rightarrow \infty} (y_{1n}^2 + y_{2n}^2 + \dots + y_{mn}^2) &\leq m\alpha^2. \end{aligned} \quad (14)$$

Then for each j , $\lim_{n \rightarrow \infty} y_{jn}$ exists and equals α .

An incredibly simple proof of this lemma is given in exercise 9. ■

Resuming our proof of Theorem C, let $x = x_1x_2\dots x_m$ be a b -ary number, and say that x occurs at position p if $X_{p-m+1}X_{p-m+2}\dots X_p = x$. Let $\nu_j(n)$ be the number of occurrences of x at position p when $p < n$ and $p \bmod m = j$. Let $y_{jn} = \nu_j(n)/n$; we wish to prove that

$$\lim_{n \rightarrow \infty} y_{jn} = 1/b^m. \quad (15)$$

First we know that

$$\lim_{n \rightarrow \infty} (y_{0n} + y_{1n} + \cdots + y_{(m-1)n}) = 1/b^m, \quad (16)$$

since the sequence is m -distributed. By Lemma E and Eq. (16), the theorem will be proved if we can show that

$$\limsup_{n \rightarrow \infty} (y_{0n}^2 + y_{1n}^2 + \cdots + y_{(m-1)n}^2) \leq 1/m b^{2m}. \quad (17)$$

This inequality is not obvious yet; some rather delicate maneuvering is necessary before we can prove it. Let q be a multiple of m , and consider

$$C(n) = \sum_{0 \leq j < m} \binom{\nu_j(n) - \nu_j(n-q)}{2}. \quad (18)$$

This is the number of pairs of occurrences of x in positions p_1, p_2 with $n-q \leq p_1 < p_2 < n$ and with $p_2 - p_1$ a multiple of m . Consider now the sum

$$S_N = \sum_{1 \leq n \leq N+q} C(n). \quad (19)$$

Each pair of occurrences of x in positions p_1, p_2 with $p_1 < p_2 < p_1 + q$, where $p_2 - p_1$ is a multiple of m and $p_1 \leq N$, is counted $p_1 + q - p_2$ times in the total S_N (namely, when $p_2 < n \leq p_1 + q$); and the pairs of such occurrences with $N < p_1 < p_2 < N + q$ are counted $N + q - p_2$ times.

Let $d_t(n)$ be the number of pairs of occurrences of x in positions p_1, p_2 with $p_1 + t = p_2 < n$. The analysis above shows that

$$\sum_{0 < t < q/m} (q - mt)d_{mt}(N+q) \geq S_N \geq \sum_{0 < t < q/m} (q - mt)d_{mt}(N). \quad (20)$$

Since the original sequence is q -distributed,

$$\lim_{N \rightarrow \infty} \frac{1}{N} d_{mt}(N) = 1/b^{2m} \quad (21)$$

for all t , $0 < t < q/m$, and therefore by (20) we have

$$\lim_{N \rightarrow \infty} \frac{S_N}{N} = \sum_{0 < t < q/m} (q - mt)/b^{2m} = q(q - m)/2mb^{2m}. \quad (22)$$

This fact will prove the theorem, after some manipulation.

By definition,

$$2S_N = \sum_{1 \leq n \leq N+q} \sum_{0 \leq j < m} ((\nu_j(n) - \nu_j(n-q))^2 - (\nu_j(n) - \nu_j(n-q))),$$

and we can remove the unsquared terms by applying (16) to get

$$\lim_{N \rightarrow \infty} \frac{T_N}{N} = q(q-m)/mb^{2m} + q/b^m, \quad (23)$$

where

$$T_N = \sum_{1 \leq n \leq N+q} \sum_{0 \leq j < m} (\nu_j(n) - \nu_j(n-q))^2.$$

Using the inequality

$$\frac{1}{r} \left(\sum_{1 \leq j \leq r} a_j \right)^2 \leq \sum_{1 \leq j \leq r} a_j^2$$

(cf. exercise 1.2.3–30), we find that

$$\begin{aligned} \limsup_{N \rightarrow \infty} \sum_{0 \leq j < m} \frac{1}{N(N+q)} \left(\sum_{1 \leq n \leq N+q} (\nu_j(n) - \nu_j(n-q)) \right)^2 \\ \leq q(q-m)/mb^{2m} + q/b^m. \end{aligned} \quad (24)$$

We also have

$$q \nu_j(N) \leq \sum_{N < n \leq N+q} \nu_j(n) = \sum_{1 \leq n \leq N+q} (\nu_j(n) - \nu_j(n-q)) \leq q \nu_j(N+q),$$

and putting this into (24) gives

$$\limsup_{N \rightarrow \infty} \sum_{0 \leq j < m} (\nu_j(N)/N)^2 \leq (q-m)/qmb^{2m} + 1/qb^m. \quad (25)$$

This formula has been established whenever q is a multiple of m ; and if we let $q \rightarrow \infty$ we obtain (17), completing the proof.

For a possibly simpler proof, see J. W. S. Cassels, *Pacific J. Math.* 2 (1952), 555–557. ■

Exercises 29 and 30 illustrate the nontriviality of this theorem, and they also demonstrate the fact that a q -distributed sequence will have probabilities deviating from the true (m, m) -distribution probabilities by essentially $1/\sqrt{q}$ at most. (Cf. (25).) The full hypothesis of ∞ -distribution is necessary for the proof of the theorem.

As a result of Theorem C, we can prove that an ∞ -distributed sequence passes the serial test, the maximum-of- t test, the collision test, and the tests on subsequences mentioned in Section 3.3.2; it is not hard to show that the gap test, the poker test, and the run test are also satisfied (see exercises 12 through 14). The coupon collector's test is considerably more difficult to deal with, but it too is satisfied (see exercises 15 and 16).

The existence of ∞ -distributed sequences of a rather simple type is guaranteed by the next theorem.

Theorem F (J. Franklin). *The $[0, 1]$ sequence U_0, U_1, \dots , with*

$$U_n = \theta^n \bmod 1 \quad (26)$$

is ∞ -distributed for almost all real numbers $\theta > 1$. That is, the set

$$\{ \theta \mid \theta > 1 \text{ and (26) is not } \infty\text{-distributed} \}$$

is of measure zero.

The proofs of this theorem and some generalizations are given in Franklin's paper cited below. ■

Franklin has shown that θ must be a transcendental number for (26) to be ∞ -distributed. The powers $(\pi^n \bmod 1)$ have been laboriously computed for $n \leq 10000$, using multiple-precision arithmetic, and the most significant 35 bits of each of these numbers, stored on a disk file, have successfully been used as a source of uniform deviates. According to Theorem F, the probability that the powers $(\pi^n \bmod 1)$ are ∞ -distributed is equal to 1; yet because there are uncountably many real numbers, this gives us no information as to whether the sequence is really ∞ -distributed or not. It is a fairly safe bet that nobody in our lifetimes will ever prove that this particular sequence is not ∞ -distributed; but it might not be. Because of these considerations, one may legitimately wonder if there is any explicit sequence that is ∞ -distributed; i.e., is there an algorithm to compute real numbers U_n for all $n \geq 0$, such that the sequence $\langle U_n \rangle$ is ∞ -distributed? The answer is yes, as shown for example by D. E. Knuth in *BIT* 5 (1965), 246–250. The sequence constructed there consists entirely of rational numbers; in fact, each number U_n has a terminating representation in the binary number system. Another construction of an explicit ∞ -distributed sequence, somewhat more complicated than the sequence just cited, follows from Theorem W below. See also N. M. Korobov, *Izv. Akad. Nauk SSSR* 20 (1956), 649–660.

C. Does ∞ -distributed = random? In view of all the above theory about ∞ -distributed sequences, we can be sure of one thing: the concept of an ∞ -distributed sequence is an important one in mathematics. There is also a good deal of evidence that the following statement is a valid formulation of the intuitive idea of randomness:

Definition R1. A $[0, 1]$ sequence is defined to be “random” if it is an ∞ -distributed sequence.

We have seen that sequences meeting this definition will satisfy all the statistical tests of Section 3.3.2 and many more.

Let us attempt to criticize this definition objectively. First of all, is every “truly random” sequence ∞ -distributed? There are uncountably many sequences U_0, U_1, \dots of real numbers between zero and one. If a truly random number generator is sampled to give values U_0, U_1, \dots , any of the possible sequences may be considered equally likely, and some of the sequences (indeed, uncountably many of them) are not even equidistributed. On the other hand, using any reasonable definition of probability on this space of all possible sequences leads us to conclude that a random sequence is ∞ -distributed *with probability one*. We are therefore led to formalize Franklin’s definition of randomness (as given at the beginning of this section) in the following way:

Definition R2. A $[0, 1]$ sequence $\langle U_n \rangle$ is defined to be “random” if, whenever P is a property such that $P(\langle V_n \rangle)$ holds with probability one for a sequence $\langle V_n \rangle$ of independent samples of random variables from the uniform distribution, then $P(\langle U_n \rangle)$ is true.

Is it perhaps possible that Definition R1 is equivalent to Definition R2? Let us try out some possible objections to Definition R1, and see whether these criticisms are valid.

In the first place, Definition R1 deals only with limiting properties of the sequence as $n \rightarrow \infty$. There are ∞ -distributed sequences in which the first million elements are all zero; should such a sequence be considered random?

This objection is not very valid. If ϵ is any positive number, there is no reason why the first million elements of a sequence should not all be less than ϵ . With probability one, a truly random sequence contains infinitely many runs of a million consecutive elements less than ϵ , so why can’t this happen at the beginning of the sequence?

On the other hand, consider Definition R2 and let P be the property that all elements of the sequence are distinct; P is true with probability one, so any sequence with a million zeros is not random by *this* criterion.

Now let P be the property that no element of the sequence is equal to zero; again, P is true with probability one, so by Definition R2 any sequence with a zero element is nonrandom. More generally, however, let x_0 be any fixed number between zero and one, and let P be the property that no element of the sequence is equal to x_0 ; Definition R2 now says that no random sequence may contain the element x_0 ! We can now prove that *no sequence satisfies the condition of Definition R2*. (For if U_0, U_1, \dots is such a sequence, take $x_0 = U_0$.)

Therefore if R1 is too weak a definition, R2 is certainly too strong. The “right” definition must be less strict than R2. We have not really shown that R1 is too weak, however, so let us continue to attack it some more. As mentioned above, an ∞ -distributed sequence of *rational* numbers has been constructed. (Indeed, this is not so surprising; see exercise 18.) Almost all real numbers are irrational; perhaps we should insist that

$$\Pr(U_n \text{ is rational}) = 0$$

for a random sequence.

Note that the definition of equidistribution says that $\Pr(u \leq U_n < v) = v - u$. There is an obvious way to generalize this definition, using measure theory: "If $S \subseteq [0, 1]$ is a set of measure μ , then

$$\Pr(U_n \in S) = \mu, \quad (27)$$

for all random sequences $\langle U_n \rangle$." In particular, if S is the set of rationals, it has measure zero, so no sequence of rational numbers is equidistributed in this generalized sense. It is reasonable to expect that Theorem B could be extended to Lebesgue integration instead of Riemann integration, if property (27) is stipulated. However, once again we find that definition (27) is too strict, for no sequence satisfies that property. If U_0, U_1, \dots is any sequence, the set $S = \{U_0, U_1, \dots\}$ is of measure zero, yet $\Pr(U_n \in S) = 1$. Thus, by the force of the same argument we used to exclude rationals from random sequences, we can exclude all random sequences.

So far Definition R1 has proved to be defensible. There are, however, some quite valid objections to it. For example, if we have a random sequence in the intuitive sense, the infinite subsequence

$$U_0, U_1, U_4, U_9, \dots, U_{n^2}, \dots \quad (28)$$

should also be a random sequence. This is not always true for an ∞ -distributed sequence. In fact, if we take any ∞ -distributed sequence and set $U_{n^2} \leftarrow 0$ for all n , the counts $\nu_k(n)$ that appear in the test of k -distributivity are changed by at most \sqrt{n} , so the limits of the ratios $\nu_k(n)/n$ remain unchanged. Definition R1 unfortunately fails to satisfy this randomness criterion.

Perhaps we should strengthen R1 as follows:

Definition R3. A $[0, 1]$ sequence is said to be "random" if each of its infinite subsequences is ∞ -distributed.

Once again, however, the definition turns out to be too strict; any equidistributed sequence $\langle U_n \rangle$ has a monotonic subsequence with $U_{s_0} < U_{s_1} < U_{s_2} < \dots$.

The secret is to restrict the subsequences so that they could be defined by somebody who does not look at U_n before deciding whether or not it is to be in the subsequence. The following definition now suggests itself:

Definition R4. A $[0, 1]$ sequence $\langle U_n \rangle$ is said to be "random" if, for every effective algorithm that specifies an infinite sequence of distinct nonnegative integers s_n for $n \geq 0$, the subsequence $U_{s_0}, U_{s_1}, U_{s_2}, \dots$ corresponding to this algorithm is ∞ -distributed.

The algorithms referred to in Definition R4 are effective procedures that compute s_n , given n . (See Section 1.1.) Thus, for example, the sequence $\langle \pi^n \bmod 1 \rangle$ will not satisfy R4, since it is either not equidistributed or there is an effective algorithm that determines an infinite subsequence s_n with $(\pi^{s_0} \bmod 1) < (\pi^{s_1} \bmod 1) < (\pi^{s_2} \bmod 1) < \dots$. Similarly, no explicitly defined sequence can satisfy Definition R4; this is appropriate, if we agree that no explicitly defined sequence can really be random. It is quite likely, however, that the sequence $\langle \theta^n \bmod 1 \rangle$ will satisfy Definition R4, for almost all real numbers $\theta > 1$; this is no

contradiction, since almost all θ are uncomputable by algorithms. The following facts are known, for example: (i) The sequence $\langle \theta^n \bmod 1 \rangle$ satisfies Definition R4 for almost all real $\theta > 1$, if “ ∞ -distributed” is replaced by “1-distributed.” This theorem was proved by J. F. Koksma, *Compositio Mathematica* 2 (1935), 250–258. (ii) The particular sequence $\langle \theta^{s(n)} \bmod 1 \rangle$ is ∞ -distributed for almost all real $\theta > 1$, if $\langle s(n) \rangle$ is a sequence of integers for which $s(n+1) - s(n) \rightarrow \infty$ as $n \rightarrow \infty$. For example, we could have $s(n) = n^2$, or $s(n) = \lfloor n \lg n \rfloor$.

Definition R4 is much stronger than Definition R1; but it is still reasonable to claim that Definition R4 is too weak. For example, let $\langle U_n \rangle$ be a truly random sequence, and define the subsequence $\langle U_{s_n} \rangle$ by the following rules: $s_0 = 0$, and (for $n > 0$) s_n is the smallest integer $\geq n$ for which $U_{s_n-1}, U_{s_n-2}, \dots, U_{s_n-n}$ are all less than $\frac{1}{2}$. Thus we are considering the subsequence of values following the first consecutive run of n values less than $\frac{1}{2}$. Suppose that “ $U_n < \frac{1}{2}$ ” corresponds to the value “heads” in the flipping of a coin. Gamblers tend to feel that a long run of “heads” makes the opposite condition, “tails,” more probable, assuming that a true coin is being used; and the subsequence $\langle U_{s_n} \rangle$ just defined corresponds to a gambling system for a man who places his n th bet on the coin toss following the first run of n consecutive “heads.” The gambler may think that $\Pr(U_{s_n} \geq \frac{1}{2})$ is more than $\frac{1}{2}$, but of course in a truly random sequence $\langle U_{s_n} \rangle$ will be completely random. No gambling system will ever be able to beat the odds! Definition R4 says nothing about subsequences formed according to such a gambling system, so apparently we need something more.

Let us define a “subsequence rule” \mathcal{R} as an infinite sequence of functions $\langle f_n(x_1, \dots, x_n) \rangle$ where, for $n \geq 0$, f_n is a function of n variables, and the value of $f_n(x_1, \dots, x_n)$ is either 0 or 1. Here x_1, \dots, x_n are elements of some set S . (Thus, in particular, f_0 is a constant function, either 0 or 1.) A subsequence rule \mathcal{R} defines a subsequence of any infinite sequence $\langle X_n \rangle$ of elements of S as follows: *The nth term X_n is in the subsequence $\langle X_n \rangle \mathcal{R}$ if and only if $f_n(X_0, X_1, \dots, X_{n-1}) = 1$.* Note that the subsequence $\langle X_n \rangle \mathcal{R}$ thus defined is not necessarily infinite, and it may in fact contain no elements at all.

For example, the gambler’s subsequence just described corresponds to the following subsequence rule: “ $f_0 = 1$; and for $n > 0$, $f_n(x_1, \dots, x_n) = 1$ if and only if there is some k in the range $0 < k \leq n$ such that the k consecutive parameters $x_m, x_{m-1}, \dots, x_{m-k+1}$ are all $< \frac{1}{2}$ when $m = n$ but not when $k \leq m < n$.”

A subsequence rule \mathcal{R} is said to be *computable* if there is an effective algorithm that determines the value of $f_n(x_1, \dots, x_n)$, when n and x_1, \dots, x_n are given as input. We had better restrict ourselves to computable subsequence rules when trying to define randomness, lest we obtain an overly restrictive definition like R3 above. But effective algorithms cannot deal nicely with arbitrary real numbers as inputs; for example, if a real number x is specified by an infinite radix-10 expansion, there is no algorithm to determine if x is $< \frac{1}{3}$ or not, since all digits of the number 0.333... have to be examined. Therefore computable subsequence rules do not apply to all $[0, 1)$ sequences, and it is convenient to base our next definition on b -ary sequences.

Definition R5. A b -ary sequence is said to be “random” if every infinite subsequence defined by a computable subsequence rule is 1-distributed.

A $[0, 1)$ sequence $\langle U_n \rangle$ is said to be “random” if the b -ary sequence $\langle [bU_n] \rangle$ is “random” for all integers $b \geq 2$.

Note that Definition R5 says only “1-distributed,” not “ ∞ -distributed.” It is interesting to verify that this may be done without loss of generality. For we may define an obviously computable subsequence rule $\mathcal{R}(a_1 \dots a_k)$ as follows, given any b -ary number $a_1 \dots a_k$: Let $f_n(x_1, \dots, x_n) = 1$ if and only if $n \geq k-1$ and $x_{n-k+1} = a_1, \dots, x_{n-1} = a_{k-1}, x_n = a_k$. Now if $\langle X_n \rangle$ is a k -distributed b -ary sequence, this rule $\mathcal{R}(a_1 \dots a_k)$ —which selects the subsequence consisting of those terms just following an occurrence of $a_1 \dots a_k$ —defines an infinite subsequence; and if this subsequence is 1-distributed, each of the $(k+1)$ -tuples $a_1 \dots a_k a_{k+1}$ for $0 \leq a_{k+1} < b$ occurs with probability $1/b^{k+1}$ in $\langle X_n \rangle$. Thus we can prove that a sequence satisfying Definition R5 is k -distributed for all k , by induction on k . Similarly, by considering the “composition” of subsequence rules—if \mathcal{R}_1 defines an infinite subsequence $\langle X_n \rangle \mathcal{R}_1$, then we can define $\mathcal{R}_1 \mathcal{R}_2$ to be the subsequence rule for which $\langle X_n \rangle \mathcal{R}_1 \mathcal{R}_2 = (\langle X_n \rangle \mathcal{R}_1) \mathcal{R}_2$ —we find that all subsequences considered in Definition R5 are ∞ -distributed. (See exercise 32.)

The fact that ∞ -distribution comes out of Definition R5 as a very special case is encouraging, and it is a good indication that we may at last have found the definition of randomness we have been seeking. But alas, there still is a problem. It is not clear that sequences satisfying Definition R5 must satisfy Definition R4. The “computable subsequence rules” we have just specified always enumerate subsequences $\langle X_{s_n} \rangle$ for which $s_0 < s_1 < \dots$, but $\langle s_n \rangle$ does not have to be monotone in R4; it must only satisfy the condition $s_n \neq s_m$ for $n \neq m$.

To meet this objection, we may combine Definitions R4 and R5 as follows:

Definition R6. A b -ary sequence $\langle X_n \rangle$ is said to be “random” if, for every effective algorithm that specifies an infinite sequence of distinct nonnegative integers $\langle s_n \rangle$ as a function of n and the values of $X_{s_0}, \dots, X_{s_{n-1}}$, the subsequence $\langle X_{s_n} \rangle$ corresponding to this algorithm is “random” in the sense of Definition R5.

A $[0, 1)$ sequence $\langle U_n \rangle$ is said to be “random” if the b -ary sequence $\langle [bU_n] \rangle$ is “random” for all integers $b \geq 2$.

The author contends* that this definition surely meets all reasonable philosophical requirements for randomness, so it provides an answer to the principal question posed in this section.

D. Existence of random sequences. We have seen that Definition R3 is too strong, in the sense that no sequence can satisfy that definition; and the formulation of Definitions R4, R5, and R6 above was carried out in an attempt to recapture the essential characteristics of Definition R3. In order to show that Definition R6 is not overly restrictive, it is still necessary for us to prove that sequences satisfying all these conditions exist. Intuitively, we feel quite sure that there is no problem,

*At least, he made such a contention when originally preparing the material for this section in 1966.

because we believe that a truly random sequence exists and satisfies R6; but a proof is really necessary to show that the definition is consistent.

An interesting method for constructing sequences satisfying Definition R5 has been found by A. Wald, starting with a very simple 1-distributed sequence.

Lemma T. *Let the sequence of real numbers $\langle V_n \rangle$ be defined in terms of the binary system as follows:*

$$\begin{aligned} V_0 &= 0, & V_1 &= .1, & V_2 &= .01, & V_3 &= .11, & V_4 &= .001, & \dots \\ V_n &= .c_r \dots c_1 1 & \text{if } n = 2^r + c_1 2^{r-1} + \dots + c_r. \end{aligned} \quad (29)$$

Let $I_{b_1 \dots b_r}$ denote the set of all real numbers in $[0, 1)$ whose binary representation begins with $0.b_1 \dots b_r$; thus

$$I_{b_1 \dots b_r} = [(0.b_1 \dots b_r)_2, (0.b_1 \dots b_r)_2 + 2^{-r}). \quad (30)$$

Then if $\nu(n)$ denotes the number of V_k in $I_{b_1 \dots b_r}$ for $0 \leq k < n$, we have

$$|\nu(n)/n - 2^{-r}| \leq 1/n. \quad (31)$$

Proof. Since $\nu(n)$ is the number of k for which $k \bmod 2^r = (b_r \dots b_1)_2$, we have $\nu(n) = t$ or $t + 1$ when $\lfloor n/2^r \rfloor = t$. Hence $|\nu(n) - n/2^r| \leq 1$. ■

It follows from (31) that the sequence $\langle \lfloor 2^r V_n \rfloor \rangle$ is an equidistributed 2^r -ary sequence; hence by Theorem A, $\langle V_n \rangle$ is an equidistributed $[0, 1)$ sequence. Indeed, it is pretty clear that $\langle V_n \rangle$ is about as equidistributed as a $[0, 1)$ sequence can be. (For further discussion of this and related sequences, see J. G. van der Corput, *Proc. Koninklijke Nederl. Akad. Wetenschappen* 38 (1935), 813–821, 1058–1066; J. H. Halton, *Numerische Math.* 2 (1960), 84–90, 196; L. H. Ramshaw, *J. Number Theory*, to appear.)

Now let $\mathcal{R}_1, \mathcal{R}_2, \dots$ be infinitely many subsequence rules; we seek a sequence $\langle U_n \rangle$ for which all the infinite subsequences $\langle U_n \rangle_{\mathcal{R}_j}$ are equidistributed.

Algorithm W (Wald sequence). Given an infinite sequence of subsequence rules $\mathcal{R}_1, \mathcal{R}_2, \dots$ that define subsequences of $[0, 1)$ sequences of rational numbers, this procedure defines a $[0, 1)$ sequence $\langle U_n \rangle$. The computation involves infinitely many auxiliary variables $C[a_1, \dots, a_r]$, where $r \geq 1$ and where $a_j = 0$ or 1 for $1 \leq j \leq r$. These variables are initially all zero.

W1. [Initialize n .] Set $n \leftarrow 0$.

W2. [Initialize r .] Set $r \leftarrow 1$.

W3. [Test \mathcal{R}_r .] If the element U_n is to be in the subsequence defined by \mathcal{R}_r , based on the values of U_k for $0 \leq k < n$, set $a_r \leftarrow 1$; otherwise set $a_r \leftarrow 0$.

W4. [$B[a_1, \dots, a_r]$ full?] If $C[a_1, \dots, a_r] < 3 \cdot 4^{r-1}$, go to W6.

W5. [Increase r .] Set $r \leftarrow r + 1$ and return to W3.

W6. [Set U_n .] Increase the value of $C[a_1, \dots, a_r]$ by 1 and let k be its new value.

Set $U_n \leftarrow V_k$, where V_k is defined in Lemma T above.

W7. [Advance n .] Increase n by 1 and return to W2. ■

Strictly speaking, this is not an algorithm, since it doesn't terminate; it would of course be easy to modify the procedure to stop when n reaches a given value. The reader will find it easier to grasp the idea of the construction by trying it out manually, replacing the number $3 \cdot 4^{r-1}$ of step W4 by 2^r during this experiment.

Algorithm W is not meant to be a practical source of random numbers. It is intended to serve only a theoretical purpose:

Theorem W. *Let $\langle U_n \rangle$ be the sequence of rational numbers defined by Algorithm W, and let k be a positive integer. If the subsequence $\langle U_n \rangle_{\mathcal{R}_k}$ is infinite, it is 1-distributed.*

Proof. Let $A[a_1, \dots, a_r]$ denote the (possibly empty) subsequence of $\langle U_n \rangle$ containing precisely those elements U_n that, for all $j \leq r$, belong to subsequence $\langle U_n \rangle_{\mathcal{R}_j}$ if $a_j = 1$ and do not belong to subsequence $\langle U_n \rangle_{\mathcal{R}_j}$ if $a_j = 0$.

It suffices to prove, for all $r \geq 1$ and all pairs of binary numbers $a_1 \dots a_r$ and $b_1 \dots b_r$, that $\Pr(U_n \in I_{b_1 \dots b_r}) = 2^{-r}$ with respect to the subsequence $A[a_1, \dots, a_r]$, whenever the latter is infinite. (See Eq. (30).) For if $r \geq k$, the infinite sequence $\langle U_n \rangle_{\mathcal{R}_k}$ is the finite union of the disjoint subsequences $A[a_1, \dots, a_r]$ for $a_k = 1$ and $a_j = 0$ or 1 for $1 \leq j \leq r$, $j \neq k$; and it follows that $\Pr(U_n \in I_{b_1 \dots b_r}) = 2^{-r}$ with respect to $\langle U_n \rangle_{\mathcal{R}_k}$. (See exercise 33.) This is enough to show that the sequence is 1-distributed, by Theorem A.

Let $B[a_1, \dots, a_r]$ denote the subsequence of $\langle U_n \rangle$ that consists of the values for those n in which $C[a_1, \dots, a_r]$ is increased by one in step W6 of the algorithm. By the algorithm, $B[a_1, \dots, a_r]$ is a finite sequence with at most $3 \cdot 4^{r-1}$ elements. All but a finite number of the members of $A[a_1, \dots, a_r]$ come from the subsequences $B[a_1, \dots, a_r, \dots, a_t]$, where $a_j = 0$ or 1 for $r < j \leq t$.

Now assume that $A[a_1, \dots, a_r]$ is infinite, and let $A[a_1, \dots, a_r] = \langle U_{s_n} \rangle$, where $s_0 < s_1 < s_2 \leq \dots$. If N is a large integer, with $4^r \leq 4^q < N \leq 4^{q+1}$, it follows that the number of values of $k < N$ for which U_{s_k} is in $I_{b_1 \dots b_r}$ is (except for finitely many elements at the beginning of the subsequence)

$$\nu(N) = \nu(N_1) + \dots + \nu(N_m).$$

Here m is the number of subsequences $B[a_1, \dots, a_t]$ listed above in which U_{s_k} appears for some $k < N$; N_j is the number of values of k with U_{s_k} in the corresponding subsequence; and $\nu(N_j)$ is the number of such values that are also in $I_{b_1 \dots b_r}$. Therefore by Lemma T,

$$\begin{aligned} |\nu(N) - 2^{-r}N| &= |\nu(N_1) - 2^{-r}N_1 + \dots + \nu(N_m) - 2^{-r}N_m| \\ &\leq |\nu(N_1) - 2^{-r}N_1| + \dots + |\nu(N_m) - 2^{-r}N_m| \\ &\leq m \leq 1 + 2 + 4 + \dots + 2^{q-r+1} < 2^{q+1}. \end{aligned}$$

The inequality on m follows here from the fact that, by our choice of N , U_{s_N} is in $B[a_1, \dots, a_t]$ for some $t \leq q+1$.

We have proved that

$$|\nu(N)/N - 2^{-r}| \leq 2^{q+1}/N < 2/\sqrt{N}. \blacksquare$$

To show finally that sequences satisfying Definition R5 exist, we note first that if $\langle U_n \rangle$ is a $[0, 1]$ sequence of rational numbers and if \mathcal{R} is a computable subsequence rule for a b -ary sequence, we can make \mathcal{R} into a computable subsequence rule \mathcal{R}' for $\langle U_n \rangle$ by letting $f'_n(x_1, \dots, x_n)$ in \mathcal{R}' equal $f_n(\lfloor bx_1 \rfloor, \dots, \lfloor bx_n \rfloor)$ in \mathcal{R} . If the $[0, 1]$ sequence $\langle U_n \rangle \mathcal{R}'$ is equidistributed, so is the b -ary sequence $\langle \lfloor bU_n \rfloor \rangle \mathcal{R}$. Now the set of all computable subsequence rules for b -ary sequences, for all values of b , is countable (since only countably many effective algorithms are possible), so they may be listed in some sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$; therefore Algorithm W defines a $[0, 1]$ sequence that is random in the sense of Definition R5.

This brings us to a somewhat paradoxical situation. As we mentioned earlier, no effective algorithm can define a sequence that satisfies Definition R4, and for the same reason there is no effective algorithm that defines a sequence satisfying Definition R5. A proof of the existence of such random sequences is necessarily nonconstructive; how then can Algorithm W construct such a sequence?

There is no contradiction here; we have merely stumbled on the fact that the set of all effective algorithms cannot be enumerated by an effective algorithm. In other words, there is no effective algorithm to select the j th computable subsequence rule \mathcal{R}_j ; this happens because there is no effective algorithm to determine if a computational method ever terminates. (We shall return to this topic in Chapter 11.) Important large classes of algorithms can be systematically enumerated; thus, for example, Algorithm W shows that it is possible to construct, with an effective algorithm, a sequence that satisfies Definition R5 if we restrict consideration to subsequence rules that are “primitive recursive.”

By modifying step W6 of Algorithm W, so that it sets $U_n \leftarrow V_{k+t}$ instead of V_k , where t is any nonnegative integer depending on a_1, \dots, a_r , we can show that there are *uncountably* many $[0, 1]$ sequences satisfying Definition R5.

The following theorem shows still another way to prove the existence of uncountably many random sequences, using a less direct argument based on measure theory, even if the strong definition R6 is used:

Theorem M. *Let the real number x , $0 \leq x < 1$, correspond to the binary sequence $\langle X_n \rangle$ if the binary representation of x is $(0.X_0X_1\dots)_2$. Under this correspondence, almost all x correspond to binary sequences that are random in the sense of Definition R6. (In other words, the set of all real x that correspond to a binary sequence that is nonrandom by Definition R6 has measure zero.)*

Proof. Let S be an effective algorithm that determines an infinite sequence of distinct nonnegative integers $\langle s_n \rangle$, where the choice of s_n depends only on n and X_{s_k} for $0 \leq k < n$; and let \mathcal{R} be a computable subsequence rule. Then any binary sequence $\langle X_n \rangle$ leads to a subsequence $\langle X_{s_n} \rangle \mathcal{R}$, and Definition R6 says this subsequence must either be finite or 1-distributed. It suffices to prove that for fixed \mathcal{R} and S the set $N(\mathcal{R}, S)$ of all real x corresponding to $\langle X_n \rangle$, such that $\langle X_{s_n} \rangle \mathcal{R}$ is infinite and not 1-distributed, has measure zero. For x has a nonrandom binary representation if and only if x is in $\bigcup N(\mathcal{R}, S)$, taken over the countably many choices of \mathcal{R} and S .

Therefore let \mathcal{R} and S be fixed. Consider the set $T(a_1 a_2 \dots a_r)$ defined for all binary numbers $a_1 a_2 \dots a_r$ as the set of all x corresponding to (X_n) , such that $(X_{s_n})\mathcal{R}$ has $\geq r$ elements whose first r elements are respectively equal to a_1, a_2, \dots, a_r . Our first result is that

$$T(a_1 a_2 \dots a_r) \text{ has measure } \leq 2^{-r}. \quad (32)$$

To prove this, we start by observing that $T(a_1 a_2 \dots a_r)$ is a measurable set: Each element of $T(a_1 a_2 \dots a_r)$ is a real number $x = (0.X_0 X_1 \dots)_2$ for which there exists an integer m such that algorithm S determines distinct values s_0, s_1, \dots, s_m , and rule \mathcal{R} determines a subsequence of $X_{s_0}, X_{s_1}, \dots, X_{s_m}$ such that X_{s_m} is the r th element of this subsequence. The set of all real $y = (0.Y_0 Y_1 \dots)_2$ such that $Y_{s_k} = X_{s_k}$ for $0 \leq k \leq m$ also belongs to $T(a_1 a_2 \dots a_r)$, and this is a measurable set consisting of the finite union of dyadic subintervals $I_{b_1 \dots b_t}$. Since there are only countably many such dyadic intervals, we see that $T(a_1 a_2 \dots a_r)$ is a countable union of dyadic intervals, and it is therefore measurable. Furthermore, this argument can be extended to show that the measure of $T(a_1 \dots a_{r-1} 0)$ equals the measure of $T(a_1 \dots a_{r-1} 1)$, since the latter is a union of dyadic intervals obtained from the former by requiring that $Y_{s_k} = X_{s_k}$ for $0 \leq k < m$ and $Y_{s_m} \neq X_{s_m}$. Now since

$$T(a_1 \dots a_{r-1} 0) \cup T(a_1 \dots a_{r-1} 1) \subseteq T(a_1 \dots a_{r-1}),$$

the measure of $T(a_1 a_2 \dots a_r)$ is at most one-half the measure of $T(a_1 \dots a_{r-1})$. The inequality (32) follows by induction on r .

Now that (32) has been established, the remainder of the proof is essentially to show that the binary representations of almost all real numbers are equidistributed. The next few paragraphs constitute a rather long but not difficult proof of this fact, and they serve to provide probability estimates that are useful in many other problems.

For $0 < \epsilon < 1$, let $B(r, \epsilon)$ be $\bigcup T(a_1 \dots a_r)$, where the union is taken over all binary numbers $a_1 \dots a_r$ for which the number $\nu(r)$ of zeros among $a_1 \dots a_r$ satisfies

$$|\nu(r) - \frac{1}{2}r| \geq 1 + \epsilon r.$$

The number of such binary numbers is $C(r, \epsilon) = \sum \binom{r}{k}$ summed over the values of k with $|k - \frac{1}{2}r| \geq 1 + \epsilon r$. A suitable estimate for the tail of the binomial distribution can be obtained by the following standard trick: Let x and p be any positive numbers less than 1, let $q = 1 - p$, and let $s = (p + \epsilon)r$. Then

$$\sum_{k \geq s} \binom{r}{k} p^k q^{r-k} \leq \sum_{k \geq s} \binom{r}{k} p^k q^{r-k} x^{s-k} \leq \sum_{k \geq 0} \binom{r}{k} p^k q^{r-k} x^{s-k} = x^s \left(q + \frac{p}{x} \right)^r.$$

By elementary calculus, the minimum value of $x^s (q + p/x)^r$ occurs when $x = (p/(p + \epsilon))/(q/(q - \epsilon))$, and this value of x yields

$$\sum_{k \geq (p+\epsilon)r} \binom{r}{k} p^k q^{r-k} \leq \left(\left(\frac{p}{p+\epsilon} \right)^{p+\epsilon} \left(\frac{q}{q-\epsilon} \right)^{q-\epsilon} \right)^r.$$

Now when $\epsilon \leq \min(p, q)$ we have

$$\ln(p/(p+\epsilon))^{p+\epsilon} = -\epsilon - \frac{\epsilon^2}{2p} + \frac{\epsilon^3}{6p^2} - \frac{\epsilon^4}{12p^3} + \dots < -\epsilon,$$

$$\ln(q/(q-\epsilon))^{q-\epsilon} = +\epsilon - \frac{\epsilon^2}{2q} - \frac{\epsilon^3}{6q^2} - \frac{\epsilon^4}{12q^3} - \dots < \epsilon - \frac{\epsilon^2}{2q},$$

so we have the following bound for all $r > 0$ and $0 \leq \epsilon \leq \min(p, q)$:

$$\sum_{k \geq (p+\epsilon)r} \binom{r}{k} p^k q^{r-k} < e^{-\epsilon^2 r / (2q)}. \quad (33)$$

But $C(r, \epsilon)$ is 2^{r+1} times this left-hand quantity, in the special case $p = q = \frac{1}{2}$, hence by (32)

$$B(r, \epsilon) \text{ has measure } \leq 2^{-r} C(r, \epsilon) < 2e^{-\epsilon^2 r}.$$

The next step is to define

$$B^*(r, \epsilon) = B(r, \epsilon) \cup B(r+1, \epsilon) \cup B(r+2, \epsilon) \cup \dots$$

The measure of $B^*(r, \epsilon)$ is at most $\sum_{k \geq r} 2e^{-\epsilon^2 k}$, and this is the remainder of a convergent series, so

$$\lim_{r \rightarrow \infty} (\text{measure of } B^*(r, \epsilon)) = 0. \quad (34)$$

Now if x is a real number whose binary expansion $(0.X_0X_1\dots)_2$ leads to an infinite sequence $\langle X_{s_n} \rangle_{\mathcal{R}}$ that is not 1-distributed, and if $\nu(r)$ denotes the number of zeros in the first r elements of the latter sequence, then

$$|\nu(r)/r - \frac{1}{2}| \geq 2\epsilon,$$

for some $\epsilon > 0$ and infinitely many r . This means x is in $B^*(r, \epsilon)$ for all r . So finally we find that

$$N(\mathcal{R}, \mathcal{S}) = \bigcup_{t \geq 2} \bigcap_{r \geq 1} B^*(r, 1/t);$$

and, by (34), $\bigcap_{r \geq 1} B^*(r, 1/t)$ has measure zero for all t . Hence $N(\mathcal{R}, \mathcal{S})$ has measure zero. ■

From the existence of *binary* sequences satisfying Definition R6, we can show the existence of $[0, 1]$ sequences that are random in this sense. For details, see exercise 36. The consistency of Definition R6 is thereby established.

E. Random finite sequences. An argument was given above to indicate that it is impossible to define the concept of randomness for finite sequences; any given finite sequence is as likely as any other. Still, nearly everyone would agree that the sequence 011101001 is “more random” than 101010101, and even the latter

sequence is "more random" than 000000000. Although it is true that truly random sequences will exhibit locally nonrandom behavior, we would expect such behavior only in a long finite sequence, not in a short one.

Several ways for defining the randomness of a finite sequence have been proposed, and only a few of the ideas will be sketched here. Let us consider only the case of b -ary sequences.

Given a b -ary sequence X_1, X_2, \dots, X_N , we can say that

$$\Pr(S(n)) \approx p, \quad \text{if } |\nu(N)/N - p| \leq 1/\sqrt{N},$$

where $\nu(n)$ is the quantity appearing in Definition A at the beginning of this section. The above sequence can be called " k -distributed" if

$$\Pr(X_n X_{n+1} \dots X_{n+k-1} = x_1 x_2 \dots x_k) \approx 1/b^k$$

for all b -ary numbers $x_1 x_2 \dots x_k$. (Cf. Definition D; unfortunately a sequence may be k -distributed by this new definition when it is not $(k-1)$ -distributed.)

A definition of randomness may now be given analogous to Definition R1, as follows:

Definition Q1. A b -ary sequence of length N is "random" if it is k -distributed (in the above sense) for all positive integers $k \leq \log_b N$.

According to this definition, for example, there are 170 nonrandom binary sequences of length 11:

00000001111	10000000111	11000000011	11100000001
00000001110	10000000110	11000000010	11100000000
00000001101	10000000101	11000000001	10100000001
00000001011	10000000011	01000000011	01100000001
00000000111			

plus 01010101010 and all sequences with nine or more zeros, plus all sequences obtained from the preceding sequences by interchanging ones and zeros.

Similarly, we can formulate a definition for finite sequences analogous to Definition R6. Let \mathbf{A} be a set of algorithms, each of which is a combination selection and choice procedure that gives a subsequence $\langle X_{s_n} \rangle \mathcal{R}$, as in the proof of Theorem M.

Definition Q2. The b -ary sequence X_1, X_2, \dots, X_N is (n, ϵ) -random with respect to a set of algorithms \mathbf{A} , if for every subsequence $X_{t_1}, X_{t_2}, \dots, X_{t_m}$ determined by an algorithm of \mathbf{A} we have either $m < n$ or

$$\left| \frac{1}{m} \nu_a(X_{t_1}, \dots, X_{t_m}) - \frac{1}{b} \right| \leq \epsilon \quad \text{for } 0 \leq a < b.$$

Here $\nu_a(x_1, \dots, x_m)$ is the number of a 's in the sequence x_1, \dots, x_m .

(In other words, every sufficiently long subsequence determined by an algorithm of \mathbf{A} must be approximately equidistributed.) The basic idea in this case is to let \mathbf{A} be a set of “simple” algorithms; the number (and the complexity) of the algorithms in \mathbf{A} can grow as N grows.

As an example of Definition Q2, let us consider binary sequences, and let \mathbf{A} be just the following four algorithms:

- Take the whole sequence.
- Take alternate terms of the sequence, starting with the first.
- Take the terms of the sequence following a zero.
- Take the terms of the sequence following a one.

Now a sequence X_1, \dots, X_8 is $(4, \frac{1}{8})$ -random if:

by (a), $|\frac{1}{8}(X_1 + \dots + X_8) - \frac{1}{2}| \leq \frac{1}{8}$, i.e., if there are 3, 4, or 5 ones;

by (b), $|\frac{1}{4}(X_1 + X_3 + X_5 + X_7) - \frac{1}{2}| \leq \frac{1}{8}$, i.e., if there are two ones in odd-numbered positions;

by (c), there are three possibilities depending on how many zeros occupy positions X_1, \dots, X_7 : if there are 2 or 3 zeros here, there is no condition to test (since $n = 4$); if there are 4 zeros, they must respectively be followed by two zeros and two ones; and if there are 5 zeros, they must respectively be followed by two or three zeros;

by (d), we get conditions similar to those implied by (c).

It turns out that only the following binary sequences of length 8 are $(4, \frac{1}{8})$ -random with respect to these rules:

00001011	00101001	01001110	01101000
00011010	00101100	01011011	01101100
00011011	00110010	01011110	01101101
00100011	00110011	01100010	01110010
00100110	00110110	01100011	01110110
00100111	00111001	01100110	

plus those obtained by interchanging 0 and 1 consistently.

It is clear that we could make the set of algorithms so large that no sequences satisfy the definition, when n and ϵ are reasonably small. A. N. Kolomogorov has proved that an (n, ϵ) -random binary sequence will always exist, for any given N , if the number of algorithms in \mathbf{A} does not exceed

$$\frac{1}{2}e^{2n\epsilon^2(1-\epsilon)}. \quad (35)$$

This result is not nearly strong enough to show that sequences satisfying Definition Q1 will exist, but the latter can be constructed efficiently using the procedure of Rees in exercise 3.2.2–21.

Still another interesting approach to a definition of randomness has been taken by Per Martin-Löf [Information and Control 9 (1966), 602–619]. Given a

finite b -ary sequence X_1, \dots, X_N , let $l(X_1, \dots, X_N)$ be the length of the shortest Turing machine program that generates this sequence. (For a definition of Turing machines, see Chapter 11; alternatively, we could use certain classes of effective algorithms, such as those defined in exercise 1.1–8.) Then $l(X_1, \dots, X_N)$ is a measure of the “patternlessness” of the sequence, and we may equate this idea with randomness. The sequences of length N that maximize $l(X_1, \dots, X_N)$ may be called random. (From the standpoint of practical random number generation by computer, this is, of course, the worst definition of “randomness” that can be imagined!)

Essentially the same definition of randomness was independently given by G. Chaitin at about the same time; see *JACM* **16** (1969), 145–159. It is interesting to note that even though this definition makes no reference to equidistribution properties as our other definitions have, Martin-Löf and Chaitin have proved that random sequences of this type also have the expected equidistribution properties. In fact, Martin-Löf has demonstrated that such sequences satisfy *all* computable statistical tests for randomness, in an appropriate sense.

For further developments in the definition of random finite sequences, see A. K. Zvonkin and L. A. Levin, *Uspekhi Mat. Nauk* **25**, 6 (Nov. 1970), 85–127 [English translation in *Russian Math. Surveys* **25**, 6 (Nov. 1970), 83–124]; L. A. Levin, *Doklady Akad. Nauk SSSR* **212** (1973), 548–550.

F. Summary, history, and bibliography. We have defined several degrees of randomness that a sequence might possess.

An infinite sequence that is ∞ -distributed satisfies a great many useful properties that are expected of random sequences, and there is a rich theory concerning ∞ -distributed sequences. (The exercises that follow develop several important properties of ∞ -distributed sequences that have not been mentioned in the text.) Definition R1 is therefore an appropriate basis for theoretical studies of randomness.

The concept of an ∞ -distributed b -ary sequence was introduced in 1909 by Emile Borel. He essentially defined the concept of an (m, k) -distributed sequence, and showed that the b -ary representations of almost all real numbers are (m, k) -distributed for all m and k . He called such numbers *normal* to base b . An excellent discussion of this topic appears in his well-known book, *Leçons sur la théorie des fonctions* (2nd ed., 1914), 182–216.

The notion of an ∞ -distributed sequence of real numbers, also called a *completely equidistributed sequence*, first appeared in a note by N. M. Korobov in *Doklady Akad. Nauk SSSR* **62** (1948), 21–22. Korobov and several of his colleagues developed the theory of such sequences quite extensively in a series of papers during the 1950s. Completely equidistributed sequences were independently studied by Joel N. Franklin, *Math. Comp.* **17** (1963), 28–59, in a paper that is particularly noteworthy because it was inspired by the problem of random number generation. The book *Uniform Distribution of Sequences* by L. Kuipers and H. Niederreiter (New York: Wiley, 1974) is an extraordinarily complete source of information about the rich mathematical literature concerning k -distributed sequences of all kinds.

We have seen, however, that ∞ -distributed sequences need not be sufficiently haphazard to qualify completely as "random." Three definitions, R4, R5, and R6, were formulated above to provide the additional conditions; and Definition R6, in particular, seems to be an appropriate way to define the concept of an infinite random sequence. It is a precise, quantitative statement that may well coincide with the intuitive idea of true randomness.

Historically, the development of these definitions was primarily influenced by R. von Mises' quest for a good definition of "probability." In *Math. Zeitschrift* 5 (1919), 52–99, von Mises proposed a definition similar in spirit to Definition R5, although stated too strongly (like our Definition R3) so that no sequences satisfying the conditions could possibly exist. Many people noticed this discrepancy, and A. H. Copeland [*Amer. J. Math.* 50 (1928), 535–552] suggested weakening von Mises' definition by substituting what he called "admissible numbers" (or Bernoulli sequences). These are equivalent to ∞ -distributed $[0, 1]$ sequences in which all entries U_n have been replaced by 1 if $U_n < p$ or by 0 if $U_n \geq p$, for a given probability p . Thus Copeland was essentially suggesting a return to Definition R1. Then Abraham Wald showed that it is not necessary to weaken von Mises' definition so drastically, and he proposed substituting a countable set of subsequence rules. In an important paper [*Ergebnisse eines math. Kolloquiums* 8 (Vienna, 1937), 38–72], Wald essentially proved Theorem W, although he made the erroneous assertion that the sequence constructed by Algorithm W also satisfies the stronger condition that $\Pr(U_n \in A) = \text{measure of } A$, for all Lebesgue measurable $A \subseteq [0, 1]$. We have observed that no sequence can satisfy this property.

The concept of "computability" was still very much in its infancy when Wald wrote his paper, and A. Church [*Bull. Amer. Math. Soc.* 47 (1940), 130–135] showed how the precise notion of "effective algorithm" could be added to Wald's theory to make his definitions completely rigorous. The extension to Definition R6 was due essentially to A. N. Kolmogorov [*Sankhyā (A)* 25 (1963), 369–376], who proposed Definition Q2 for finite sequences in that same paper. Another definition of randomness for finite sequences, somewhere "between" Definitions Q1 and Q2, had been formulated many years earlier by A. S. Besicovitch [*Math. Zeitschrift* 39 (1934), 146–156].

The publications of Church and Kolmogorov considered only binary sequences for which $\Pr(X_n = 1) = p$ for a given probability p . Our discussion in this section has been slightly more general, since a $[0, 1]$ sequence essentially represents all p at once. The von Mises–Wald–Church definition has been refined in yet another interesting way by J. V. Howard, *Zeitschr. f. math. Logik und Grundlagen d. Math.* 21 (1975), 215–224.

Another important contribution was made by Donald W. Loveland [*Zeitschr. f. math. Logik und Grundlagen d. Math.* 12 (1966), 279–294], who discussed Definitions R4, R5, R6, and several intermediate concepts. Loveland proved that there are R5-random sequences that do not satisfy R4, thereby establishing the need for a stronger definition such as R6. In fact, he defined a rather simple permutation $\langle f(n) \rangle$ of the nonnegative integers, and an Algorithm W' analogous

to Algorithm W, such that $\overline{\Pr}(U_{f(n)} \geq \frac{1}{2}) - \underline{\Pr}(U_{f(n)} \geq \frac{1}{2}) \geq \frac{1}{2}$ for every R5-random sequence $\langle U_n \rangle$ produced by Algorithm W' when it is given an infinite set of subsequence rules \mathcal{R}_k .

Although Definition R6 is intuitively much stronger than R4, it is apparently not a simple matter to prove this rigorously, and for several years it was an open question whether or not R4 implies R6. Finally Thomas Herzog and James C. Owings, Jr., discovered how to construct a large family of sequences that satisfy R4 but not R6. [See *Zeitschr. f. math. Logik und Grundlagen d. Math.* **22** (1976), 385–389.]

Kolmogorov wrote another significant paper [*Problemy Peredači Informatsii* **1** (1965), 3–11] in which he considered the problem of defining the “information content” of a sequence, and this work led to Chaitin and Martin-Löf’s interesting definition of finite random sequences via “patternlessness.” [See *IEEE Trans. IT-14* (1968), 662–664.]

For a philosophical discussion of random sequences, see K. R. Popper, *The Logic of Scientific Discovery* (London, 1959), especially the interesting construction on pp. 162–163, which he first published in 1934.

Further connections between random sequences and recursive function theory have been explored by D. W. Loveland, *Trans. Amer. Math. Soc.* **125** (1966), 497–510. See also C.-P. Schnorr [*Zeitschr. Wahr. verw. Geb.* **14** (1969), 27–35], who found strong relations between random sequences and the “species of measure zero” defined by L. E. J. Brouwer in 1919. Schnorr’s subsequent book *Zufälligkeit und Wahrscheinlichkeit* [*Lecture Notes in Math.* **218** (Berlin: Springer, 1971)] gives a detailed treatment of the entire subject of randomness and makes an excellent introduction to the ever-growing advanced literature on the topic.

EXERCISES

1. [10] Can a periodic sequence be equidistributed?
 2. [10] Consider the periodic binary sequence 0, 0, 1, 1, 0, 0, 1, 1, Is it 1-distributed? Is it 2-distributed? Is it 3-distributed?
 3. [M22] Construct a periodic ternary sequence that is 3-distributed.
 4. [HM22] Let $U_n = (2^{\lfloor \lg(n+1) \rfloor} / 3) \bmod 1$. What is $\Pr(U_n < \frac{1}{2})$?
 5. [HM14] Prove that $\Pr(S(n) \text{ and } T(n)) + \Pr(S(n) \text{ or } T(n)) = \Pr(S(n)) + \Pr(T(n))$, for any two statements $S(n)$, $T(n)$, provided that at least three of the limits exist. For example, if a sequence is 2-distributed, we would find that
- $$\Pr(u_1 \leq U_n < v_1 \text{ or } u_2 \leq U_{n+1} < v_2) = v_1 - u_1 + v_2 - u_2 - (v_1 - u_1)(v_2 - u_2).$$
6. [HM23] Let $S_1(n), S_2(n), \dots$ be an infinite sequence of statements about mutually disjoint events; i.e., $S_i(n)$ and $S_j(n)$ cannot simultaneously be true if $i \neq j$. Assume that $\Pr(S_j(n))$ exists for each $j \geq 1$. Show that $\underline{\Pr}(S_j(n))$ is true for some $j \geq 1$) $\geq \sum_{j \geq 1} \Pr(S_j(n))$, and give an example to show that equality need not hold.

7. [HM27] Let $\{S_{ij}(n)\}$ be a family of statements such that $\Pr(S_{ij}(n))$ exists for all $i, j \geq 1$. Assume that for all $n > 0$, $S_{ij}(n)$ is true for exactly one pair of integers i, j . If $\sum_{i,j \geq 1} \Pr(S_{ij}(n)) = 1$, does it follow that “ $\Pr(S_{ij}(n) \text{ is true for some } j \geq 1)$ ” exists for all $i \geq 1$, and that it equals $\sum_{j \geq 1} \Pr(S_{ij}(n))$?

8. [M15] Prove (13).

9. [HM20] Prove Lemma E. [Hint: Consider $\sum_{1 \leq j \leq m} (y_{jn} - \alpha)^2$.]

► **10.** [HM22] Where was the fact that m divides q used in the proof of Theorem C?

11. [M10] Use Theorem C to prove that if $\langle U_n \rangle$ is ∞ -distributed, so is the subsequence $\langle U_{2n} \rangle$.

12. [HM20] Show that a k -distributed sequence passes the “maximum-of- k test,” in the following sense: $\Pr(u \leq \max(U_n, U_{n+1}, \dots, U_{n+k-1}) < v) = v^k - u^k$.

► **13.** [HM27] Show that an ∞ -distributed $[0, 1]$ sequence passes the “gap test” in the following sense: If $0 \leq \alpha < \beta \leq 1$ and $p = \beta - \alpha$, let $f(0) = 0$, and for $n \geq 1$ let $f(n)$ be the smallest integer $m > f(n-1)$ such that $\alpha \leq U_m < \beta$; then

$$\Pr(f(n) - f(n-1) = k) = p(1-p)^{k-1}.$$

14. [HM25] Show that an ∞ -distributed sequence passes the “run test” in the following sense: If $f(0) = 1$ and if, for $n \geq 1$, $f(n)$ is the smallest integer $m > f(n-1)$ such that $U_{m-1} > U_m$, then

$$\Pr(f(n) - f(n-1) = k) = 2k/(k+1)! - 2(k+1)/(k+2)!.$$

► **15.** [HM30] Show that an ∞ -distributed sequence passes the “coupon-collector’s test” when there are only two kinds of coupons, in the following sense: Let X_1, X_2, \dots be an ∞ -distributed binary sequence. Let $f(0) = 0$, and for $n \geq 1$ let $f(n)$ be the smallest integer $m > f(n-1)$ such that $\{X_{f(n-1)+1}, \dots, X_m\}$ is the set $\{0, 1\}$. Prove that $\Pr(f(n) - f(n-1) = k) = 2^{1-k}$, for $k \geq 2$. (Cf. exercise 7.)

16. [HM38] Does the coupon-collector’s test hold for ∞ -distributed sequences when there are more than two kinds of coupons? (Cf. the previous exercise.)

17. [HM50] If r is any given rational number, Franklin has proved that the sequence $\langle r^n \bmod 1 \rangle$ is not 2-distributed. But is there any rational number r for which this sequence is equidistributed? In particular, is the sequence equidistributed when $r = \frac{3}{2}$? [Cf. K. Mahler, *Mathematika* 4 (1957), 122–124.]

► **18.** [HM22] Prove that if U_0, U_1, \dots is k -distributed, so is the sequence V_0, V_1, \dots , where $V_n = \lfloor nU_n \rfloor / n$.

19. [HM35] Consider a modification of Definition R4 that requires the subsequences to be only 1-distributed instead of ∞ -distributed. Is there a sequence that satisfies this weaker definition, but that is not ∞ -distributed? (Is the weaker definition really weaker?)

20. [HM47] Does the sequence $\langle \theta^n \bmod 1 \rangle$ satisfy Definition R4 for almost all real numbers $\theta > 1$?

21. [HM20] Let S be a set and let \mathcal{M} be a collection of subsets of S . Suppose that p is a real-valued function of the sets in \mathcal{M} , such that $p(M)$ denotes the probability that a “randomly” chosen element of S lies in M . Generalize Definitions B and D to obtain a good definition of the concept of a k -distributed sequence $\langle Z_n \rangle$ of elements of S with respect to the probability distribution p .

- 22. [HM30] (Hermann Weyl.) Show that the $[0, 1)$ sequence $\langle U_n \rangle$ is k -distributed if and only if

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{0 \leq n < N} \exp(2\pi i(c_1 U_n + \cdots + c_k U_{n+k-1})) = 0$$

for every set of integers c_1, c_2, \dots, c_k not all zero.

23. [M34] Show that a b -ary sequence $\langle X_n \rangle$ is k -distributed if and only if all of the sequences $\langle c_1 X_n + c_2 X_{n+1} + \cdots + c_k X_{n+k-1} \rangle$ are equidistributed, whenever c_1, c_2, \dots, c_k are integers with $\gcd(c_1, \dots, c_k) = 1$.

24. [M25] Show that a $[0, 1)$ sequence $\langle U_n \rangle$ is k -distributed if and only if all of the sequences $\langle c_1 U_n + c_2 U_{n+1} + \cdots + c_k U_{n+k-1} \rangle$ are equidistributed, whenever c_1, c_2, \dots, c_k are integers not all zero.

25. [HM20] A sequence is called a “white sequence” if all serial correlations are zero, i.e., if the equation in Corollary S is true for all $k \geq 1$. (By Corollary S, an ∞ -distributed sequence is white.) Show that if a $[0, 1)$ sequence is equidistributed, it is white if and only if

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{0 \leq j < n} (U_j - \frac{1}{2})(U_{j+k} - \frac{1}{2}) = 0, \quad \text{for all } k \geq 1.$$

26. [HM34] (J. Franklin.) A white sequence, as defined in the previous exercise, can definitely fail to be random. Let U_0, U_1, \dots be an ∞ -distributed sequence, and define the sequence V_0, V_1, \dots as follows:

$$\begin{aligned} (V_{2n-1}, V_{2n}) &= (U_{2n-1}, U_{2n}) && \text{if } (U_{2n-1}, U_{2n}) \in G, \\ (V_{2n-1}, V_{2n}) &= (U_{2n}, U_{2n-1}) && \text{if } (U_{2n-1}, U_{2n}) \notin G, \end{aligned}$$

where G is the set $\{(x, y) \mid x - \frac{1}{2} \leq y \leq x \text{ or } x + \frac{1}{2} \leq y\}$. Show that (a) V_0, V_1, \dots is equidistributed and white; (b) $\Pr(V_n > V_{n+1}) = \frac{5}{8}$. (This points out the weakness of the serial correlation test.)

27. [HM48] What is the highest possible value for $\Pr(V_n > V_{n+1})$ in an equidistributed, white sequence? (D. Coppersmith has constructed such a sequence achieving the value $\frac{7}{8}$.)

► 28. [HM21] Use the sequence (11) to construct a $[0, 1)$ sequence that is 3-distributed, for which $\Pr(U_{2n} \geq \frac{1}{2}) = \frac{3}{4}$.

29. [HM34] Let X_0, X_1, \dots be a $(2k)$ -distributed binary sequence. Show that

$$\overline{\Pr}(X_{2n} = 0) \leq \frac{1}{2} + \binom{2k-1}{k} / 2^{2k}.$$

► 30. [M39] Construct a binary sequence that is $(2k)$ -distributed, and for which

$$\Pr(X_{2n} = 0) = \frac{1}{2} + \binom{2k-1}{k} / 2^{2k}.$$

(Therefore the inequality in the previous exercise is the best possible.)

31. [M30] Show that $[0, 1]$ sequences exist that satisfy Definition R5, yet $\nu_n/n \geq \frac{1}{2}$ for all $n > 0$, where ν_n is the number of $j < n$ for which $U_n < \frac{1}{2}$. (This might be considered a nonrandom property of the sequence.)

32. [M24] Given that $\langle X_n \rangle$ is a “random” b -ary sequence according to Definition R5, and that \mathcal{R} is a computable subsequence rule that specifies an infinite subsequence $\langle X_n \rangle_{\mathcal{R}}$, show that the latter subsequence is not only 1-distributed, it is “random” by Definition R5.

33. [HM22] Let $\langle U_{r_n} \rangle$ and $\langle U_{s_n} \rangle$ be infinite disjoint subsequences of a sequence $\langle U_n \rangle$. (Thus, $r_0 < r_1 < r_2 < \dots$ and $s_0 < s_1 < s_2 < \dots$ are increasing sequences of integers and $r_m \neq s_n$ for any m, n .) Let $\langle U_{t_n} \rangle$ be the combined subsequence, so that $t_0 < t_1 < t_2 < \dots$ and the set $\{t_n\} = \{r_n\} \cup \{s_n\}$. Show that if $\Pr(U_{r_n} \in A) = \Pr(U_{s_n} \in A) = p$, then $\Pr(U_{t_n} \in A) = p$.

► **34.** [M25] Define subsequence rules $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \dots$ such that Algorithm W can be used with these rules to give an effective algorithm to construct a $[0, 1]$ sequence satisfying Definition R1.

► **35.** [HM35] (D. W. Loveland.) Show that if a binary sequence $\langle X_n \rangle$ is R5-random, and if $\langle s_n \rangle$ is any computable sequence as in Definition R4, then $\overline{\Pr}(X_{s_n} = 1) \geq \frac{1}{2}$ and $\underline{\Pr}(X_{s_n} = 1) \leq \frac{1}{2}$.

36. [HM30] Let $\langle X_n \rangle$ be a binary sequence that is “random” according to Definition R6. Show that the $[0, 1]$ sequence $\langle U_n \rangle$ defined in binary notation by the scheme

$$\begin{aligned} U_0 &= (0.X_0)_2 \\ U_1 &= (0.X_1X_2)_2 \\ U_2 &= (0.X_3X_4X_5)_2 \\ U_3 &= (0.X_6X_7X_8X_9)_2 \\ &\dots \end{aligned}$$

is random in the sense of Definition R6.

37. [M37] (D. Coppersmith.) Define a sequence that satisfies Definition R4 but not Definition R5. [Hint: Consider changing $U_0, U_1, U_4, U_9, \dots$ in a truly random sequence.]

38. [M49] (A. N. Kolmogorov.) Given N, n and ϵ , what is the smallest number of algorithms in a set \mathbf{A} such that no (n, ϵ) -random binary sequences of length N exist with respect to \mathbf{A} ? (If exact formulas cannot be given, can asymptotic formulas be found? The point of this problem is to discover how close the bound (35) comes to being “best possible.”)

39. [HM45] (W. M. Schmidt.) Let U_n be a $[0, 1]$ sequence, and let $\nu_n(u)$ be the number of nonnegative integers $j \leq n$ such that $0 \leq U_j < u$. Prove that there is a positive constant c such that, for any N and for any $[0, 1]$ sequence $\langle U_n \rangle$, we have

$$|\nu_n(u) - un| > c \ln N$$

for some n and u with $0 \leq n < N$, $0 \leq u < 1$. (In other words, no $[0, 1]$ sequence can be too equidistributed.)

► **40.** [16] (I. J. Good.) Can a valid table of random digits contain just one misprint?

3.6. SUMMARY

WE HAVE COVERED a fairly large number of topics in this chapter: how to generate random numbers, how to test them, how to modify them in applications, and how to derive theoretical facts about them. Perhaps the main question in many readers' minds will be, "What is the result of all this theory? What is a simple, virtuous generator I can use in my programs in order to have a reliable source of random numbers?"

The detailed investigations in this chapter suggest that the following procedure gives the "nicest" and "simplest" random number generator for the machine language of most computers: At the beginning of the program, set an integer variable X to some value X_0 . This variable X is to be used only for the purpose of random number generation. Whenever a new random number is required by the program, set

$$X \leftarrow (aX + c) \bmod m \quad (1)$$

and use the new value of X as the random value. It is necessary to choose X_0 , a , c , and m properly, and to use the random numbers wisely, according to the following principles:

- i) The "seed" number X_0 may be chosen arbitrarily. If the program is run several times and a different source of random numbers is desired each time, set X_0 to the last value attained by X on the preceding run; or (if more convenient) set X_0 to the current date and time. If the program may need to be rerun later with the same random numbers (e.g., when debugging), be sure to print out X_0 if it isn't otherwise known.
- ii) The number m should be large, say at least 2^{30} . It may conveniently be taken as the computer's word size, since this makes the computation of $(aX + c) \bmod m$ quite efficient. Section 3.2.1.1 discusses the choice of m in more detail. The computation of $(aX + c) \bmod m$ must be done exactly, with no roundoff error.
- iii) If m is a power of 2 (i.e., if a binary computer is being used), pick a so that $a \bmod 8 = 5$. If m is a power of 10 (i.e., if a decimal computer is being used), choose a so that $a \bmod 200 = 21$. This choice of a together with the choice of c given below ensures that the random number generator will produce all m different possible values of X before it starts to repeat (see Section 3.2.1.2) and ensures high "potency" (see Section 3.2.1.3).
- iv) The multiplier a should preferably be chosen between $.01m$ and $.99m$, and its binary or decimal digits should not have a simple, regular pattern. By choosing some haphazard constant like $a = 3141592621$ (which satisfies both of the conditions in (iii)), one almost always obtains a reasonably good multiplier. Further testing should of course be done if the random number generator is to be used extensively; for example, there should be no large quotients when Euclid's algorithm is used to find the gcd of a and m (see Section 3.3.3). The multiplier should pass the spectral test (Section 3.3.4).

and several tests of Section 3.3.2, before it is considered to have a truly clean bill of health.

- v) The value of c is immaterial when a is a good multiplier, except that c must have no factor in common with m . Thus we may choose $c = 1$ or $c = a$.
- vi) The least significant (right-hand) digits of X are not very random, so decisions based on the number X should always be influenced primarily by the most significant digits. It is generally best to think of X as a random fraction X/m between 0 and 1, that is, to visualize X with a decimal point at its left, rather than to regard X as a random integer between 0 and $m - 1$. To compute a random integer between 0 and $k - 1$, one should multiply by k and truncate the result (see the beginning of Section 3.4.2).
- vii) An important limitation on the randomness of sequence (1) is discussed in Section 3.3.4, where it is shown that the “accuracy” in t dimensions will be only about one part in \sqrt{m} . Monte Carlo applications requiring higher resolution can improve the randomness by employing techniques discussed in Section 3.2.2.

The above comments apply primarily to machine-language coding. In higher-level programming languages, we are often unable to use such machine-dependent features as integer arithmetic modulo the word size, and careful compilers will not allow us to compute the product of two large integers. Another technique that we might call the *subtractive method* (exercise 3.2.2–23) can be used to provide a “portable” random number generator that is efficiently describable in any higher-level programming language, since it makes use only of integer arithmetic between -10^9 and $+10^9$. Here is how the subtractive method might be coded in FORTRAN, as a subroutine that delivers an array of 55 random integers at once:

```

FUNCTION IRN55(IA)
DIMENSION IA(1)
C ASSUMING THAT IA(1), . . . , IA(55) HAVE BEEN SET UP PROPERLY,
C THIS SUBROUTINE RESETS THE IA ARRAY TO THE NEXT 55 NUMBERS
C OF A PSEUDO-RANDOM SEQUENCE, AND RETURNS THE VALUE 1.
DO 1 I = 1, 24
    J = IA(I) - IA(I+31)
    IF (J .LT. 0) J = J + 1000000000
    IA(I) = J
1 CONTINUE
DO 2 I = 25, 55
    J = IA(I) - IA(I-24)
    IF (J .LT. 0) J = J + 1000000000
    IA(I) = J
2 CONTINUE
IRN55=1
RETURN
END

```

To use these numbers in a FORTRAN program, let JRAND be an auxiliary integer variable; we may obtain the next random number U (as a fraction between 0 and 1) by writing the following three statements:

```
JRAND = JRAND + 1
IF (JRAND .GT. 55) JRAND = IRN55(IA)
U = FLOAT(IA(JRAND)) * 1.0E-9
```

At the beginning of our program we need to write "DIMENSION IA(55)" and "JRAND=55" and we must also initialize the IA array. Appropriate initialization can be done by calling the following subroutine with IX set to any integer value (selected like X_0 in rule (i) above, preferably large):

```
SUBROUTINE IN55(IA, IX)
DIMENSION IA(1)
C THIS SUBROUTINE SETS IA(1), ..., IA(55) TO STARTING
C VALUES SUITABLE FOR LATER CALLS ON IRN55(IA).
C IX IS AN INTEGER "SEED" VALUE BETWEEN 0 AND 999999999.
IA(55) = IX
J = IX
K = 1
DO 1 I = 1, 54
    II = MOD(21*I, 55)
    IA(II) = K
    K = J - K
    IF (K .LT. 0) K = K + 1000000000
    J = IA(II)
1 CONTINUE
C THE NEXT THREE LINES "WARM UP" THE GENERATOR.
IRN55(IA)
IRN55(IA)
IRN55(IA)
RETURN
END
```

(This subroutine computes a Fibonacci-like sequence; multiplication of indices by 21 spreads the values around so as to alleviate initial nonrandomness problems such as those in exercise 3.2.2-2. Note that $2^{29} < 10^9 < 2^{30}$; any large even number may actually be substituted for 10^9 in these routines, if a corresponding change is made in the computation of the random fraction U. Furthermore it would be possible to work directly with floating point numbers instead of integers by making appropriate changes to these programs, provided that the computer's floating point arithmetic is sufficiently accurate to give exact results in all the computations required here. Most binary computers will be able to meet this requirement when all of the numbers to be dealt with have the form $a/2^p$, where a is an integer, $0 \leq a < 2^p$, and there are p bits of precision in floating point fractions. The numbers (24, 55) in these routines may be replaced by any pair

of values (j, k) in Table 3.2.2-1, for $k \geq 50$; the constants 31, 25, 54, 21 should then be replaced by $k - j$, $j + 1$, $k - 1$, and d respectively, where d is relatively prime to k and $d \approx 0.382k$.)

Although a great deal is known about linear congruential sequences like (1), very little has yet been proved about the randomness properties of the subtractive method. Both approaches seem to be reliable in practice.

Unfortunately, quite a bit of published material in existence at the time this chapter was written recommends the use of generators that violate the suggestions above; and the most common generator in actual use, RANDU, is really horrible (cf. Section 3.3.4). The authors of many contributions to the science of random number generation were unaware that particular methods they were advocating would prove to be inadequate. Perhaps further research will show that even the random number generators recommended here are unsatisfactory; we hope this is not the case, but the history of the subject warns us to be cautious. The most prudent policy for a person to follow is to run each Monte Carlo program at least twice using quite different sources of random numbers, before taking the answers of the program seriously; this not only will give an indication of the stability of the results, it also will guard against the danger of trusting in a generator with hidden deficiencies. (Every random number generator will fail in at least one application.)

Excellent bibliographies of the pre-1972 literature on random number generation have been compiled by Richard E. Nance and Claude Overstreet, Jr., *Computing Reviews* 13 (1972), 495–508, and by E. R. Sowey, *International Stat. Review* 40 (1972), 355–371. The period 1972–1976 is covered by Sowey in *International Stat. Review* 46 (1978), 89–102.

For a detailed study of the use of random numbers in numerical analysis, see J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods* (London: Methuen, 1964). This book shows that some numerical methods are enhanced by using numbers that are “quasi-random,” designed specifically for a certain purpose (not necessarily satisfying the statistical tests we have discussed).

Every reader is urged to work exercise 6 in the following set of problems.

EXERCISES

1. [21] Write a MIX subroutine with the following characteristics, using method (1):

Calling sequence: JMP RANDI

Entry conditions: rA = k , a positive integer < 5000 .

Exit conditions: rA \leftarrow a random integer Y , $1 \leq Y \leq k$, with each integer about equally probable; rX = ?; overflow off.

- 2. [15] Some people have been afraid that computers will someday take over the world; but they are reassured by the statement that a machine cannot do anything really new, since it is only obeying the commands of its master, the programmer. Lady Lovelace wrote in 1844, “The Analytical Engine has no pretensions to originate

anything. It can do whatever we know how to order it to perform." Her statement has been further elaborated by many philosophers. Discuss this topic, with random number generators in mind.

3. [32] (*A dice game.*) Write a program that simulates a roll of two dice, each of which takes on the values 1, 2, ..., 6 with equal probability. If the total is 7 or 11 on the first roll, the game is won; a total of 2, 3, or 12 loses; and on any other total, call that total the "point" and continue rolling dice until either a 7 occurs (a loss) or the point occurs again (a win).

Play ten games. The result of each roll of the dice should be printed in the form $m\ n$, where m and n are the contents of the two dice, followed by some appropriate comment (like "snake eyes" or "little Joe" or "the hard way", etc.).

4. [40] (*Solitaire or patience.*) Some people spend a lot of valuable time playing card games of solitaire, and perhaps automation will make an important inroad in this area. Write a program that (a) shuffles a simulated deck of cards; (b) plays some common game of solitaire based on the order of the cards in the deck; (c) prints out the result of the game, i.e., how close the program came to winning. Several games should be played. The program might be set up to "cheat" upon request.

5. [46] (*Creative writing by computer.*) A television program entitled "The Thinking Machine," broadcast by the CBS television network on October 26, 1960, featured (among other things) two Western-style playlets that were written by a computer program. Here are the two scripts as they were printed out by the computer:

Saga #1. (The gun is in the right hand; the money is in the left hand; the glass is on the table; the bottle is on the table; the holster is on the robber; the sheriff's gun is in the sheriff's right hand; the sheriff's holster is on the sheriff.)

ROBBER: (The robber is at the window.) Go to door; open door; go thru door; close door; go to corner; put money down at corner; go to table; put gun on table; sit and fidgit; sit at table; pick up glass with right hand (empty); put glass on table; pick up bottle with right hand; pour; put bottle on table; pick up glass with right hand; take a drink from glass; put glass on table; pick up bottle with right hand; sit at table; sit at table; go to corner; go to window; go to table; put bottle on table; sit and fidgit; sit at table; sit and fidgit; go to window; go to table; pick up glass with right hand.

SHERIFF: (The sheriff is at the window.) See robber; (robber sees sheriff); go to door.

ROBBER: Put glass on table; pick up gun with right hand; check gun.

SHERIFF: Wait; open door; see robber; (robber sees sheriff); go thru door.

ROBBER: Go to window; aim; fire; SHERIFF NICKED.

SHERIFF: Go to window; aim; fire; MISSED; go to door; go to window.

ROBBER: Go to door; aim; aim.

SHERIFF: Aim; fire; MISSED.

ROBBER: Fire; SHERIFF NICKED.

SHERIFF: Go to door; aim; fire; MISSED; go thru door; aim.

ROBBER: Aim; fire; MISSED; aim; fire; MISSED.

SHERIFF: Fire; MISSED; go to window; aim; fire; MISSED.

ROBBER: Aim; fire; MISSED; aim; fire; MISSED; aim; fire; SHERIFF NICKED.

SHERIFF: Aim; fire; ROBBER HIT.

ROBBER: Drop gun; robber dies.



SHERIFF: Put gun in holster; go to table; pick up glass with right hand (empty); take glass from right hand with left hand; pick up bottle with right hand; pour; put bottle on table; take glass from left hand with right hand; take a drink from glass; take glass from right hand with left hand; pick up bottle with right hand; pour; put bottle on table; take glass from left hand with right hand; take a drink from glass; put glass on table; go to corner; pick up money with right hand; go to door; go thru door; close door. CURTAIN.

Saga #2. (The gun is in the right hand; the money is in the left hand; the glass is on the table; the bottle is on the table; the holster is on the robber; the sheriff's gun is in the sheriff's right hand; the sheriff's holster is on the sheriff.)

ROBBER: (The robber is at the window.) Go to door; open door; go thru door; close door; go to corner; put money down at corner; go to window; put gun down at window; lean on window and look; lean on window and look; go to corner; count money; go to table; pick up glass with right hand (empty); take glass from right hand with left hand; pick up bottle with right hand; pour; put bottle on table; take glass from left hand with right hand; take a drink from glass; put glass on table; pick up bottle with right hand; pour; go to corner; put bottle down at corner; go to window; pick up gun with right hand; check gun; put gun in holster; go to table; pick up glass with right hand; take a drink from glass; go to window; put glass down at window.

SHERIFF: (The sheriff is at the window.) See robber; (robber sees sheriff); go to door.

ROBBER: Take gun from holster with right hand; check gun; go to door; check gun; put gun down at door.

SHERIFF: Open door; see robber; (robber sees sheriff); go thru door; go to window.

ROBBER: Pick up gun with right hand.

SHERIFF: Go to table.

ROBBER: Aim; fire; MISSED; aim; fire; SHERIFF HIT; blow out barrel; put gun in holster.

SHERIFF: Drop gun; sheriff dies.

ROBBER: Go to corner; pick up money with right hand; go to door; go thru door; close door. CURTAIN.

A careful reading of the above scripts reveals the highly intense drama present here. The computer program was careful to keep track of the locations of each player, the contents of his hands, etc. Actions taken by the players were random, governed by certain probabilities; the probability of a foolish action was increased depending on how much that player had had to drink and on how often he had been nicked in a shot. The reader will be able to deduce further properties of the program by studying the sample scripts.

Of course, even the best scripts are rewritten before they are produced, and this is especially true when an inexperienced writer has prepared the original draft. Here are the scripts just as they were actually used in the show:

Scene #1. Music up.

MS Robber peering thru window of shack.

CU Robber's face.

MS Robber entering shack.

CU Robber sees whiskey bottle on table.

CU Sheriff outside shack.

MS Robber sees sheriff.

LS Sheriff in doorway over shoulder of robber, both draw.

MS Sheriff drawing gun.

LS Shooting it out. Robber gets shot.

MS Sheriff picking up money bags.

MS Robber staggering.

MS Robber dying. Falls across table, after trying to take last shot at sheriff.

MS Sheriff walking thru doorway with money.

MS of robber's body, now still, lying across table top. Camera dollies back. (Laughter)

Scene #2. Music up.

CU of window. Robber appears.

MS Robber entering shack with two sacks of money.

MS Robber puts money bags on barrel.

CU Robber—sees whiskey on table.

MS Robber pouring himself a drink at table. Goes to count money. Laughs.

MS Sheriff outside shack.

MS thru window.

MS Robber sees sheriff thru window.

LS Sheriff entering shack. Draw. Shoot it out.

CU Sheriff. Wrigging from shot.

M/2 shot Sheriff staggering to table for a drink . . . falls dead.

MS Robber leaves shack with money bags.*

[Note: CU = "close up", MS = "medium shot", etc. The above details were kindly furnished to the author by Thomas H. Wolf, producer of the television show, who suggested the idea of a computer-written playlet in the first place, and also by Douglas T. Ross and Harrison R. Morse who produced the computer program.]

The reader will undoubtedly have many ideas about how he could prepare his own computer program to do creative writing; and that is the point of this exercise.

- 6. [40] Look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find.

*© 1962 by Columbia Broadcasting System, Inc. All Rights Reserved. Used by permission. For further information, see J. E. Pfeiffer, *The Thinking Machine* (New York: J. B. Lippincott, 1962).

- 7. [M40] A programmer decided to encipher his files by using a linear congruential sequence $\langle X_n \rangle$ of period 2^{32} generated by (a, c, X_0) with $m = 2^{32}$. He took the most significant bits $\lfloor X_n/2^{16} \rfloor$ and exclusive-or'ed them onto his data, but kept the parameters a , c , and X_0 secret.

Show that this isn't a very secure scheme, by devising a method that deduces the multiplier a and the first difference $X_1 - X_0$ in a reasonable amount of time, given only the values of $\lfloor X_n/2^{16} \rfloor$ for $0 \leq n < 150$.

CHAPTER FOUR

ARITHMETIC

Seeing there is nothing (right well beloved Students in the Mathematickes) that is so troublesome to Mathematicall practise, nor that doth more molest and hinder Calculators, then the Multiplications, Divisions, square and cubical Extractions of great numbers, which besides the tedious expence of time are for the most part subject to many slippery errors.

I began therefore to consider in my minde, by what certaine and ready Art I might remove those hindrances.

—JOHN NAPIER (1614)

I do hate sums. There is no greater mistake than to call arithmetic an exact science. There are . . . hidden laws of number which it requires a mind like mine to perceive. For instance, if you add a sum from the bottom up, and then again from the top down, the result is always different.

—MRS. LA TOUCHE (19th century)

I cannot conceive that anybody will require multiplications at the rate of 40,000, or even 4,000 per hour; such a revolutionary change as the octonary scale should not be imposed upon mankind in general for the sake of a few individuals.

—F. H. WALES (1936)

Most numerical analysts have no interest in arithmetic.

—B. PARLETT (1979)

THE CHIEF PURPOSE of this chapter is to make a careful study of the four basic processes of arithmetic: addition, subtraction, multiplication, and division. Many people regard arithmetic as a trivial thing that children learn and computers do, but we will see that arithmetic is a fascinating topic with many interesting facets. It is important to make a thorough study of efficient methods for calculating with numbers, since arithmetic underlies so many computer applications.

Arithmetic is, in fact, a lively subject that has played an important part in the history of the world, and it still is undergoing rapid development. In this chapter, we shall analyze algorithms for doing arithmetic operations on many types of quantities, such as “floating point” numbers, extremely large numbers, fractions (rational numbers), polynomials, and power series; and we will also discuss related topics such as radix conversion, factoring of numbers, and the evaluation of polynomials.

4.1. POSITIONAL NUMBER SYSTEMS

THE WAY WE DO ARITHMETIC is intimately related to the way we represent the numbers we deal with, so it is appropriate to begin our study of the subject with a discussion of the principal means for representing numbers.

Positional notation using base b (or radix b) is defined by the rule

$$(\dots a_3 a_2 a_1 a_0.a_{-1} a_{-2} \dots)_b = \dots + a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0 + a_{-1} b^{-1} + a_{-2} b^{-2} + \dots; \quad (1)$$

for example, $(520.3)_6 = 5 \cdot 6^2 + 2 \cdot 6^1 + 0 + 3 \cdot 6^{-1} = 192\frac{1}{2}$. Our conventional decimal number system is, of course, the special case when b is ten, and when the a 's are chosen from the "decimal digits" 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; in this case the subscript b in (1) may be omitted.

The simplest generalizations of the decimal number system are obtained when we take b to be an integer greater than 1 and when we require the a 's to be integers in the range $0 \leq a_k < b$. This gives us the standard binary ($b = 2$), ternary ($b = 3$), quaternary ($b = 4$), quinary ($b = 5$), ... number systems. In general, we could take b to be any nonzero number, and we could choose the a 's from any specified set of numbers; this leads to some interesting situations, as we shall see.

The dot that appears between a_0 and a_{-1} in (1) is called the *radix point*. (When $b = 10$, it is also called the decimal point, and when $b = 2$, it is sometimes called the binary point, etc.) Continental Europeans often use a comma instead of a dot to denote the radix point; Englishmen often use a raised dot.

The a 's in (1) are called the *digits* of the representation. A digit a_k for large k is often said to be "more significant" than the digits a_k for small k ; accordingly, the leftmost or "leading" digit is referred to as the *most significant digit* and the rightmost or "trailing" digit is referred to as the *least significant digit*. In the standard binary system the binary digits are often called *bits*; in the standard hexadecimal system (radix sixteen) the hexadecimal digits zero through fifteen are usually denoted by

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.$$

The historical development of number representations is a fascinating story, since it parallels the development of civilization itself. We would be going far afield if we were to examine this history in minute detail, but it will be instructive to look at its main features here.

The earliest forms of number representations, still found in primitive cultures, are generally based on groups of fingers, piles of stones, etc., usually with special conventions about replacing a larger pile or group of, say, five or ten objects by one object of a special kind or in a special place. Such systems lead naturally to the earliest ways of representing numbers in written form, as in the systems of Babylonian, Egyptian, Greek, Chinese, and Roman numerals; but such notations are comparatively inconvenient for performing arithmetic operations except in the simplest cases.

During the twentieth century, historians of mathematics have made extensive studies of early cuneiform tablets found by archeologists in the Middle East. These studies show that the Babylonian people actually had two distinct systems of number representation: Numbers used in everyday business transactions were written in a notation based on grouping by tens, hundreds, etc.; this notation was inherited from earlier Mesopotamian civilizations, and large numbers were seldom required. When more difficult mathematical problems were considered, however, Babylonian mathematicians made extensive use of a sexagesimal (radix sixty) positional notation that was highly developed at least as early as 1750 B.C. This notation was unique in that it was actually a *floating point* form of representation with exponents omitted; the proper scale factor or power of sixty was to be supplied by the context, so that, for example, the numbers 2, 120, 7200, and $\frac{1}{30}$ were all written in an identical manner. The notation was especially convenient for multiplication and division, using auxiliary tables, since radix-point alignment had no effect on the answer. As examples of this Babylonian notation, consider the following excerpts from early tables: The square of 30 is 15 (which may also be read, "The square of $\frac{1}{2}$ is $\frac{1}{4}$ "); the reciprocal of 81 = (1 21)₆₀ is (44 26 40)₆₀; and the square of the latter is (32 55 18 31 6 40)₆₀. The Babylonians had a symbol for zero, but because of their "floating point" philosophy, it was used only within numbers, not at the right end to denote a scale factor. For the interesting story of early Babylonian mathematics, see O. Neugebauer, *The Exact Sciences in Antiquity* (Princeton, N. J.: Princeton University Press, 1952), and B. L. van der Waerden, *Science Awakening*, tr. by A. Dresden (Groningen: P. Noordhoff, 1954); see also D. E. Knuth, *CACM* 15 (1972), 671-677; 19 (1976), 108.

Fixed point positional notation was apparently first conceived by the Maya Indians in central America 2000 years ago; their radix-20 system was highly developed, especially in connection with astronomical records and calendar dates. But the Spanish conquerors destroyed nearly all of the Maya books on history and science, so we have comparatively little knowledge about how sophisticated the native Americans had become at arithmetic; special-purpose multiplication tables have been found, but no examples of division are known [cf. J. Eric S. Thompson, *Contributions to Amer. Anthropology and History* 7 (Carnegie Inst. of Washington, 1942), 37-62].

Several centuries before Christ, the Greek people employed an early form of the abacus to do their arithmetical calculations, using sand and/or pebbles on a board that had rows or columns corresponding in a natural way to our decimal system. It is perhaps surprising to us that the same positional notation was never adapted to written forms of numbers, since we are so accustomed to reckoning with the decimal system using pencil and paper; but the greater ease of calculating by abacus (since handwriting was not a common skill, and since abacus calculation makes it unnecessary to memorize addition and multiplication tables) probably made the Greeks feel it would be silly even to suggest that computing could be done better on "scratch paper." At the same time Greek astronomers did make use of a sexagesimal positional notation for fractions, which they had learned from the Babylonians.

Our decimal notation, which differs from the more ancient forms primarily because of its fixed radix point, together with its symbol for zero to mark an empty position, was developed first in India within the Hindu culture. The exact date when this notation first appeared is quite uncertain; about 600 A.D. seems to be a good guess. Hindu science was rather highly developed at that time, particularly in astronomy. The earliest known Hindu manuscripts that show this notation have numbers written backwards (with the most significant digit at the right), but soon it became standard to put the most significant digit at the left.

About 750 A.D., the Hindu principles of decimal arithmetic were brought to Persia, as several important works were translated into Arabic; a picturesque account of this development is given in a Hebrew document, which has been translated into English in *AMM* 15 (1918), 99–108. Not long after this, al-Khwârizmî wrote his Arabic textbook on the subject. (As noted in Chapter 1, our word “algorithm” comes from al-Khwârizmî’s name.) His work was translated into Latin and was a strong influence on Leonardo Pisano (Fibonacci), whose book on arithmetic (1202 A.D.) played a major role in the spreading of Hindu-Arabic numerals into Europe. It is interesting to note that the left-to-right order of writing numbers was unchanged during these two transitions, although Arabic is written from right to left while Hindu and Latin scholars generally wrote from left to right. A detailed account of the subsequent propagation of decimal numeration and arithmetic into all parts of Europe during the period from 1200 to 1600 A.D. has been given by David Eugene Smith in his *History of Mathematics* 1 (Boston: Ginn and Co., 1923), Chapters 6 and 8.

Decimal notation was applied at first only to integer numbers, not to fractions. Arabic astronomers, who required fractions in their star charts and other tables, continued to use the notation of Ptolemy (the famous Greek astronomer), a notation based on sexagesimal fractions. This system still survives today, in our trigonometric units of “degrees, minutes, and seconds,” and also in our units of time, as a remnant of the original Babylonian sexagesimal notation. Early European mathematicians also used sexagesimal fractions when dealing with noninteger numbers; for example, Fibonacci gave the value

$$1^\circ 22' 7'' 42''' 33^{IV} 4^V 40^{VI}$$

as an approximation to the root of the equation $x^3 + 2x^2 + 10x = 20$. (The correct answer is $1^\circ 22' 7'' 42''' 33^{IV} 4^V 38^{VI} 30^{VII} 50^{VIII} 15^{IX} 43^{X} \dots$.)

The use of decimal notation also for tenths, hundredths, etc., in a similar way seems to be a comparatively minor change; but, of course, it is hard to break with tradition, and sexagesimal fractions have an advantage over decimal fractions in that numbers such as $\frac{1}{3}$ can be expressed exactly, in a simple way.

Chinese mathematicians—who never used sexagesimals—were apparently the first people to work with the equivalent of decimal fractions, although their numeral system (lacking zero) was not originally a positional number system in the strict sense. Chinese units of weights and measures were decimal, so that Tsu Chhung-Chih (who died c. 500 A.D.) was able to express an approximation

to π in the following form:

3 chang, 1 chhih, 4 tshun, 1 fēn, 5 li, 9 hao, 2 miao, 7 hu.

Here chang, . . . , hu are units of length; 1 hu (the diameter of a silk thread) equals $1/10$ miao, etc. The use of such decimal-like fractions was fairly widespread in China after about 1250 A.D.

The first known appearance of decimal fractions in a true positional system occurs in a 10th-century arithmetic text written in Damascus by an obscure mathematician named al-Uqlīdisī ("the Euclidean"). He used the symbol ' for a decimal point, for example in connection with a problem about compound interest, the computation of 135 times $(1.1)^n$ for $1 \leq n \leq 5$. [See A. S. Saidan, *The Arithmetic of al-Uqlīdisī* (Dordrecht: D. Reidel, 1975), 110, 114, 343, 355, 481–485.] But he did not develop the idea very fully, and his trick was soon forgotten; five centuries passed before decimal fractions were reinvented by a Persian mathematician, al-Kashī, who died c. 1436. Al-Kashī was a highly skillful calculator, who gave the value of 2π as follows, correct to 16 decimal places:

integer	fractions																
0	6	2	8	3	1	8	5	3	0	7	1	7	9	5	8	6	5

This was by far the best approximation to π known until Ludolph van Ceulen laboriously calculated 35 decimal places during the period 1596–1610.

The earliest known example of decimal fractions in Europe occurs in a 15th-century text where, for example, 153.5 is multiplied by 16.25 to get 2494.375; this was referred to as a "Turkish method." In 1525, Christof Rudolff of Germany discovered decimal fractions for himself; but like al-Uqlīdisī, his work seems to have had little influence. François Viète suggested the idea again in 1579. Finally, an arithmetic text by Simon Stevin of Belgium, who independently hit on the idea of decimal fractions in 1585, became popular. Stevin's work, and the discovery of logarithms soon afterwards, made decimal fractions commonplace in Europe during the 17th century. [See D. E. Smith, *History of Mathematics 2* (Boston: Ginn and Co., 1925), 228–247, and C. B. Boyer, *History of Mathematics* (New York: Wiley, 1968), for further remarks and references.]

The binary system of notation has its own interesting history. Many primitive tribes in existence today are known to use a binary or "pair" system of counting (making groups of two instead of five or ten), but they do not count in a true radix-2 system, since they do not treat powers of 2 in a special manner. See *The Diffusion of Counting Practices* by Abraham Seidenberg, *Univ. Calif. Publ. in Math.* 3 (1960), 215–300, for interesting details about primitive number systems. Another "primitive" example of an essentially binary system is the conventional musical notation for expressing rhythms and durations of time.

Nondecimal number systems were discussed in Europe during the seventeenth century. For many years astronomers had occasionally used sexagesimal arithmetic both for the integer and the fractional parts of numbers, primarily when performing multiplication [see John Wallis, *Treatise of Algebra* (Oxford,

1685), 18–22, 30]. The fact that any integer greater than 1 could serve as radix was apparently first stated in print by Blaise Pascal in *De numeris multiplicibus*, which was written about 1658 [see Pascal's *Oeuvres Complètes* (Paris: Éditions de Seuil, 1963), 84–89]. Pascal wrote, “Denaria enim ex institute hominum, non ex necessitate naturæ ut vulgus arbitratur, et sane satis inepte, posita est”; i.e., “The decimal system has been established, somewhat foolishly to be sure, according to man’s custom, not from a natural necessity as most people would think.” He stated that the duodecimal (radix twelve) system would be a welcome change, and he gave a rule for testing a duodecimal number for divisibility by nine. Erhard Weigel tried to drum up enthusiasm for the quaternary (radix four) system in a series of publications beginning in 1673. A detailed discussion of radix-twelve arithmetic was given by Joshua Jordaine, *Duodecimal Arithmetick* (London, 1687).

Although decimal notation was almost exclusively used for arithmetic during that era, other systems of weights and measures were rarely if ever based on multiples of 10, and many business transactions required a good deal of skill in adding quantities such as pounds, shillings, and pence. For centuries merchants had therefore learned to compute sums and differences of quantities expressed in peculiar units of currency, weights, and measures; and this was actually arithmetic in a nondecimal number system. The common units of liquid measure in England, dating from the 13th century or earlier, are particularly noteworthy:

2 gills = 1 chopin	2 demibushels = 1 bushel or firkin
2 chopins = 1 pint	2 firkins = 1 kilderkin
2 pints = 1 quart	2 kilderkins = 1 barrel
2 quarts = 1 pottle	2 barrels = 1 hogshead
2 pottles = 1 gallon	2 hogsheads = 1 pipe
2 gallons = 1 peck	2 pipes = 1 tun
2 pecks = 1 demibushel	

Quantities of liquid expressed in gallons, pottles, quarts, pints, etc. were essentially written in binary notation. Perhaps the true inventors of binary arithmetic were English wine merchants!

The first known appearance of pure binary notation was about 1605 in some unpublished manuscripts of Thomas Harriot (1560–1621). Harriot was a creative man, who first became famous by coming to America as a representative of Sir Walter Raleigh. He invented (among other things) a notation like that now used for “less than” and “greater than” relations; but for some reason he chose not to publish many of his discoveries. Excerpts from his notes on binary arithmetic have been reproduced by John W. Shirley, *Amer. J. Physics* **19** (1951), 452–454. The first published discussion of the binary system was given in a comparatively little-known work by a Spanish bishop, Juan Caramuel Lobkowitz, *Mathesis biceps* **1** (Campaniae, 1670), 45–48; Caramuel discussed the representation of numbers in radices 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, and 60 at some length, but gave no examples of arithmetic operations in nondecimal systems (except for the trivial operation of adding unity).

Ultimately, an article by G. W. Leibniz [*Memoires de l'Academie Royale des Sciences* (Paris: 1703), 110–116], which illustrated binary addition, subtraction, multiplication, and division, really brought binary notation into the limelight, and this article is usually referred to as the birth of radix-2 arithmetic. Leibniz later referred to the binary system quite frequently. He did not recommend it for practical calculations, but he stressed its importance in number-theoretical investigations, since patterns in number sequences are often more apparent in binary notation than they are in decimal; he also saw a mystical significance in the fact that everything is expressible in terms of zero and one. Leibniz's unpublished manuscripts show that he had been interested in binary notation as early as 1679, when he referred to it as a "bimal" system (analogous to "decimal").

A careful study of Leibniz's early work with binary numbers has been made by Hans J. Zacher, *Die Hauptschriften zur Dyadik von G. W. Leibniz* (Frankfurt am Main: Klostermann, 1973). Zacher points out that Leibniz was familiar with John Napier's so-called "local arithmetic," a way for calculating with stones that amounts to using a radix-2 abacus. [Napier had published the idea of local arithmetic as an appendix to his little book *Rhabdologia* in 1617; it may be called the world's first "binary computer," and it is surely the world's cheapest, although Napier felt that it was more amusing than practical. See Martin Gardner's discussion in *Scientific American* 228 (April 1973), 106–111.]

It is interesting to note that the important concept of negative powers to the right of the radix point was not yet well understood at that time. Leibniz asked James Bernoulli to calculate π in the binary system, and Bernoulli "solved" the problem by taking a 35-digit approximation to π , multiplying it by 10^{35} , and then expressing this integer in the binary system as his answer. On a smaller scale this would be like saying that $\pi \approx 3.14$, and $(314)_{10} = (100111010)_2$; hence π in binary is $100111010!$ [See Leibniz, *Math. Schriften*, ed. by K. Gehrhardt, 3 (Halle: 1855), 97; two of the 118 bits in the answer are incorrect, due to computational errors.] The motive for Bernoulli's calculation was apparently to see whether any simple pattern could be observed in this representation of π .

Charles XII of Sweden, whose talent for mathematics perhaps exceeded that of all other kings in the history of the world, hit on the idea of radix-8 arithmetic about 1717. This was probably his own invention, although he had met Leibniz briefly in 1707. Charles felt that radix 8 or 64 would be more convenient for calculation than the decimal system, and he considered introducing octal arithmetic into Sweden; but he died in battle before decreeing such a change. [See *The Works of Voltaire* 21 (Paris: E. R. DuMont, 1901), 49; E. Swedenborg, *Gentleman's Magazine* 24 (1754), 423–424.]

Octal notation was proposed also in colonial America before 1750, by the Rev. Hugh Jones, rector of a parish in Maryland [cf. *Gentleman's Magazine* 15 (1745), 377–379; H. R. Phalen, *AMM* 56 (1949), 461–465].

More than a century later, a prominent Swedish-American civil engineer named John W. Nystrom decided to carry Charles XII's plans a step further, by devising a complete system of numeration, weights, and measures based on radix-16 arithmetic. He wrote, "I am not afraid, or do not hesitate, to advocate a

binary system of arithmetic and metrology. I know I have nature on my side; if I do not succeed to impress upon you its utility and great importance to mankind, it will reflect that much less credit upon our generation, upon our scientific men and philosophers." Nystrom devised special means for pronouncing hexadecimal numbers; e.g., $(B0160)_{16}$ was to be read "vybong, bysanton." His entire system was called the Tonal System, and it is described in *J. Franklin Inst.* **46** (1863), 263–275, 337–348, 402–407. A similar system, but using radix 8, was worked out by Alfred B. Taylor [Proc. Amer. Pharmaceutical Assoc. 8 (1859), 115–216; Proc. Amer. Philosophical Soc. **24** (1887), 296–366]. Increased use of the French (metric) system of weights and measures prompted extensive debate about the merits of decimal arithmetic during that era; indeed, octal arithmetic was even being proposed in France [J. D. Colenne, *Le système octaval* (Paris: 1845); Aimé Mariage, *Numération par huit* (Paris: Le Nonnant, 1857)].

The binary system was well known as a curiosity ever since Leibniz's time, and about 20 early references to it have been compiled by R. C. Archibald [AMM **25** (1918), 139–142]. It was applied chiefly to the calculation of powers, as explained in Section 4.6.3, and to the analysis of certain games and puzzles. Giuseppe Peano [*Atti della R. Accademia delle Scienze di Torino* **34** (1898), 47–55] used binary notation as the basis of a "logical" character set of 256 symbols. Joseph Bowden [*Special Topics in Theoretical Arithmetic* (Garden City: 1936), 49] gave his own system of nomenclature for hexadecimal numbers.

The book *History of Binary and Other Nondecimal Numeration* by Anton Glaser (privately printed, 1971) contains an informative and nearly complete discussion of the development of binary notation, including English translations of many of the works cited above.

Much of the recent history of number systems is connected with the development of calculating machines. Charles Babbage's notebooks for 1838 show that he considered using nondecimal numbers in his Analytical Engine [cf. M. V. Wilkes, *Historia Math.* **4** (1977), 421]. Increased interest in mechanical devices for arithmetic, especially for multiplication, led several people in the 1930s to consider the binary system for this purpose. A particularly delightful account of such activity appears in the article "Binary Calculation" by E. William Phillips [*Journal of the Institute of Actuaries* **67** (1936), 187–221] together with a record of the discussion that followed a lecture he gave on the subject. Phillips began by saying, "The ultimate aim [of this paper] is to persuade the whole civilized world to abandon decimal numeration and to use octonal [i.e., radix 8] numeration in its place."

Modern readers of Phillips's article will perhaps be surprised to discover that a radix-8 number system was properly referred to as "octonary" or "octonal," according to all dictionaries of the English language at that time, just as the radix-10 number system is properly called either "denary" or "decimal"; the word "octal" did not appear in English language dictionaries until 1961, and it apparently originated as a term for the "base" of a certain class of vacuum tubes. The word "hexadecimal," which has crept into our language even more recently, is a mixture of Greek and Latin stems; more proper terms would be

"senidenary" or "sedecimal" or even "sexadecimal," but the latter is perhaps too risqué for computer programmers.

The comment by Mr. Wales that is quoted at the beginning of this chapter has been taken from the discussion printed with Phillips's paper. Another man who attended the same lecture objected to the octal system for business purposes: "5% becomes 3.1463 per 64, which sounds rather horrible."

Phillips got the inspiration for his proposals from an electronic circuit that was capable of counting in binary [C. E. Wynn-Williams, Proc. Roy. Soc. London A136 (1932), 312–324]. Electromechanical and electronic circuitry for general arithmetic operations was developed during the late 1930s, notably by John V. Atanasoff and George R. Stibitz in the U.S.A., L. Couffignal and R. Valtat in France, Helmut Schreyer and Konrad Zuse in Germany. All of these inventors used the binary system, although Stibitz later developed excess-3 binary-coded-decimal notation. A fascinating account of these early developments, including reprints and translations of important contemporary documents, appears in Brian Randell's book *The Origins of Digital Computers* (Berlin: Springer, 1973).

The first American high-speed computers, built in the early 1940s, used decimal arithmetic. But in 1946, an important memorandum by A. W. Burks, H. H. Goldstine, and J. von Neumann, in connection with the design of the first stored-program computers, gave detailed reasons for the decision to make a radical departure from tradition and to use base-two notation [see John von Neumann, *Collected Works* 5, 41–65]. Since then binary computers have multiplied. After a dozen years of experience with binary machines, a discussion of the relative advantages and disadvantages of radix-2 notation was given by W. Buchholz in his paper "Fingers or Fists?" [CACM 2 (December 1959), 3–11].

The MIX computer used in this book has been defined so that it can be either binary or decimal. It is interesting to note that nearly all MIX programs can be expressed without knowing whether binary or decimal notation is being used—even when we are doing calculations involving multiple-precision arithmetic. Thus we find that the choice of radix does not significantly influence computer programming. (Noteworthy exceptions to this statement, however, are the "Boolean" algorithms discussed in Section 7.1; see also Algorithm 4.5.2B.)

There are several different methods for representing negative numbers in a computer, and this sometimes influences the way arithmetic is done. In order to understand these other notations, let us first consider MIX as if it were a decimal computer; then each word contains 10 digits and a sign, for example

$$-12345\ 67890. \quad (2)$$

This is called the *signed-magnitude* representation. Such a representation agrees with common notational conventions, so it is preferred by many programmers. A potential disadvantage is that minus zero and plus zero can both be represented, while they usually should mean the same number; this possibility requires some care in practice, although it turns out to be useful at times.

Most mechanical calculators that do decimal arithmetic use another system called *ten's complement* notation. If we subtract 1 from 00000 00000, we get

99999 99999 in this notation; in other words, no explicit sign is attached to the number, and calculation is done *modulo* 10^{10} . The number —12345 67890 would appear as

$$87654\ 32110 \quad (3)$$

in ten's complement notation. It is conventional to regard any number whose leading digit is 5, 6, 7, 8, or 9 as a negative value in this notation, although with respect to addition and subtraction there is no harm in regarding (3) as the number +87654 32110 if it is convenient to do so. Note that there is no problem of "minus zero" in such a system.

The major difference between signed magnitude and ten's complement notations in practice is that shifting right does not divide the magnitude by ten; for example, the number —11 = ... 99989, shifted right one, gives ... 99998 = —2 (assuming that a shift to the right inserts "9" as the leading digit when the number shifted is negative). In general, x shifted right one digit in ten's complement notation will give $\lfloor x/10 \rfloor$, whether x is positive or negative.

A possible disadvantage of the ten's complement system is the fact that it is not symmetric about zero; the largest negative number representable in p digits is 500...0, and it is not the negative of any p -digit positive number. Thus it is possible that changing x to $-x$ will cause overflow. (See exercises 7 and 31 for a discussion of radix-complement notation with *infinite* precision.)

Another notation that has been used since the earliest days of high-speed computers is called *nines' complement* representation. In this case the number —12345 67890 would appear as

$$87654\ 32109. \quad (4)$$

Each digit of a negative number ($-x$) is equal to 9 minus the corresponding digit of x . It is not difficult to see that the nines' complement notation for a negative number is always one less than the corresponding ten's complement notation. Addition and subtraction are done modulo $10^{10} - 1$, which means that a carry off the left end is to be added at the right end. (Cf. Section 3.2.1.1.) Again there is a potential problem with minus zero, since 99999 99999 and 00000 00000 denote the same value.

The ideas just explained for radix 10 arithmetic apply in a similar way to radix 2 arithmetic, where we have *signed magnitude*, *two's complement*, and *ones' complement* notations. The MIX computer, as used in the examples of this chapter, deals only with signed-magnitude arithmetic; however, alternative procedures for complement notations are discussed in the accompanying text when it is important to do so.

Most computer manuals tell us that the machine's circuitry assumes that the radix point is situated in a particular place within each computer word. This advice should usually be disregarded. It is better to learn the rules concerning where the radix point will appear in the result of an instruction if we assume that it lies in a certain place beforehand. For example, in the case of MIX we could regard our operands either as integers with the radix point at the extreme

right, or as fractions with the radix point at the extreme left, or as some mixture of these two extremes; the rules for the appearance of the radix point in each result are straightforward.

It is easy to see that there is a simple relation between radix b and radix b^k :

$$(\dots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} \dots)_b = (\dots A_3 A_2 A_1 A_0 . A_{-1} A_{-2} \dots)_{b^k}, \quad (5)$$

where

$$A_j = (a_{kj+k-1} \dots a_{kj+1} a_{kj})_b;$$

see exercise 8. Thus we have simple techniques for converting at sight between, say, binary and octal notation.

Many interesting variations on positional number systems are possible besides the standard b -ary systems discussed so far. For example, we might have numbers in base (-10) , so that

$$\begin{aligned} & (\dots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} \dots)_{-10} \\ &= \dots + a_3(-10)^3 + a_2(-10)^2 + a_1(-10)^1 + a_0 + \dots \\ &= \dots - 1000a_3 + 100a_2 - 10a_1 + a_0 - \frac{1}{10}a_{-1} + \frac{1}{100}a_{-2} - \dots \end{aligned}$$

Here the individual digits satisfy $0 \leq a_k \leq 9$ just as in the decimal system. The number 12345 67890 appears in the "negadecimal" system as

$$(1\ 93755\ 73910)_{-10}, \quad (6)$$

since the latter represents 10305070900 — 9070503010. It is interesting to note that the negative of this number, —12345 67890, would be written

$$(28466\ 48290)_{-10}, \quad (7)$$

and, in fact, every *real* number whether positive or negative can be represented without a sign in the -10 system.

Negative-base systems were first considered by Vittorio Grünwald [Giornale di matematiche di Battaglini 23 (1885), 203–221, 367], who explained how to perform the four arithmetic operations in such systems; Grünwald also discussed root extraction, divisibility tests, and radix conversion. However, since his work was published in a rather obscure journal, it seems to have had no effect on other research, and it was soon forgotten. The next publication about negative-base systems was apparently by A. J. Kempner [AMM 43 (1936), 610–617], who discussed the properties of non-integer radices and remarked in a footnote that negative radices would be feasible too. After twenty more years the idea was rediscovered again, this time by Z. Pawlak and A. Wakulicz [Bulletin de l'Academie Polonaise des Sciences, Classe III, 5 (1957), 233–236; Série des sciences techniques 7 (1959), 713–721], and also by L. Wadel [IRE Transactions EC-6

(1957), 123]. For further references see *IEEE Transactions EC-12* (1963), 274–276; *Computer Design* 6 (May 1967), 52–63. There is evidence that the idea of negative bases occurred independently to quite a few people. For example, D. E. Knuth had discussed negative-base systems in 1955, together with a further generalization to complex-valued bases, in a short paper submitted to a “science talent search” contest for high-school seniors.

The base $2i$ gives rise to a system called the “quater-imaginary” number system (by analogy with “quaternary”), which has the unusual feature that every complex number can be represented with the digits 0, 1, 2, and 3 without a sign. [See D. E. Knuth, *CACM* 3 (1960), 245–247.] For example,

$$\begin{aligned}(11210.31)_{2i} &= 1 \cdot 16 + 1 \cdot (-8i) + 2 \cdot (-4) + 1 \cdot (2i) + 3 \cdot (-\frac{1}{2}i) + 1(-\frac{1}{4}) \\ &= 7\frac{3}{4} - 7\frac{1}{2}i.\end{aligned}$$

Here the number $(a_{2n} \dots a_1 a_0. a_{-1} \dots a_{-2k})_{2i}$ is equal to

$$(a_{2n} \dots a_2 a_0. a_{-2} \dots a_{-2k})_{-4} + 2i(a_{2n-1} \dots a_3 a_1. a_{-1} \dots a_{-2k+1})_{-4},$$

so conversion to and from quater-imaginary notation reduces to conversion to and from negative quaternary representation of the real and imaginary parts. The interesting property of this system is that it allows multiplication and division of complex numbers to be done in a fairly unified manner without treating real and imaginary parts separately. For example, we can multiply two numbers in this system much as we do with any base, merely using a different “carry” rule: whenever a digit exceeds 3 we subtract 4 and “carry” -1 two columns to the left; when a digit is negative, we add 4 to it and “carry” $+1$ two columns to the left. A study of the following example shows this peculiar carry rule at work:

$$\begin{array}{rcl} 1 & 2 & 2 & 3 & 1 & [9 - 10i] \\ 1 & 2 & 2 & 3 & 1 & [9 - 10i] \\ \hline 1 & 2 & 2 & 3 & 1 \\ 1 & 0 & 3 & 2 & 0 & 2 & 1 & 3 \\ & 1 & 3 & 0 & 2 & 2 \\ & 1 & 3 & 0 & 2 & 2 \\ 1 & 2 & 2 & 3 & 1 \\ \hline 0 & 2 & 1 & 3 & 3 & 3 & 1 & 2 & 1 & [-19 - 180i] \end{array}$$

A similar system that uses just the digits 0 and 1 may be based on $\sqrt{2}i$, but this requires an infinite nonrepeating expansion for the simple number “ i ” itself. Vittorio Grünwald proposed using the digits 0 and $1/\sqrt{2}$ in odd-numbered positions, to avoid such a problem, but this actually spoils the whole system [cf. *Commentari dell’ Ateneo di Brescia* (1886), 43–54].

Another “binary” complex number system may be obtained by using the base $i - 1$, as suggested by W. Penney [*JACM* 12 (1965), 247–248]:

$$\begin{aligned}(\dots a_4 a_3 a_2 a_1 a_0. a_{-1} \dots)_{i-1} \\ = \dots - 4a_4 + (2+2i)a_3 - 2ia_2 + (i-1)a_1 + a_0 - \frac{1}{2}(i+1)a_{-1} + \dots\end{aligned}$$

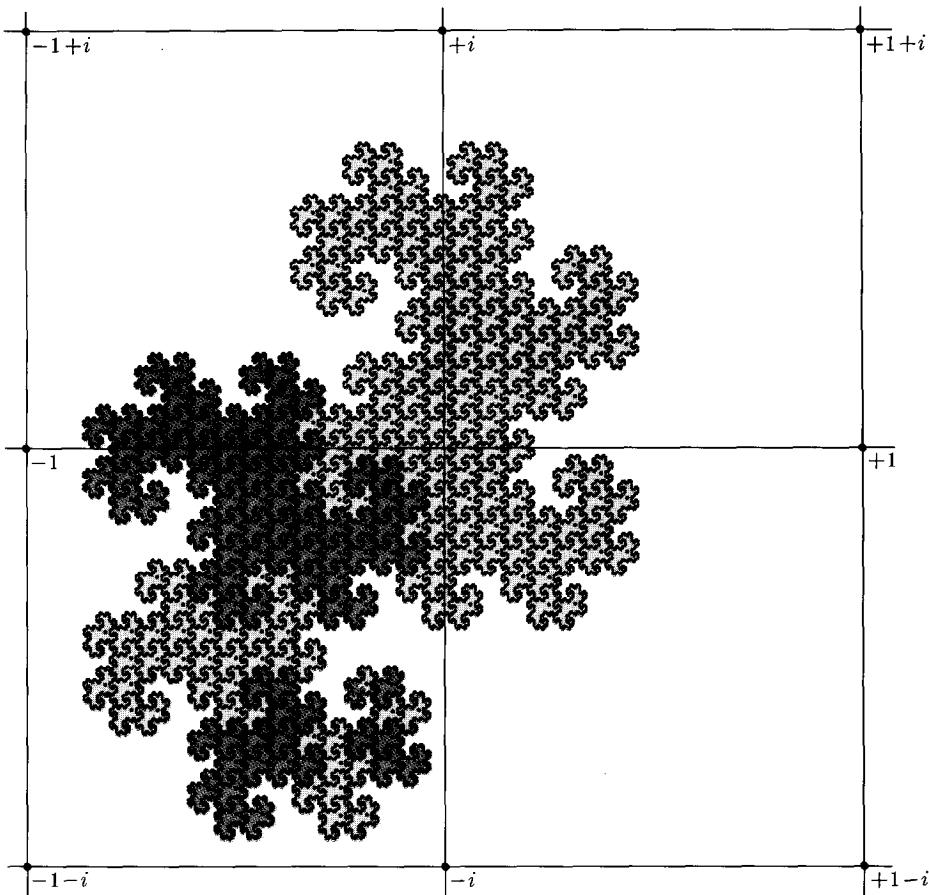


Fig. 1. The set S . (Illustration by P. M. Farmwald, R. W. Gosper, and R. E. Maas.)

In this system, only the digits 0 and 1 are needed. One way to demonstrate that every complex number has such a representation is to consider the interesting set S shown in Fig. 1; this set is, by definition, all points that can be written as $\sum_{k \geq 1} a_k(i-1)^{-k}$, for an infinite sequence a_1, a_2, a_3, \dots of zeros and ones. Figure 1 shows that S can be decomposed into 256 pieces congruent to $\frac{1}{16}S$; note that if the diagram of S is rotated counterclockwise by 135° , we obtain two adjacent sets congruent to $(1/\sqrt{2})S$ (since $(i-1)S = S \cup (S+1)$). For details of a proof that S contains all complex numbers that are of sufficiently small magnitude, see exercise 18.

Perhaps the prettiest number system of all is the balanced ternary notation, which consists of base-3 representation using -1 , 0 , and $+1$ as “trits” (ternary digits) instead of 0 , 1 , and 2 . If we use the symbol $\bar{1}$ to stand for -1 , we have the following examples of balanced ternary numbers:

Balanced ternary	Decimal
1 0 $\bar{1}$	8
1 1 $\bar{1}$ 0.1 $\bar{1}$	$32\frac{5}{9}$
$\bar{1}$ 1 1 0.1 1	$-32\frac{5}{9}$
$\bar{1}$ 1 1 0	-33
0.1 1 1 1 1 ...	$\frac{1}{2}$

One way to find the representation of a number in the balanced ternary system is to start by representing it in ternary notation; for example,

$$208.3 = (21201.022002200220\dots)_3.$$

(A very simple pencil-and-paper method for converting to ternary notation is given in exercise 4.4-12.) Now add the infinite number ...11111.11111... in ternary notation; we obtain, in the above example, the infinite number

$$(\dots 11111210012.210121012101\dots)_3.$$

Finally, subtract ...11111.11111... by decrementing each digit; we get

$$208.3 = (10\bar{1}\bar{1}01.10\bar{1}010\bar{1}010\bar{1}0\dots)_3. \quad (8)$$

This process may clearly be made rigorous if we replace the artificial infinite number ...11111.11111... by a number with suitably many ones.

The balanced ternary number system has many pleasant properties:

- a) The negative of a number is obtained by interchanging 1 and $\bar{1}$.
- b) The sign of a number is given by its most significant nonzero “trit,” and in general we can compare any two numbers by reading them from left to right and using lexicographic order, as in the decimal system.
- c) The operation of rounding to the nearest integer is identical to truncation (i.e., deleting everything to the right of the radix point).

Addition in the balanced ternary system is quite simple, using the table

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	1	1	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0	1	1	1	$\bar{1}$	$\bar{1}$	0	0
$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1
10	1	$\bar{1}$	1	$\bar{1}$	1	1	0	$\bar{1}$	0	1	$\bar{1}$	1	1	1	0	1	0	1	1	10

(The three inputs to the addition are the digits of the numbers to be added and the carry digits.) Subtraction is negation followed by addition; and multiplication also reduces to negation and addition, as in the following example:

$$\begin{array}{r}
 1 \bar{1} 0 \bar{1} \qquad [17] \\
 1 \bar{1} 0 \bar{1} \qquad [17] \\
 \hline
 \bar{1} 1 0 1
 \end{array}$$

$$\begin{array}{r}
 \bar{1} 1 0 1 0 \\
 1 \bar{1} 0 \bar{1} \\
 \hline
 0 1 1 \bar{1} \bar{1} 0 1 \qquad [289]
 \end{array}$$

Representation of numbers in the balanced ternary system is implicitly present in a famous mathematical puzzle, which is commonly called "Bachet's problem of weights" although it was already stated by Fibonacci four centuries before Bachet wrote his book. [See W. Ahrens, *Mathematische Unterhaltungen und Spiele 1* (Leipzig: Teubner, 1910), Section 3.4.] Positional number systems with negative digits have apparently been known for more than 1000 years in India; see J. Bharati, *Vedic Mathematics* (Delhi: Motilal Banarsi Dass, 1965). They were independently rediscovered by J. Colson [*Philos. Trans.* **34** (1726), 161–173], and by Sir John Leslie [*The Philosophy of Arithmetic* (Edinburgh, 1817); see pp. 33–34, 54, 64–65, 117, 150]; and also by A. Cauchy [*Comptes Rendus* **11** (Paris: 1840), 789–798], who pointed out that negative digits make it unnecessary for a person to memorize the multiplication table past 5×5 . The first true appearance of "pure" balanced ternary notation was in an article by Léon Lalanne [*Comptes Rendus* **11** (Paris: 1840), 903–905], who was a designer of mechanical devices for arithmetic. The system was mentioned only rarely for 100 years after Lalanne's paper, until the development of the first electronic computers at the Moore School of Electrical Engineering in 1945–1946; at that time it was given serious consideration along with the binary system as a possible replacement for the decimal system. The complexity of arithmetic circuitry for balanced ternary arithmetic is not much greater than it is for the binary system, and a given number requires only $\ln 2 / \ln 3 \approx 63\%$ as many digit positions for its representation. Discussions of the balanced ternary number system appear in *AMM* **57** (1950), 90–93, and in *High-speed Computing Devices*, Engineering Research Associates (McGraw-Hill, 1950), 287–289. The experimental Russian computer SETUN was based on balanced ternary notation [see *CACM* **3** (1960), 149–150], and perhaps the symmetric properties and simple arithmetic of this number system will prove to be quite important some day—when the "flip-flop" is replaced by a "flip-flap-flap".

Positional notation generalizes in another important way to a *mixed-radix* system. Given a sequence of numbers $\langle b_n \rangle$ (where n may be negative), we define

$$\begin{aligned} & [\dots, a_3, a_2, a_1, a_0; a_{-1}, a_{-2}, \dots] \\ & [\dots, b_3, b_2, b_1, b_0; b_{-1}, b_{-2}, \dots] \end{aligned} \quad (9)$$

$$= \dots + a_3 b_2 b_1 b_0 + a_2 b_1 b_0 + a_1 b_0 + a_0 + a_{-1}/b_{-1} + a_{-2}/b_{-1} b_{-2} + \dots$$

In the simplest mixed-radix systems, we work only with integers; we let b_0, b_1, b_2, \dots be integers greater than one, and deal only with numbers that have no radix point, where a_n is required to lie in the range $0 \leq a_n < b_n$.

One of the most important mixed-radix systems is the *factorial number system*, where $b_n = n + 2$. Using this system, we can represent every positive integer uniquely in the form

$$c_n n! + c_{n-1} (n-1)! + \dots + c_2 2! + c_1, \quad (10)$$

where $0 \leq c_k \leq k$ for $1 \leq k \leq n$, and $c_n \neq 0$. (See Algorithm 3.3.2P.)

Mixed-radix systems are familiar in everyday life, when we deal with units of measure. For example, the quantity “3 weeks, 2 days, 9 hours, 22 minutes, 57 seconds, and 492 milliseconds” is equal to

$$\left[\begin{matrix} 3, 2, & 9, 22, 57; & 492 \\ & 7, 24, 60, 60; & 1000 \end{matrix} \right] \text{seconds.}$$

The quantity “10 pounds, 6 shillings, and thruppence ha’penny” was once equal to $\left[\begin{smallmatrix} 10, & 6, & 3; & 1 \\ 20, & 12, & 2; & 2 \end{smallmatrix} \right]$ pence in British currency, before Great Britain changed to a purely decimal monetary system.

It is possible to add and subtract mixed-radix numbers by using a straightforward generalization of the usual addition and subtraction algorithms, provided of course that the same mixed-radix system is being used for both operands (see exercise 4.3.1–9). Similarly, we can easily multiply or divide a mixed-radix number by small integer constants, using simple extensions of the familiar pencil-and-paper methods.

Mixed-radix systems were first discussed in full generality by Georg Cantor [*Zeitschrift für Math. und Physik* 14 (1869), 121–128]. Exercises 26 and 29 give further information about them.

Some questions concerning *irrational* radices have been investigated by W. Parry, *Acta Mathematica*, Acad. Sci. Hung., 11 (1960), 401–416.

Besides the systems described in this section, several other ways to represent numbers are mentioned elsewhere in this series of books: the binomial number system (exercise 1.2.6–56); the Fibonacci number system (exercises 1.2.8–34, 5.4.2–10); the phi number system (exercise 1.2.8–35); modular representations (Section 4.3.2); Gray code (Section 7.2.1); and roman numerals (Section 9.1).

EXERCISES

1. [15] Express $-10, -9, \dots, 9, 10$ in the number system whose base is -2 .
- 2. [24] Consider the following four number systems: (a) binary (signed magnitude); (b) negabinary (radix -2); (c) balanced ternary; and (d) radix $b = \frac{1}{10}$. Use each of these four number systems to express each of the following three numbers: (i) -49 ; (ii) $-3\frac{1}{7}$ (show the repeating cycle); (iii) π (to a few significant figures).
3. [20] Express $-49 + i$ in the quater-imaginary system.
4. [15] Assume that we have a MIX program in which location A contains a number for which the radix point lies between bytes 3 and 4, while location B contains a number whose radix point lies between bytes 2 and 3. (The leftmost byte is number 1). Where will the radix point be, in registers A and X, after the following instructions?

a) LDA A
MUL B ■

b) LDA A
SRAX 5
DIV B ■

5. [00] Explain why a negative integer in nines' complement notation has a representation in ten's complement notation that is always one greater, if the representations are regarded as positive.

6. [16] What are the largest and smallest p -bit integers that can be represented in (a) signed-magnitude binary notation (including one bit for the sign), (b) two's complement notation, (c) ones' complement notation?

7. [M20] The text defines ten's complement notation only for integers represented in a single computer word. Is there a way to define a ten's complement notation for all real numbers, having "infinite precision," analogous to the text's definition? Is there a similar way to define a nines' complement notation for all real numbers?

8. [M10] Prove Eq. (5).

► 9. [15] Change the following octal numbers to hexadecimal notation, using the hexadecimal digits 0, 1, ..., F: 12; 5655; 2550276; 76545336; 3726755.

10. [M22] Generalize Eq. (5) to mixed-radix notation.

11. [22] Design an algorithm that uses the -2 number system to compute the sum of $(a_n \dots a_1 a_0)_{-2}$ and $(b_n \dots b_1 b_0)_{-2}$, obtaining the answer $(c_{n+2} \dots c_1 c_0)_{-2}$.

12. [23] Specify algorithms that convert (a) the binary signed magnitude number $\pm(a_n \dots a_0)_2$ to its negabinary form $(b_{n+1} \dots b_0)_{-2}$; and (b) the negabinary number $(b_{n+1} \dots b_0)_{-2}$ to its signed magnitude form $\pm(a_{n+1} \dots a_0)_2$.

► 13. [M21] In the decimal system there are some numbers with two infinite decimal expansions; e.g., $2.3599999\dots = 2.3600000\dots$. Does the negadecimal (base -10) system have unique expansions, or are there real numbers with two different infinite expansions in this base also?

14. [14] Multiply $(11321)_2$ by itself in the quater-imaginary system using the method illustrated in the text.

15. [M24] What are the sets

$$S = \left\{ \sum_{k \geq 1} a_k b^{-k} \mid a_k \text{ an allowable digit} \right\},$$

analogous to Fig. 1, for the negative decimal and for the quater-imaginary number systems?

16. [M24] Design an algorithm to add 1 to $(a_n \dots a_1 a_0)_{i-1}$ in the $i-1$ number system.

17. [M30] It may seem peculiar that $i-1$ has been suggested as a number-system base, instead of the similar but intuitively simpler number $i+1$. Can every complex number $a+bi$, where a and b are integers, be represented in a positional number system to base $i+1$, using only the digits 0 and 1?

18. [HM32] Show that the set S of Fig. 1 is a closed set that contains a neighborhood of the origin. (Consequently, every complex number has a "binary" representation to base $i-1$.)

► 19. [23] (David W. Matula.) Let D be a set of b integers, containing exactly one solution to the congruence $x \equiv j \pmod{b}$ for $0 \leq j < b$. Prove that all integers m (positive, negative, or zero) can be represented in the form $m = (a_n \dots a_0)_b$, where all the a_j are in D , if and only if all integers in the range $l \leq m \leq u$ can be so represented, where $l = -\max\{a \mid a \in D\}/(b-1)$, $u = -\min\{a \mid a \in D\}/(b-1)$. For example, $D = \{-1, 0, \dots, b-2\}$ satisfies the conditions for all $b \geq 3$. [Hint: Design an algorithm that constructs a suitable representation.]

- 20.** [HM28] (David W. Matula.) Consider a decimal number system that uses the digits $D = \{-1, 0, 8, 17, 26, 35, 44, 53, 62, 71\}$ instead of $\{0, 1, \dots, 9\}$. The result of exercise 19 implies (as in exercise 18) that all real numbers have an infinite decimal expansion using digits from D .

In the usual decimal system, exercise 13 points out that some numbers have two representations. (a) Find a real number that has more than two D -decimal representations. (b) Show that no real number has infinitely many D -decimal representations. (c) Show that uncountably many numbers have two or more D -decimal representations.

- **21.** [M22] (C. E. Shannon.) Can every real number (positive, negative, or zero) be expressed in a “balanced decimal” system, i.e., in the form $\sum_{k \leq n} a_k 10^k$, for some integer n and some sequence $a_n, a_{n-1}, a_{n-2}, \dots$, where each a_k is one of the ten numbers $\{-4\frac{1}{2}, -3\frac{1}{2}, -2\frac{1}{2}, -1\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, 1\frac{1}{2}, 2\frac{1}{2}, 3\frac{1}{2}, 4\frac{1}{2}\}$? (Note that zero is not one of the allowed digits, but we implicitly assume that a_{n+1}, a_{n+2}, \dots are zero.) Find all representations of zero in this number system, and find all representations of unity.

- 22.** [HM25] Let $\alpha = -\sum_{m \geq 1} 10^{-m^2}$. Given $\epsilon > 0$ and any real number x , prove that there is a “decimal” representation such that $0 < |x - \sum_{0 \leq k \leq n} a_k 10^k| < \epsilon$, where each a_k is allowed to be only one of the three values 0, 1, or α . (Note that no negative powers of 10 are used in this representation!)

- 23.** [HM90] Let D be a set of b real numbers such that every positive real number has a representation $\sum_{k \leq n} a_k b^k$ with all $a_k \in D$. Exercise 20 shows that there may be many numbers without unique representations; but prove that the set T of all such numbers has measure zero.

- 24.** [M35] Find infinitely many different sets D of ten nonnegative integers satisfying the following three conditions: (i) $\gcd(D) = 1$; (ii) $0 \in D$; (iii) every positive real number can be represented in the form $\sum_{k \leq n} a_k 10^k$ with all $a_k \in D$.

- 25.** [M25] (S. A. Cook.) Let b , u , and v be positive integers, where $b \geq 2$ and $0 < v < b^m$. Show that the base b representation of u/v does not contain a run of m consecutive digits equal to $b-1$, anywhere to the right of the radix point. (By convention, no runs of infinitely many $(b-1)$'s are permitted in the standard base b representation.)

- **26.** [HM90] (N. S. Mendelsohn.) Let $\langle \beta_n \rangle$ be a sequence of real numbers defined for all integers n , $-\infty < n < \infty$, such that

$$\beta_n < \beta_{n+1}; \quad \lim_{n \rightarrow \infty} \beta_n = \infty; \quad \lim_{n \rightarrow -\infty} \beta_n = 0.$$

Let $\langle c_n \rangle$ be an arbitrary sequence of positive integers that is defined for all integers n , $-\infty < n < \infty$. Let us say that a number x has a “generalized representation” if there is an integer n and an infinite sequence of integers $a_n, a_{n-1}, a_{n-2}, \dots$ such that $x = \sum_{k \leq n} a_k \beta_k$, where $a_n \neq 0$, $0 \leq a_k \leq c_k$, and $a_k < c_k$ for infinitely many k .

Show that every positive real number x has exactly one generalized representation if and only if $\beta_{n+1} = \sum_{k \leq n} c_k \beta_k$ for all n . (Consequently, the mixed-radix systems with integer bases have this property; and mixed-radix systems with $\beta_1 = (c_0 + 1)\beta_0$, $\beta_2 = (c_1 + 1)(c_0 + 1)\beta_0$, \dots , $\beta_{-1} = \beta_0/(c_{-1} + 1)$, \dots are the most general number systems of this type.)

- 27.** [M21] Show that every nonzero integer has a unique “reversing binary representation”

$$2^{e_0} - 2^{e_1} + \cdots + (-1)^t 2^{e_t},$$

where $e_0 < e_1 < \cdots < e_t$.

- 28.** [M24] Show that every nonzero complex number of the form $a + bi$ where a and b are integers has a unique “revolving binary representation”

$$(1+i)^{e_0} + i(1+i)^{e_1} - (1+i)^{e_2} - i(1+i)^{e_3} + \cdots + i^t(1+i)^{e_t},$$

where $e_0 < e_1 < \cdots < e_t$. (Cf. exercise 27.)

- 29.** [M35] (N. G. de Bruijn.) Let S_0, S_1, S_2, \dots be sets of nonnegative integers; we will say that the collection $\{S_0, S_1, S_2, \dots\}$ has Property B if every nonnegative integer n can be written in the form

$$n = s_0 + s_1 + s_2 + \cdots, \quad s_j \in S_j,$$

in exactly one way. (Property B implies that $0 \in S_j$ for all j , since $n = 0$ can only be represented as $0 + 0 + 0 + \cdots$) Any mixed-radix number system with radices b_0, b_1, b_2, \dots provides an example of a collection of sets satisfying Property B, if we let $S_j = \{0, B_j, \dots, (b_j - 1)B_j\}$, where $B_j = b_0 b_1 \dots b_{j-1}$; here the representation of $n = s_0 + s_1 + s_2 + \cdots$ corresponds in an obvious manner to its mixed-radix representation (9). Furthermore, if the collection $\{S_0, S_1, S_2, \dots\}$ has Property B, and if A_0, A_1, A_2, \dots is any partition of the nonnegative integers (so that we have $A_0 \cup A_1 \cup A_2 \cup \cdots = \{0, 1, 2, \dots\}$ and $A_i \cap A_j = \emptyset$ for $i \neq j$; some A_j 's may be empty), then the “collapsed” collection $\{T_0, T_1, T_2, \dots\}$ also has Property B, where T_j is the set of all sums $\sum_{i \in A_j} s_i$ taken over all possible choices of $s_i \in S_i$.

Prove that any collection $\{T_0, T_1, T_2, \dots\}$ that satisfies Property B may be obtained by collapsing some collection $\{S_0, S_1, S_2, \dots\}$ that corresponds to a mixed-radix number system.

- 30.** [M39] (N. G. de Bruijn.) The radix-(-2) number system shows us that every integer (positive, negative, or zero) has a unique representation of the form

$$(-2)^{e_1} + (-2)^{e_2} + \cdots + (-2)^{e_t}, \quad e_1 > e_2 > \cdots > e_t \geq 0, \quad t \geq 0.$$

The purpose of this exercise is to explore generalizations of this phenomenon.

- a) Let b_0, b_1, b_2, \dots be a sequence of integers such that every integer n has a unique representation of the form

$$n = b_{e_1} + b_{e_2} + \cdots + b_{e_t}, \quad e_1 > e_2 > \cdots > e_t \geq 0, \quad t \geq 0.$$

(Such a sequence $\langle b_n \rangle$ is called a “binary basis.”) Show that there is an index j such that b_j is odd, but b_k is even for all $k \neq j$.

- b) Prove that a binary basis $\langle b_n \rangle$ can always be rearranged into the form $d_0, 2d_1, 4d_2, \dots = \langle 2^n d_n \rangle$, where each d_k is odd.
- c) If each of d_0, d_1, d_2, \dots in (b) is ± 1 , prove that $\langle b_n \rangle$ is a binary basis if and only if there are infinitely many $+1$'s and infinitely many -1 's.
- d) Prove that $7, -13 \cdot 2, 7 \cdot 2^2, -13 \cdot 2^3, \dots, 7 \cdot 2^{2k}, -13 \cdot 2^{2k+1}, \dots$ is a binary basis, and find the representation of $n = 1$.

- 31. [M35] A generalization of two's complement arithmetic, called “2-adic numbers,” was invented about 1900 by K. Hensel. (In fact he treated p -adic numbers, for any prime p .) A 2-adic number may be regarded as a binary number

$$u = (\dots u_3 u_2 u_1 u_0. u_{-1} \dots u_{-n})_2,$$

whose representation extends infinitely far to the left, but only finitely many places to the right, of the binary point. Addition, subtraction, and multiplication of 2-adic numbers are done according to the ordinary procedures of arithmetic, which can in principle be extended indefinitely to the left. For example,

$$\begin{array}{ll} 7 = (\dots 00000000000111)_2 & \frac{1}{7} = (\dots 110110110110111)_2 \\ -7 = (\dots 11111111111001)_2 & -\frac{1}{7} = (\dots 001001001001001)_2 \\ \frac{7}{4} = (\dots 0000000000001.11)_2 & \frac{1}{10} = (\dots 110011001100110.1)_2 \\ \sqrt{-7} = (\dots 100000010110101)_2 \text{ or } (\dots 01111101001011)_2. \end{array}$$

Here 7 appears as the ordinary binary integer seven, while -7 is its two's complement (extending infinitely to the left); it is easy to verify that the ordinary procedure for addition of binary numbers will give $-7 + 7 = (\dots 00000)_2 = 0$, when the procedure is continued indefinitely. The values of $\frac{1}{7}$ and $-\frac{1}{7}$ are the unique 2-adic numbers that, when formally multiplied by 7, give 1 and -1 , respectively. The values of $\frac{7}{4}$ and $\frac{1}{10}$ are examples of 2-adic numbers that are not 2-adic “integers,” since they have nonzero bits to the right of the binary point. The two values of $\sqrt{-7}$, which are negatives of each other, are the only 2-adic numbers that, when formally squared, yield the value $(\dots 11111111111001)_2$.

- a) Prove that any 2-adic number u can be divided by any nonzero 2-adic number v to obtain a unique 2-adic number w satisfying $u = vw$. (Hence the set of 2-adic numbers forms a “field”; cf. Section 4.6.1.)
- b) Prove that the 2-adic representation of the rational number $-1/(2n+1)$ may be obtained as follows, when n is a positive integer: First find the ordinary binary expansion of $+1/(2n+1)$, which has the periodic form $(0.\alpha\alpha\alpha\dots)_2$ for some string α of 0's and 1's. Then $-1/(2n+1)$ is the 2-adic number $(\dots\alpha\alpha\alpha)_2$.
- c) Prove that the representation of a 2-adic number u is ultimately periodic (that is, $u_{N+\lambda} = u_N$ for all large N , for some $\lambda \geq 1$) if and only if u is rational (that is, $u = m/n$, for some integers m and n).
- d) Prove that, when n is an integer, \sqrt{n} is a 2-adic number if and only if it satisfies $n \bmod 2^{2k+3} = 2^{2k}$ for some nonnegative integer k . (Thus, the possibilities are either $n \bmod 8 = 1$, or $n \bmod 32 = 4$, etc.)

32. [M40] (I. Z. Ruzsa.) Prove that there are infinitely many integers whose ternary representation uses only 0's and 1's and whose quinary representation uses only 0's, 1's, and 2's.

33. [M40] (D. A. Klarner.) Let D be any set of integers, let b be any positive integer, and let k_n be the number of distinct integers that can be written as n -digit numbers $(a_{n-1} \dots a_1 a_0)_b$ to base b with digits a_i in D . Prove that the sequence $\langle k_n \rangle$ satisfies a linear recurrence relation, and explain how to compute the generating function $\sum_n k_n z^n$. Illustrate your algorithm in the case $b = 3$ and $D = \{-1, 0, 3\}$.

4.2. FLOATING POINT ARITHMETIC

IN THIS SECTION, we shall study the basic principles of doing arithmetic on “floating point” numbers, by analyzing the internal mechanisms underlying such calculations. Perhaps many readers will have little interest in this subject, since their computers either have built-in floating point instructions or their computer manufacturer has supplied suitable subroutines. But, in fact, the material of this section should not merely be the concern of computer-design engineers or of a small clique of people who write library subroutines for new machines; every well-rounded programmer ought to have a knowledge of what goes on during the elementary steps of floating point arithmetic. This subject is not at all as trivial as most people think; it involves a surprising amount of interesting information.

4.2.1. Single-Precision Calculations

A. Floating point notation. We have discussed “fixed point” notation for numbers in Section 4.1; in such a case the programmer knows where the radix point is assumed to lie in the numbers he manipulates. For many purposes it is considerably more convenient to let the position of the radix point be dynamically variable or “floating” as a program is running, and to carry with each number an indication of its current radix point position. This idea has been used for many years in scientific calculations, especially for expressing very large numbers like Avogadro’s number $N = 6.02252 \times 10^{23}$, or very small numbers like Planck’s constant $\hbar = 1.0545 \times 10^{-27}$ erg sec.

In this section we shall work with *base b*, *excess q*, *floating point numbers with p digits*: Such numbers will be represented by pairs of values (e, f) , denoting

$$(e, f) = f \times b^{e-q}. \quad (1)$$

Here e is an integer having a specified range, and f is a signed fraction. We will adopt the convention that

$$|f| < 1;$$

in other words, the radix point appears at the left of the positional representation of f . More precisely, the stipulation that we have p -digit numbers means that $b^p f$ is an integer, and that

$$-b^p < b^p f < b^p. \quad (2)$$

The term “floating binary” implies that $b = 2$, “floating decimal” implies $b = 10$, etc. Using excess-50 floating decimal numbers with 8 digits, we can write, for example,

$$\begin{aligned} \text{Avogadro's number } N &= (74, +.60225200); \\ \text{Planck's constant } \hbar &= (24, +.10545000). \end{aligned} \quad (3)$$

The two components e and f of a floating point number are called the exponent and the fraction parts, respectively. (Other names are occasionally

used for this purpose, notably “characteristic” and “mantissa”; but it is an abuse of terminology to call the fraction part a mantissa, since this concept has quite a different meaning in connection with logarithms. Furthermore the English word mantissa means “a worthless addition.”)

The MIX computer assumes that its floating point numbers have the form

$$\boxed{\pm \quad e \quad f \quad f \quad f \quad f}.$$
 (4)

Here we have base b , excess q , floating point notation with four bytes of precision, where b is the byte size (e.g., $b = 64$ or $b = 100$), and q is equal to $\lfloor \frac{1}{2}b \rfloor$. The fraction part is $\pm f f f f$, and e is the exponent, which lies in the range $0 \leq e < b$. This internal representation is typical of the conventions in most existing computers, although b is a much larger base than usual.

B. Normalized calculations. A floating point number (e, f) is *normalized* if the most significant digit of the representation of f is nonzero, so that

$$1/b \leq |f| < 1; \quad (5)$$

or if $f = 0$ and e has its smallest possible value. It is possible to tell which of two normalized floating point numbers has a greater magnitude by comparing the exponent parts first, and then testing the fraction parts only if the exponents are equal.

Most floating point routines now in use deal almost entirely with normalized numbers: inputs to the routines are assumed to be normalized, and the outputs are always normalized. Under these conventions we lose the ability to represent a few numbers of very small magnitude—for example, the value $(0, .00000001)$ can't be normalized without producing a negative exponent—but we gain in speed, uniformity, and the ability to give relatively simple bounds on the relative error in our computations. (Unnormalized floating point arithmetic is discussed in Section 4.2.2.)

Let us now study the normalized floating point operations in detail. At the same time we can consider the construction of subroutines for these operations, assuming that we have a computer without built-in floating point hardware.

Machine-language subroutines for floating point arithmetic are usually written in a very machine-dependent manner, using many of the wildest idiosyncrasies of the computer at hand; so floating point addition subroutines for two different machines usually bear little superficial resemblance to each other. Yet a careful study of numerous subroutines for both binary and decimal computers reveals that these programs actually have quite a lot in common, and it is possible to discuss the topics in a machine-independent way.

The first (and by far the most difficult!) algorithm we shall discuss in this section is a procedure for floating point addition,

$$(e_u, f_u) \oplus (e_v, f_v) = (e_w, f_w). \quad (6)$$

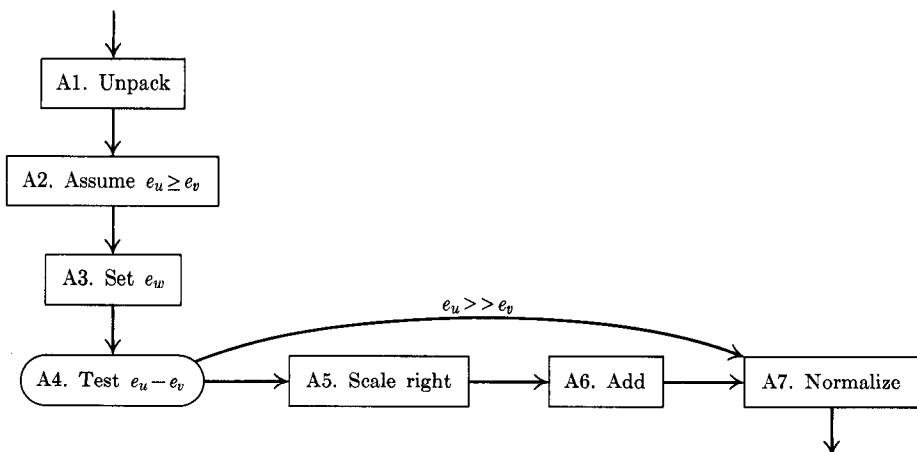


Fig. 2. Floating point addition.

Note: Since floating point arithmetic is inherently approximate, not exact, we will use “round” symbols

$\oplus, \ominus, \otimes, \oslash$

to stand for floating point addition, subtraction, multiplication, and division, respectively, in order to distinguish approximate operations from the true ones.

The basic idea involved in floating point addition is fairly simple: Assuming that $e_u \geq e_v$, we take $e_w = e_u$, $f_w = f_u + f_v/b^{e_u - e_v}$ (thereby aligning the radix points for a meaningful addition), and normalize the result. Several situations can arise that make this process nontrivial, and the following algorithm explains the method more precisely.

Algorithm A (Floating point addition). Given base b , excess q , p -digit, normalized floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$, this algorithm forms the sum $w = u \oplus v$. The same procedure may be used for floating point subtraction, if $-v$ is substituted for v .

- A1. [Unpack.] Separate the exponent and fraction parts of the representations of u and v .
- A2. [Assume $e_u \geq e_v$.] If $e_u < e_v$, interchange u and v . (In many cases, it is best to combine step A2 with step A1 or with some of the later steps.)
- A3. [Set e_w .] Set $e_w \leftarrow e_u$.
- A4. [Test $e_u - e_v$.] If $e_u - e_v \geq p+2$ (large difference in exponents), set $f_w \leftarrow f_u$ and go to step A7. (Actually, since we are assuming that u is normalized, we could terminate the algorithm; but it is occasionally useful to be able to normalize a possibly unnormalized number by adding zero to it.)
- A5. [Scale right.] Shift f_v to the right $e_u - e_v$ places; i.e., divide it by $b^{e_u - e_v}$. [Note: This will be a shift of up to $p+1$ places, and the next step (which adds f_u to f_v) thereby requires an accumulator capable of holding $2p+1$ digits.]

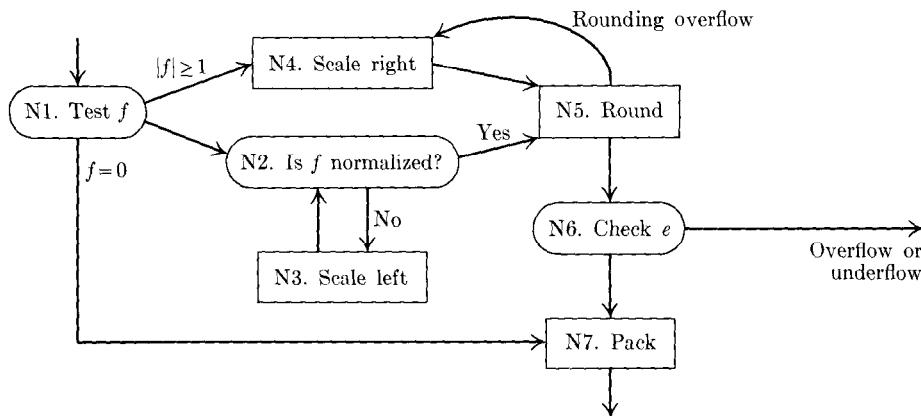


Fig. 3. Normalization of (e, f) .

base- b digits to the right of the radix point. If such a large accumulator is not available, it is possible to shorten the requirement to $p + 2$ or $p + 3$ places if proper precautions are taken; the details are given in exercise 5.]

A6. [Add.] Set $f_w \leftarrow f_u + f_v$.

A7. [Normalize.] (At this point (e_w, f_w) represents the sum of u and v , but $|f_w|$ may have more than p digits, and it may be greater than unity or less than $1/b$.) Perform Algorithm N below, to normalize and round (e_w, f_w) into the final answer. ■

Algorithm N (Normalization). A “raw exponent” e and a “raw fraction” f are converted to normalized form, rounding if necessary to p digits. This algorithm assumes that $|f| < b$.

N1. [Test f .] If $|f| \geq 1$ (“fraction overflow”), go to step N4. If $f = 0$, set e to its lowest possible value and go to step N7.

N2. [Is f normalized?] If $|f| \geq 1/b$, go to step N5.

N3. [Scale left.] Shift f to the left by one digit position (i.e., multiply it by b), and decrease e by 1. Return to step N2.

N4. [Scale right.] Shift f to the right by one digit position (i.e., divide it by b), and increase e by 1.

N5. [Round.] Round f to p places. (We take this to mean that f is changed to the nearest multiple of b^{-p} . It is possible that $(b^p f) \bmod 1 = \frac{1}{2}$ so that there are two nearest multiples; if b is even, we choose the one that makes $b^p f + \frac{1}{2}b$ odd. Further discussion of rounding appears in Section 4.2.2.) It is important to note that this rounding operation can make $|f| = 1$ (“rounding overflow”); in such a case, return to step N4.

N6. [Check e .] If e is too large, i.e., larger than its allowed range, an exponent overflow condition is sensed. If e is too small, an exponent underflow condition is sensed. (See the discussion below; since the result cannot

be expressed as a normalized floating point number in the required range, special action is necessary.)

N7. [Pack.] Put e and f together into the desired output representation. ■

Some simple examples of floating point addition are given in exercise 4.

The following MIX subroutines, for addition and subtraction of numbers having the form (4), show how Algorithms A and N can be expressed as computer programs. The subroutines below are designed to take one input u from symbolic location ACC, and the other input v comes from register A upon entrance to the subroutine. The output w appears both in register A and location ACC. Thus, a fixed point coding sequence

LDA A; ADD B; SUB C; STA D (7)

would correspond to the floating point coding sequence

LDA A, STA ACC; LDA B, JMP FADD; LDA C, JMP FSUB; STA D. (8)

Program A (Addition, subtraction, and normalization). The following program is a subroutine for Algorithm A, and it is also designed so that the normalization portion can be used by other subroutines that appear later in this section. In this program and in many others throughout this chapter, OFLO stands for a subroutine that prints out a message to the effect that MIX's overflow toggle was unexpectedly found to be "on." The byte size b is assumed to be a multiple of 4. The normalization routine NORM assumes that rI2 = e and rAX = f , where rA = 0 implies rX = 0 and rI2 < b .

00	BYTE	EQU	1(4:4)	Byte size b
01	EXP	EQU	1:1	Definition of exponent field
02	FSUB	STA	TEMP	Floating point subtraction subroutine:
03		LDAN	TEMP	Change sign of operand.
04	FADD	STJ	EXITF	Floating point addition subroutine:
05		JOV	OFLO	Ensure overflow is off.
06		STA	TEMP	TEMP $\leftarrow v$.
07		LDX	ACC	rX $\leftarrow u$.
08		CMPA	ACC(EXP)	<u>Steps A1, A2, A3 are combined here:</u>
09		JGE	1F	Jump if $e_v \geq e_u$.
10		STX	FU(0:4)	FU $\leftarrow \pm f f f f 0$.
11		LD2	ACC(EXP)	rI2 $\leftarrow e_w$.
12		STA	FV(0:4)	
13		LD1N	TEMP(EXP)	rI1 $\leftarrow -e_v$.
14		JMP	4F	
15	1H	STA	FU(0:4)	FU $\leftarrow \pm f f f f 0$ (u, v interchanged).
16		LD2	TEMP(EXP)	rI2 $\leftarrow e_w$.
17		STX	FV(0:4)	
18		LD1N	ACC(EXP)	rI1 $\leftarrow -e_v$.

19	4H	INC1 0,2	rI1 $\leftarrow e_u - e_v$. (Step A4 unnecessary.)
20	5H	LDA FV	<u>A5. Scale right.</u>
21		ENTX 0	Clear rX.
22		SRAZ 0,1	Shift right $e_u - e_v$ places.
23	6H	ADD FU	<u>A6. Add.</u>
24		JOV N4	<u>A7. Normalize.</u> Jump if fraction overflow.
25		JXZ NORM	Easy case?
26		CMPA =0=(1:1)	Is f normalized?
27		JNE N5	If so, round it.
28		SRC 5	$ rX \leftrightarrow rA $.
29		DECX 1	(rX is positive.)
30		STA TEMP	(Operands had opposite signs,
31		STA HALF(0:0)	registers must be adjusted
32		LDAN TEMP	before rounding and normalization.)
33		ADD HALF	
34		ADD HALF	Complement least significant portion.
35		SRC 4	Jump into normalization routine.
36		JMP N3A	
37	HALF	CON 1//2	One half the word size (Sign varies)
38	FU	CON 0	Fraction part f_u
39	FV	CON 0	Fraction part f_v
40	NORM	JAZ ZRO	<u>N1. Test f.</u>
41	N2	CMPA =0=(1:1)	<u>N2. Is f normalized?</u>
42		JNE N5	To N5 if leading byte nonzero.
43	N3	SLAX 1	<u>N3. Scale left.</u>
44	N3A	DEC2 1	Decrease e by 1.
45		JMP N2	Return to N2.
46	N4	ENTX 1	<u>N4. Scale right.</u>
47		SRC 1	Shift right, insert "1" with proper sign.
48		INC2 1	Increase e by 1.
49	N5	CMPA =BYTE/2=(5:5)	<u>N5. Round.</u>
50		JL N6	Is $ \text{tail} < \frac{1}{2}b$?
51		JG 5F	
52		JXNZ 5F	Is $ \text{tail} > \frac{1}{2}b$?
53		STA TEMP	$ \text{tail} = \frac{1}{2}b$; round to odd.
54		LDX TEMP(4:4)	
55		JXO N6	To N6 if rX is odd.
56	5H	STA **+1(0:0)	Store sign of rA.
57		INCA BYTE	Add b^{-4} to $ f $. (Sign varies)
58		JOV N4	Check for rounding overflow.
59	N6	J2N EXPUN	<u>N6. Check e.</u> Underflow if $e < 0$.
60	N7	ENTX 0,2	<u>N7. Pack.</u> rX $\leftarrow e$.
61		SRC 1	
62	ZRO	DEC2 BYTE	$rI2 \leftarrow e - b$.
63	8H	STA ACC	
64	EXITF	J2N *	Exit, unless $e \geq b$.
65	EXPOV	HLT 2	Exponent overflow detected
66	EXPUN	HLT 1	Exponent underflow detected
67	ACC	CON 0	Floating point accumulator ■

The rather long section of code from lines 25 to 37 is needed because MIX has only a 5-byte accumulator for adding signed numbers while in general $2p+1 = 9$ places of accuracy are required by Algorithm A. The program could be shortened to about half its present length if we were willing to sacrifice a little bit of its accuracy, but we shall see in the next section that full accuracy is important. Line 55 uses a nonstandard MIX instruction defined in Section 4.5.2. The running time for floating point addition and subtraction depends on several factors that are analyzed in Section 4.2.4.

Now let us consider multiplication and division, which are simpler than addition, and which are somewhat similar to each other.

Algorithm M (*Floating point multiplication or division*). Given base b , excess q , p -digit, normalized floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$, this algorithm forms the product $w = u \otimes v$ or the quotient $w = u \oslash v$.

M1. [Unpack.] Separate the exponent and fraction parts of the representations of u and v . (Sometimes it is convenient, but not necessary, to test the operands for zero during this step.)

M2. [Operate.] Set

$$\begin{aligned} e_w &\leftarrow e_u + e_v - q, & f_w &\leftarrow f_u f_v && \text{for multiplication;} \\ e_w &\leftarrow e_u - e_v + q + 1, & f_w &\leftarrow (b^{-1} f_u) / f_v && \text{for division.} \end{aligned} \quad (9)$$

(Since the input numbers are assumed to be normalized, it follows that either $f_w = 0$, or $1/b^2 \leq |f_w| < 1$, or a division-by-zero error has occurred.) If necessary, the representation of f_w may be reduced to $p+2$ or $p+3$ digits at this point, as in exercise 5.

M3. [Normalize.] Perform Algorithm N on (e_w, f_w) to normalize, round, and pack the result. (Note: Normalization is simpler in this case, since scaling left occurs at most once, and since rounding overflow cannot occur after division.) ■

The following MIX subroutines, which are intended to be used in connection with Program A, illustrate the machine considerations necessary in connection with Algorithm M.

Program M (*Floating point multiplication and division*).

01	Q	EQU	BYTE/2	q is half the byte size
02	FMUL	STJ	EXITF	Floating point multiplication subroutine:
03		JOV	OFLO	Ensure overflow is off.
04		STA	TEMP	$\text{TEMP} \leftarrow v$.
05		LDX	ACC	$rX \leftarrow u$.
06		STX	FU(0:4)	$FU \leftarrow \pm f f f f 0$.
07		LD1	TEMP(EXP)	
08		LD2	ACC(EXP)	
09		INC2	-Q,1	$rI2 \leftarrow e_u + e_v - q$.
10		SLA	1	
11		MUL	FU	Multiply f_u times f_v .
12		JMP	NORM	Normalize, round, and exit.

13	FDIV	STJ EXITF	Floating point division subroutine:
14		JOV OFLO	Ensure overflow is off.
15		STA TEMP	TEMP $\leftarrow v$.
16		STA FV(0:4)	FV $\leftarrow \pm f f f f 0$.
17		LD1 TEMP(EXP)	
18		LD2 ACC(EXP)	
19		DEC2 -Q,1	rI2 $\leftarrow e_u - e_v + q$.
20		ENTX 0	
21		LDA ACC	
22		SLA 1	rA $\leftarrow f_u$.
23		CMPA FV(1:5)	
24		JL *+3	Jump if $ f_u < f_v $.
25		SRA 1	Otherwise, scale f_u right
26		INC2 1	and increase rI2 by 1.
27		DIV FV	Divide.
28		JNOV NORM	Normalize, round, and exit.
29	DVZRO	HLT 3	Unnormalized or zero divisor ■

The most noteworthy feature of this program is the provision for division in lines 23–26, which is made in order to ensure enough accuracy to round the answer. If $|f_u| < |f_v|$, straightforward application of Algorithm M would leave a result of the form “ $\pm 0 f f f f$ ” in register A, and this would not allow a proper rounding without a careful analysis of the remainder (which appears in register X). So the program computes $f_w \leftarrow f_u/f_v$ in this case, ensuring that f_w is either zero or normalized in all cases; rounding can proceed with five significant bytes, possibly testing whether the remainder is zero.

We occasionally need to convert values between fixed and floating point representations. A “fix-to-float” routine is easily obtained with the help of the normalization algorithm above; for example, in MIX, the following subroutine converts an integer to floating point form:

01	FLOT	STJ EXITF	Assume that rA = u, an integer.
02		JOV OFLO	Ensure overflow is off.
03		ENT2 Q+5	Set raw exponent. (10)
04		ENTX 0	
05		JMP NORM	Normalize, round, and exit. ■

A “float-to-fix” subroutine is the subject of exercise 14.

The debugging of floating point subroutines is usually a difficult job, since there are so many cases to consider. Here is a list of common pitfalls that often trap a programmer or machine designer who is preparing floating point routines:

1) *Losing the sign.* On many machines (not MIX), shift instructions between registers will affect the sign, and the shifting operations used in normalizing and scaling numbers must be carefully analyzed. The sign is also lost frequently when minus zero is present. (For example, Program A is careful to retain the sign of register A in lines 30–34. See also exercise 6.)

2) *Failure to treat exponent underflow or overflow properly.* The size of e_w should not be checked until after the rounding and normalization, because preliminary tests may give an erroneous indication. Exponent underflow and overflow can occur on floating point addition and subtraction, not only during multiplication and division; and even though this is a rather rare occurrence, it must be tested each time. Enough information should be retained so that meaningful corrective actions are possible after overflow or underflow has occurred.

It has unfortunately become customary in many instances to ignore exponent underflow and simply to set underflowed results to zero with no indication of error. This causes a serious loss of accuracy in most cases (indeed, it is the loss of *all* the significant digits), and the assumptions underlying floating point arithmetic have broken down, so the programmer really must be told when underflow has occurred. Setting the result to zero is appropriate only in certain cases when the result is later to be added to a significantly larger quantity. When exponent underflow is not detected, we find mysterious situations in which $(u \otimes v) \otimes w$ is zero, but $u \otimes (v \otimes w)$ is not, since $u \otimes v$ results in exponent underflow but $u \otimes (v \otimes w)$ can be calculated without any exponents falling out of range. Similarly, we can find positive numbers a, b, c, d , and y such that

$$(a \otimes y \oplus b) \oslash (c \otimes y \oplus d) \approx \frac{2}{3}, \quad (11)$$

$$(a \oplus b \oslash y) \oslash (c \oplus d \oslash y) = 1$$

if exponent underflow is not detected. (See exercise 9.) Even though floating point routines are not precisely accurate, such a disparity as (11) is certainly unexpected when a, b, c, d , and y are all *positive!* Exponent underflow is usually not anticipated by a programmer, so he needs to be told about it.*

3) *Inserted garbage.* When scaling to the left it is important to keep from introducing anything but zeros at the right. For example, note the "ENTX 0" instruction in line 21 of Program A, and the all-too-easily-forgotten "ENTX 0" instruction in line 04 of the FLOT subroutine (10). (But it would be a mistake to clear register X after line 27 in the division subroutine.)

*On the other hand, it must be admitted that today's high-level programming languages give the programmer little or no satisfactory way to make use of the information that a floating point routine wants to tell him; and the MIX programs in this section, which simply 'HLT' when errors are detected, are even worse. There are numerous important applications in which exponent underflow is relatively harmless, and it is desirable to find a way for programmers to cope with such situations easily and safely. The practice of silently replacing underflows by zero has been thoroughly discredited, but there is another alternative that has recently been gaining much favor, namely to modify the definition we have given for floating point numbers, allowing an unnormalized fraction part when the exponent has its smallest possible value. This idea of "gradual underflow," which was first embodied in the hardware of the Electrologica X8 computer, adds only a small amount of complexity to the algorithms, and it makes exponent underflow impossible during addition or subtraction. The simple formulas for relative error in Section 4.2.2 no longer hold in the presence of gradual underflow, so the topic is beyond the scope of this book. However, by using formulas like $\text{round}(x) = x(1 - \delta) + \epsilon$, where $|\delta| < \frac{1}{2}b^{1-p}$ and $|\epsilon| < \frac{1}{2}b^{-p-q}$, one can show that gradual underflow succeeds in many important cases. See W. M. Kahan and J. Palmer, *ACM SIGNUM Newsletter* (Oct. 1979), 13-21.

4) *Unforeseen rounding overflow.* When a number like .999999997 is rounded to 8 digits, a carry will occur to the left of the decimal point, and the result must be scaled to the right. Many people have mistakenly concluded that rounding overflow is impossible during multiplication, since they look at the maximum value of $|f_u f_v|$, which is $1 - 2b^{-p} + b^{-2p}$; and this cannot round up to 1. The fallacy in this reasoning is exhibited in exercise 11. Curiously, it turns out that the phenomenon of rounding overflow is impossible during floating point division (see exercise 12).

There is a school of thought that says it is harmless to “round” a value like .999999997 to .99999999 instead of to 1.0000000, since this does not increase the worst-case bounds on relative error. The floating point number 1.0000000 may be said to represent all real values in the interval $[1.0000000 - 5 \times 10^{-8}, 1.0000000 + 5 \times 10^{-8}]$, while .99999999 represents all values in the much smaller interval $(.99999999 - 5 \times 10^{-9}, .99999999 + 5 \times 10^{-9})$. Even though the latter interval does not contain the original value .999999997, each number of the second interval is contained in the first, so subsequent calculations with the second interval are no less accurate than with the first. This ingenious argument is, however, incompatible with the mathematical philosophy of floating point arithmetic expressed in Section 4.2.2.

5) *Rounding before normalizing.* Inaccuracies are caused by premature rounding in the wrong digit position. This error is obvious when rounding is being done to the left of the appropriate position; but it is also dangerous in the less obvious cases where rounding is first done too far to the right, followed by rounding in the true position. For this reason it is a mistake to round during the “scaling-right” operation in step A5, except as prescribed in exercise 5. (The special case of rounding in step N5, then rounding again after rounding overflow has occurred, is harmless, however, because rounding overflow always yields ± 1.0000000 and this is unaffected by the subsequent rounding process.)

6) *Failure to retain enough precision in intermediate calculations.* Detailed analyses of the accuracy of floating point arithmetic, made in the next section, suggest strongly that normalizing floating point routines should always deliver a properly rounded result to the maximum possible accuracy. There should be no exceptions to this dictum, even in cases that occur with extremely low probability; the appropriate number of significant digits should be retained throughout the computations, as stated in Algorithms A and M.

C. Floating point hardware. Nearly every large computer intended for scientific calculations includes floating point arithmetic as part of its repertoire of built-in operations. Unfortunately, the design of such hardware usually includes some anomalies that result in dismally poor behavior in certain circumstances, and we hope that future computer designers will pay more attention to providing the proper behavior than they have in the past. It costs only a little more to build the machine right, and considerations in the following section show that substantial benefits will be gained. Yesterday’s compromises are no longer appropriate for modern machines, based on what we know now.

The MIX computer, which is being used as an example of a "typical" machine in this series of books, has an optional "floating point attachment" (available at extra cost) that includes the following seven operations:

- FADD, FSUB, FMUL, FDIV, FLOT, FCMP ($C = 1, 2, 3, 4, 5, 56$, respectively; $F = 6$). The contents of rA after the operation "FADD V" are precisely the same as the contents of rA after the operations

```
STA ACC
LDA V
JMP FADD
```

where FADD is the subroutine that appears earlier in this section, except that both operands are automatically normalized before entry to the subroutine if they are not already in normalized form. (If exponent underflow occurs during this pre-normalization, but not during the normalization of the answer, no underflow is signalled.) Similar remarks apply to FSUB, FMUL, and FDIV. The contents of rA after the operation "FLOT" are the contents after "JMP FLOT" in the subroutine (10) above.

The contents of rA are unchanged by the operation "FCMP V"; this instruction sets the comparison indicator to less, equal, or greater, depending on whether the contents of rA are "definitely less than," "approximately equal to," or "definitely greater than" V; this subject is discussed in the next section, and the precise action is defined by the subroutine FCMP of exercise 4.2.2-17 with EPSILON in location 0.

No register other than rA is affected by any of the floating point operations. If exponent overflow or underflow occurs, the overflow toggle is turned on and the exponent of the answer is given modulo the byte size. Division by zero leaves undefined garbage in rA. Execution times: $4u$, $4u$, $9u$, $11u$, $3u$, $4u$, respectively.

- FIX ($C = 5$; $F = 7$). The contents of rA are replaced by the integer "round(rA)", rounding to the nearest integer as in step N5 of Algorithm N. However, if this answer is too large to fit in the register, the overflow toggle is set on and the result is undefined. Execution time: $3u$.

Sometimes it is helpful to use floating point operators in a nonstandard way. For example, if the operation FLOT had not been included as part of MIX's floating point attachment, we could easily achieve its effect on 4-byte numbers by writing

```
FLOT STJ 9F
      SLA 1
      ENTX Q+4
      SRC 1
      FADD =0=
9H    JMP *   ■
```

(12)

This routine is not strictly equivalent to the FLOT operator, since it assumes that the 1:1 byte of rA is zero, and it destroys rX. The handling of more general situations is a little tricky, because rounding overflow can occur even during a FLOT operation.

Similarly, suppose MIX had a FADD operation but not FIX. If we wanted to round a number u from floating point form to the nearest fixed point integer, and if we knew that the number was nonnegative and would fit in at most three bytes, we could write

FADD FUDGE

where location FUDGE contains the constant

+	Q+4	1	0	0	0	;
---	-----	---	---	---	---	---

the result in rA would be

+	Q+4	1	round(u)	.
---	-----	---	--------------	---

(13)

D. History and bibliography. The origins of floating point notation can be traced back to Babylonian mathematicians (1800 B.C. or earlier), who made extensive use of radix-60 floating point arithmetic but did not have a notation for the exponents. The appropriate exponent was always somehow “understood” by the man doing the calculations. At least one case has been found in which the wrong answer was given because addition was performed with improper alignment of the operands, but such examples are very rare; see O. Neugebauer, *The Exact Sciences in Antiquity* (Princeton, N. J.: Princeton University Press, 1952), 26–27. Another early contribution to floating point notation is due to the Greek mathematician Apollonius (3rd century B.C.), who apparently was the first to explain how to simplify multiplication by collecting powers of 10 separately from their coefficients, at least in simple cases. [For a discussion of Apollonius’s method, see Pappus, *Mathematical Collections* (4th century A.D.).] After the Babylonian civilization died out, the first significant uses of floating point notation for products and quotients did not emerge until much later, about the time logarithms were invented (1600) and shortly afterwards when Oughtred invented the slide rule (1630). The modern notation “ x^n ” for exponents was being introduced at about the same time; separate symbols for x squared, x cubed, etc., had been in use before this.

Floating point arithmetic was incorporated into the design of some of the earliest computers. It was independently proposed by Leonardo Torres y Quevedo in Madrid, 1914; by Konrad Zuse in Berlin, 1936; and by George Stibitz in New Jersey, 1939. Zuse’s machines used a floating binary representation that he called “semi-logarithmic notation”; he also incorporated conventions for dealing with special quantities like “ ∞ ” and “undefined.” The first American computers to operate with floating point arithmetic hardware were the Bell Laboratories’ Model V and the Harvard Mark II, both of which were relay calculators designed in 1944. [See B. Randell, *The Origins of Digital Computers* (Berlin: Springer, 1973), 100, 155, 163–164, 259–260; *Proc. Symp. Large-Scale Digital Calculating Machinery* (Harvard, 1947), 41–68, 69–79; *Datamation* 13 (April 1967), 35–44 (May 1967), 45–49; *Zeit. für angew. Math. und Physik* 1 (1950), 345–346.]

The use of floating binary arithmetic was seriously considered in 1944–1946 by researchers at the Moore School in their plans for the first electronic digital computers, but it turned out to be much harder to implement floating point circuitry with tubes than with relays. The group realized that scaling was a problem in programming; but at the time it was only a very small part of a total programming job, and it seemed to be worth the time and trouble it took, since it tended to keep a programmer aware of the numerical accuracy he was getting. Furthermore, they argued that floating point representation would take up valuable memory space, since the exponents must be stored, and that it would be difficult to adapt floating point arithmetic to multiple-precision calculations. [See von Neumann's *Collected Works 5* (New York: Macmillan, 1963), 43, 73–74.] At this time, of course, they were designing the first stored-program computer and the second electronic computer, and their choice had to be either fixed point or floating point arithmetic, not both. They anticipated the coding of floating binary routines, and in fact “shift left” and “shift right” instructions were put into their machine primarily to make such routines more efficient. The first machine to have both kinds of arithmetic in its hardware was apparently a computer developed at General Electric Company [see *Proc. 2nd Symp. Large-Scale Digital Calculating Machinery* (Cambridge: Harvard University Press, 1951), 65–69].

Floating point subroutines and interpretive systems for early machines were coded by D. J. Wheeler and others, and the first publication of such routines was in *The Preparation of Programs for an Electronic Digital Computer* by Wilkes, Wheeler, and Gill (Reading, Mass.: Addison-Wesley, 1951), subroutines A1–A11, pp. 35–37, 105–117. It is interesting to note that floating decimal subroutines are described here, although a binary computer was being used; in other words, the numbers were represented as $10^e f$, not $2^e f$, and therefore the scaling operations required multiplication or division by 10. On this particular machine such decimal scaling was about as easy as shifting, and the decimal approach greatly simplified input/output conversions.

Most published references to the details of floating point arithmetic routines are scattered in “technical memorandums” distributed by various computer manufacturers, but there have been occasional appearances of these routines in the open literature. Besides the reference above, the following are of historical interest: R. H. Stark and D. B. MacMillan, *Math. Comp.* 5 (1951), 86–92, where a plugboard-wired program is described; D. McCracken, *Digital Computer Programming* (New York: Wiley, 1957), 121–131; J. W. Carr III, *CACM* 2,5 (May 1959), 10–15; W. G. Wadey, *JACM* 7 (1960), 129–139; D. E. Knuth, *JACM* 8 (1961), 119–128; O. Kesner, *CACM* 5 (1962), 269–271; F. P. Brooks and K. E. Iverson, *Automatic Data Processing* (New York: Wiley, 1963), 184–199. For a discussion of floating point arithmetic from a computer designer's standpoint, see “Floating point operation” by S. G. Campbell, in *Planning a computer System*, ed. by W. Buchholz (New York: McGraw-Hill, 1962), 92–121. A set of algorithms by J. Coonen, W. M. Kahan, and H. S. Stone, submitted to the IEEE Microprocessor Floating-Point Standards Committee during 1978–1980, represented

the state of the floating point art as of 1980; these carefully considered procedures will probably be published some day. Additional references, which deal primarily with the accuracy of floating point methods, are given in Section 4.2.2.

EXERCISES

1. [10] How would Avogadro's number and Planck's constant be represented in base 100, excess 50, four-digit floating point notation? (This would be the representation used by MIX, as in (4), if the byte size is 100.)

2. [12] Assume that the exponent e is constrained to lie in the range $0 \leq e \leq E$; what are the largest and smallest positive values that can be written as base b , excess q , p -digit floating point numbers? What are the largest and smallest positive values that can be written as *normalized* floating point numbers with these specifications?

3. [11] (K. Zuse, 1936.) Show that if we are using normalized floating binary arithmetic, there is a way to increase the precision slightly without loss of memory space: A p -bit fraction part can be represented using only $p - 1$ bit positions of a computer word, if the range of exponent values is decreased very slightly.

► 4. [12] Assume that $b = 10$, $p = 8$. What result does Algorithm A give for $(50, +.98765432) \oplus (49, +.33333333)$? For $(53, -.99987654) \oplus (54, +.10000000)$? For $(45, -.50000001) \oplus (54, +.10000000)$?

► 5. [24] Let us say that $x \sim y$ (with respect to a given radix b) if x and y are real numbers satisfying the following conditions:

$$\lfloor x/b \rfloor = \lfloor y/b \rfloor;$$

$$x \bmod b = 0 \quad \text{iff} \quad y \bmod b = 0;$$

$$0 < x \bmod b < \frac{1}{2}b \quad \text{iff} \quad 0 < y \bmod b < \frac{1}{2}b;$$

$$x \bmod b = \frac{1}{2}b \quad \text{iff} \quad y \bmod b = \frac{1}{2}b;$$

$$\frac{1}{2}b < x \bmod b < b \quad \text{iff} \quad \frac{1}{2}b < y \bmod b < b.$$

Prove that if f_v is replaced by $b^{-p-2}F_v$ between steps A5 and A6 of Algorithm A, where $F_v \sim b^{p+2}f_v$, the result of that algorithm will be unchanged. (If F_v is an integer and b is even, this operation essentially truncates f_v to $p + 2$ places while remembering whether any nonzero digits have been dropped, thereby minimizing the length of register that is needed for the addition in step A6.)

6. [20] If the result of a FADD instruction is zero, what will be the sign of rA, according to the definitions of MIX's floating point attachment given in this section?

7. [27] Discuss floating point arithmetic using balanced ternary notation.

8. [20] Give examples of normalized eight-digit floating decimal numbers u and v for which addition yields (a) exponent underflow, (b) exponent overflow, assuming that exponents must satisfy $0 \leq e < 100$.

9. [M24] (W. M. Kahan.) Assume that the occurrence of exponent underflow causes the result to be replaced by zero, with no error indication given. Using excess zero, eight-digit floating decimal numbers with e in the range $-50 \leq e < 50$, find positive values of a , b , c , d , and y such that (11) holds.

10. [12] Give an example of normalized eight-digit floating decimal numbers u and v for which rounding overflow occurs in addition.

- 11. [M20] Give an example of normalized, excess 50, eight-digit floating decimal numbers u and v for which rounding overflow occurs in multiplication.
12. [M25] Prove that rounding overflow cannot occur during the normalization phase of floating point division.
13. [30] When doing “interval arithmetic” we don’t want to round the results of a floating point computation; we want rather to implement operations such as ∇ and Δ , which give the tightest possible representable bounds on the true sum:

$$u \nabla v \leq u + v \leq u \Delta v.$$

How should the algorithms of this section be modified for such a purpose?

14. [25] Write a MIX subroutine that begins with an arbitrary floating point number in register A, not necessarily normalized, and converts it to the nearest fixed point integer (or determines that the number is too large in absolute value to make such a conversion possible).
- 15. [28] Write a MIX subroutine, to be used in connection with the other subroutines of this section, that calculates $u \text{ (mod) } 1$, that is, $u - \lfloor u \rfloor$ rounded to nearest floating point number, given a floating point number u . Note that when u is a very small negative number, $u \text{ (mod) } 1$ will be rounded so that the result is unity (even though $u \bmod 1$ has been defined to be always less than unity, as a real number).
16. [HM21] (Robert L. Smith.) Design an algorithm to compute the real and imaginary parts of the complex number $(a+bi)/(c+di)$, given real floating point values a , b , c , and d . Avoid the computation of $c^2 + d^2$, since it would cause floating point overflow even when $|c|$ or $|d|$ is approximately the square root of the maximum allowable floating point value.
17. [40] (John Cocke.) Explore the idea of extending the range of floating point numbers by defining a single-word representation in which the precision of the fraction decreases as the magnitude of the exponent increases.
18. [25] Consider a binary computer with 36-bit words, on which positive floating binary numbers are represented as $(0e_1e_2\dots e_8f_1f_2\dots f_{27})_2$; here $(e_1e_2\dots e_8)_2$ is an excess $(10000000)_2$ exponent and $(f_1f_2\dots f_{27})_2$ is a 27-bit fraction. Negative floating point numbers are represented by the two’s complement of the corresponding positive representation (see Section 4.1). Thus, 1.5 is $201|600000000$ in octal notation, while -1.5 is $576|200000000$; the octal representations of 1.0 and -1.0 are $201|400000000$ and $576|400000000$, respectively. (A vertical line is used here to show the boundary between exponent and fraction.) Note that bit f_1 of a normalized positive number is always 1, while it is almost always zero for negative numbers; the exceptional cases are representations of -2^k .

Suppose that the exact result of a floating point operation has the octal code $572|740000000|01$; this (negative) 33-bit fraction must be normalized and rounded to 27 bits. If we shift left until the leading fraction bit is zero, we get $576|00000000|20$, but this rounds to the illegal value $576|00000000$; we have over-normalized, since the correct answer is $575|400000000$. On the other hand if we start (in some other problem) with the value $572|740000000|05$ and stop before over-normalizing it, we get $575|400000000|50$, which rounds to the unnormalized number $575|400000001$; subsequent normalization yields $576|00000002$ while the correct answer is $576|000000001$.

Give a simple, correct rounding rule that resolves this dilemma on such a machine (without abandoning two’s complement notation).

Round numbers are always false.

—SAMUEL JOHNSON (1750)

I shall speak in round numbers, not absolutely accurate, yet not so wide from truth as to vary the result materially.

—THOMAS JEFFERSON (1824)

19. [24] What is the running time for the FADD subroutine in Program A, in terms of relevant characteristics of the data? What is the maximum running time, over all inputs that do not cause overflow or underflow?

4.2.2. Accuracy of Floating Point Arithmetic

Floating point computation is by nature inexact, and it is not difficult to misuse it so that the computed answers consist almost entirely of “noise.” One of the principal problems of numerical analysis is to determine how accurate the results of certain numerical methods will be. A “credibility-gap” problem is involved here: we don’t know how much of the computer’s answers to believe. Novice computer users solve this problem by implicitly trusting in the computer as an infallible authority; they tend to believe that all digits of a printed answer are significant. Disillusioned computer users have just the opposite approach, they are constantly afraid that their answers are almost meaningless. Many a serious mathematician has attempted to give rigorous analyses of a sequence of floating point operations, but has found the task to be so formidable that he has tried to content himself with plausibility arguments instead.

A thorough examination of error analysis techniques is, of course, beyond the scope of this book, but in this section we shall study some of the characteristics of floating point arithmetic errors. Our goal is to discover how to perform floating point arithmetic in such a way that reasonable analyses of error propagation are facilitated as much as possible.

A rough (but reasonably useful) way to express the behavior of floating point arithmetic can be based on the concept of “significant figures” or *relative error*. If we are representing an exact real number x inside a computer by using the approximation $\hat{x} = x(1 + \epsilon)$, the quantity $\epsilon = (\hat{x} - x)/x$ is called the relative error of approximation. Roughly speaking, the operations of floating point multiplication and division do not magnify the relative error by very much; but floating point subtraction of nearly equal quantities (and floating point addition, $u \oplus v$, where u is nearly equal to $-v$) can very greatly increase the relative error. So we have a general rule of thumb, that a substantial loss of accuracy is expected from such additions and subtractions, but not from multiplications and divisions. On the other hand, the situation is somewhat paradoxical and needs to be understood properly, since “bad” additions and subtractions are performed with perfect accuracy! (See exercise 25.)

One of the consequences of the possible unreliability of floating point addition is that the associative law breaks down:

$$(u \oplus v) \oplus w \neq u \oplus (v \oplus w), \quad \text{for many } u, v, w. \quad (1)$$

For example,

$$\begin{aligned} (11111113. \oplus -11111111.) \oplus 7.5111111 &= 2.0000000 \oplus 7.5111111 \\ &= 9.5111111; \end{aligned}$$

$$\begin{aligned} 11111113. \oplus (-11111111. \oplus 7.5111111) &= 11111113. \oplus -11111103. \\ &= 10.000000. \end{aligned}$$

(All examples in this section are given in eight-digit floating decimal arithmetic, with exponents indicated by an explicit decimal point. Recall that, as in Section 4.2.1, the symbols \oplus , \ominus , \otimes , \oslash are used to stand for floating point operations corresponding to the exact operations $+$, $-$, \times , $/$.)

In view of the failure of the associative law, the comment of Mrs. La Touche that appears at the beginning of this chapter [taken from *Math. Gazette* 12 (1924), 95] makes a good deal of sense with respect to floating point arithmetic. Mathematical notations like " $a_1 + a_2 + a_3$ " or " $\sum_{1 \leq k \leq n} a_k$ " are inherently based upon the assumption of associativity, so a programmer must be especially careful that he does not implicitly assume the validity of the associative law.

A. An axiomatic approach. Although the associative law is not valid, the commutative law

$$u \oplus v = v \oplus u \quad (2)$$

does hold, and this law can be a valuable conceptual asset in programming and in the analysis of programs. This example suggests that we should look for important laws that are satisfied by \oplus , \ominus , \otimes , and \oslash ; it is not unreasonable to say that *floating point routines should be designed to preserve as many of the ordinary mathematical laws as possible*. If more axioms are valid, it becomes easier to write good programs, and programs also become more portable from machine to machine.

Let us therefore consider some of the other basic laws that are valid for normalized floating point operations as described in the previous section. First we have

$$u \ominus v = u \oplus -v; \quad (3)$$

$$-(u \oplus v) = -u \oplus -v; \quad (4)$$

$$u \oplus v = 0 \quad \text{if and only if} \quad v = -u; \quad (5)$$

$$u \oplus 0 = u. \quad (6)$$

From these laws we can derive further identities; for example (exercise 1),

$$u \ominus v = -(v \ominus u). \quad (7)$$

Identities (2) to (6) are easily deduced from the algorithms in Section 4.2.1. The following rule is slightly less obvious:

$$\text{if } u \leq v \quad \text{then} \quad u \oplus w \leq v \oplus w. \quad (8)$$

Instead of attempting to prove this rule by analyzing Algorithm 4.2.1A, let us go back to the principle underlying the design of that algorithm. (Algorithmic proofs aren't always easier than mathematical ones.) Our idea was that the floating point operations should satisfy

$$\begin{aligned} u \oplus v &= \text{round}(u + v), & u \ominus v &= \text{round}(u - v), \\ u \otimes v &= \text{round}(u \times v), & u \oslash v &= \text{round}(u / v), \end{aligned} \quad (9)$$

where $\text{round}(x)$ denotes the best floating point approximation to x as defined in Algorithm 4.2.1N. We have

$$\text{round}(-x) = -\text{round}(x), \quad (10)$$

$$x \leq y \quad \text{implies} \quad \text{round}(x) \leq \text{round}(y), \quad (11)$$

and these fundamental relations prove properties (2) through (8) immediately. We can also write down several more identities:

$$u \otimes v = v \otimes u, \quad (-u) \otimes v = -(u \otimes v), \quad 1 \otimes v = v;$$

$$u \otimes v = 0 \quad \text{if and only if} \quad u = 0 \text{ or } v = 0;$$

$$(-u) \oslash v = u \oslash (-v) = -(u \oslash v);$$

$$0 \oslash v = 0, \quad u \oslash 1 = u, \quad u \oslash u = 1.$$

If $u \leq v$ and $w > 0$, then $u \otimes w \leq v \otimes w$ and $u \oslash w \leq v \oslash w$ and $w \oslash u \geq w \oslash v$. If $u \oplus v = u + v$, then $(u \oplus v) \ominus v = u$; and if $u \otimes v = u \times v \neq 0$, then $(u \otimes v) \oslash v = u$. We see that a good deal of regularity is present in spite of the inexactness of the floating point operations, when things have been defined properly.

Several familiar rules of algebra are still, of course, conspicuously absent from the collection of identities above; the associative law for floating point multiplication is not strictly true, as shown in exercise 3, and the distributive law between \otimes and \oplus can fail rather badly: Let $u = 20000.000$, $v = -6.0000000$, and $w = 6.0000003$; then

$$(u \otimes v) \oplus (u \otimes w) = -120000.00 \oplus 120000.01 = .010000000$$

$$u \otimes (v \oplus w) = 20000.000 \otimes .00000030000000 = .0060000000$$

so

$$u \otimes (v \oplus w) \neq (u \otimes v) \oplus (u \otimes w). \quad (12)$$

On the other hand we do have $b \otimes (v \oplus w) = (b \otimes v) \oplus (b \otimes w)$, when b is the floating point radix, since

$$\text{round}(bx) = b \text{round}(x). \quad (13)$$

(Strictly speaking, the identities and inequalities we are considering in this section implicitly assume that exponent underflow and overflow do not occur. The function $\text{round}(x)$ is undefined when $|x|$ is too small or too large, and equations such as (13) hold only when both sides are defined.)

The failure of Cauchy's fundamental inequality

$$(x_1^2 + \cdots + x_n^2)(y_1^2 + \cdots + y_n^2) \geq (x_1 y_1 + \cdots + x_n y_n)^2$$

is another important example of the breakdown of traditional algebra in the presence of floating point arithmetic. Exercise 7 shows that Cauchy's inequality can fail even in the simple case $n = 2$, $x_1 = x_2 = 1$. Novice programmers who calculate the standard deviation of some observations by using the textbook formula

$$\sigma = \sqrt{\left(n \sum_{1 \leq k \leq n} x_k^2 - \left(\sum_{1 \leq k \leq n} x_k \right)^2 \right) / n(n-1)} \quad (14)$$

often find themselves taking the square root of a negative number! A much better way to calculate means and standard deviations with floating point arithmetic is to use the recurrence formulas

$$M_1 = x_1, \quad M_k = M_{k-1} \oplus (x_k \ominus M_{k-1}) \oslash k, \quad (15)$$

$$S_1 = 0, \quad S_k = S_{k-1} \oplus (x_k \ominus M_{k-1}) \otimes (x_k \ominus M_k), \quad (16)$$

for $2 \leq k \leq n$, where $\sigma = \sqrt{S_n / (n-1)}$. [Cf. B. P. Welford, *Technometrics* 4 (1962), 419–420.] With this method S_n can never be negative, and we avoid other serious problems encountered by the naïve method of accumulating sums, as shown in exercise 16. (See exercise 19 for a summation technique that provides an even better guarantee on the accuracy.)

Although algebraic laws do not always hold exactly, we can often show that they aren't too far off base. When $b^{e-1} \leq x < b^e$ we have $\text{round}(x) = x + \rho(x)$, where $|\rho(x)| \leq \frac{1}{2}b^{e-p}$; hence

$$\text{round}(x) = x(1 + \delta(x)), \quad (17)$$

where the relative error is bounded independently of x :

$$|\delta(x)| \leq \frac{1}{2}/(b^{1-p} + \frac{1}{2}) < \frac{1}{2}b^{1-p}. \quad (18)$$

We can use this inequality to estimate the relative error of normalized floating point calculations in a simple way, since $u \oplus v = (u + v)(1 + \delta(u + v))$, etc.

As an example of typical error-estimation procedures, let us consider the associative law for multiplication. Exercise 3 shows that $(u \otimes v) \otimes w$ is not in general equal to $u \otimes (v \otimes w)$; but the situation in this case is much better than it was with respect to the associative law of addition (1) and the distributive law (12). In fact, we have

$$(u \otimes v) \otimes w = ((uv)(1 + \delta_1)) \otimes w = uvw(1 + \delta_1)(1 + \delta_2),$$

$$u \otimes (v \otimes w) = u \otimes ((vw)(1 + \delta_3)) = uvw(1 + \delta_3)(1 + \delta_4),$$

for some $\delta_1, \delta_2, \delta_3, \delta_4$, provided that no exponent underflow or overflow occurs, where $|\delta_j| < \frac{1}{2}b^{1-p}$ for each j . Hence

$$\frac{(u \otimes v) \otimes w}{u \otimes (v \otimes w)} = \frac{(1 + \delta_1)(1 + \delta_2)}{(1 + \delta_3)(1 + \delta_4)} = 1 + \delta,$$

where

$$|\delta| < 2b^{1-p}/(1 - \frac{1}{2}b^{1-p})^2. \quad (19)$$

The number b^{1-p} occurs so often in such analyses, it has been given a special name, one *ulp*, meaning one “unit in the last place” of the fraction part. Floating point operations are correct to within half an ulp, and the calculation of uvw by two floating point multiplications will be correct within about one ulp (ignoring second-order terms). Hence the associative law for multiplication holds to within about two ulps of relative error.

We have shown that $(u \otimes v) \otimes w$ is approximately equal to $u \otimes (v \otimes w)$, except when exponent overflow or underflow is a problem. It is worthwhile to study this intuitive idea of being “approximately equal” in more detail; can we make such a statement more precise in a reasonable way?

A programmer using floating point arithmetic almost never wants to test if two computed values are exactly equal to each other (or at least he hardly ever should try to do so), because this is an extremely improbable occurrence. For example, if a recurrence relation

$$x_{n+1} = f(x_n)$$

is being used, where the theory in some textbook says that x_n approaches a limit as $n \rightarrow \infty$, it is usually a mistake to wait until $x_{n+1} = x_n$ for some n , since the sequence x_n might be periodic with a longer period due to the rounding of intermediate results. The proper procedure is to wait until $|x_{n+1} - x_n| < \delta$, for some suitably chosen number δ ; but since we don't necessarily know the order of magnitude of x_n in advance, it is even better to wait until

$$|x_{n+1} - x_n| \leq \epsilon|x_n|; \quad (20)$$

now ϵ is a number that is much easier to select. This relation (20) is another way of saying that x_{n+1} and x_n are approximately equal; and our discussion

indicates that a relation of "approximately equal" would be more useful than the traditional relation of equality, when floating point computations are involved, if we could only define a suitable approximation relation.

In other words, the fact that strict equality of floating point values is of little importance implies that we ought to have a new operation, *floating point comparison*, which is intended to help assess the relative values of two floating point quantities. The following definitions seem to be appropriate for base b , excess q , floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$:

$$u \prec v \quad (\epsilon) \quad \text{if and only if} \quad v - u > \epsilon \max(b^{e_u-q}, b^{e_v-q}); \quad (21)$$

$$u \sim v \quad (\epsilon) \quad \text{if and only if} \quad |v - u| \leq \epsilon \max(b^{e_u-q}, b^{e_v-q}); \quad (22)$$

$$u \succ v \quad (\epsilon) \quad \text{if and only if} \quad u - v > \epsilon \max(b^{e_u-q}, b^{e_v-q}); \quad (23)$$

$$u \approx v \quad (\epsilon) \quad \text{if and only if} \quad |v - u| \leq \epsilon \min(b^{e_u-q}, b^{e_v-q}). \quad (24)$$

These definitions apply to unnormalized values as well as to normalized ones. Note that exactly one of the conditions $u \prec v$ (definitely less than), $u \sim v$ (approximately equal to), or $u \succ v$ (definitely greater than) must always hold for any given pair of values u and v . The relation $u \approx v$ is somewhat stronger than $u \sim v$, and it might be read "u is essentially equal to v." All of the relations are given in terms of a positive real number ϵ that measures the degree of approximation being considered.

One way to view the above definitions is to associate a "neighborhood" set $N(u) = \{x \mid |x - u| \leq \epsilon b^{e_u-q}\}$ with each floating point number u ; thus, $N(u)$ represents a set of values near u based on the exponent of u 's floating point representation. In these terms, we have $u \prec v$ if and only if $N(u) < v$ and $u < N(v)$; $u \sim v$ if and only if $u \in N(v)$ or $v \in N(u)$; $u \succ v$ if and only if $u > N(v)$ and $N(u) > v$; $u \approx v$ if and only if $u \in N(v)$ and $v \in N(u)$. (Here we are assuming that the parameter ϵ , which measures the degree of approximation, is a constant; a more complete notation would indicate the dependence of $N(u)$ upon ϵ .)

Here are some simple consequences of the above definitions:

$$\text{if } u \prec v \quad (\epsilon) \quad \text{then } v \succ u \quad (\epsilon); \quad (25)$$

$$\text{if } u \approx v \quad (\epsilon) \quad \text{then } u \sim v \quad (\epsilon); \quad (26)$$

$$u \approx u \quad (\epsilon); \quad (27)$$

$$\text{if } u \prec v \quad (\epsilon) \quad \text{then } u < v; \quad (28)$$

$$\text{if } u \prec v \quad (\epsilon_1) \text{ and } \epsilon_1 \geq \epsilon_2 \quad \text{then } u \prec v \quad (\epsilon_2); \quad (29)$$

$$\text{if } u \sim v \quad (\epsilon_1) \text{ and } \epsilon_1 \leq \epsilon_2 \quad \text{then } u \sim v \quad (\epsilon_2); \quad (30)$$

$$\text{if } u \approx v \quad (\epsilon_1) \text{ and } \epsilon_1 \leq \epsilon_2 \quad \text{then } u \approx v \quad (\epsilon_2); \quad (31)$$

$$\text{if } u \prec v \quad (\epsilon_1) \text{ and } v \prec w \quad (\epsilon_2) \quad \text{then } u \prec w \quad (\epsilon_1 + \epsilon_2); \quad (32)$$

$$\text{if } u \approx v \quad (\epsilon_1) \text{ and } v \approx w \quad (\epsilon_2) \quad \text{then } u \sim w \quad (\epsilon_1 + \epsilon_2). \quad (33)$$

Moreover, we can prove without difficulty that

$$|u - v| \leq \epsilon|u| \quad \text{and} \quad |u - v| \leq \epsilon|v| \quad \text{implies} \quad u \approx v \quad (\epsilon); \quad (34)$$

$$|u - v| \leq \epsilon|u| \quad \text{or} \quad |u - v| \leq \epsilon|v| \quad \text{implies} \quad u \sim v \quad (\epsilon); \quad (35)$$

and conversely, for *normalized* floating point numbers u and v , when $\epsilon < 1$,

$$u \approx v \quad (\epsilon) \quad \text{implies} \quad |u - v| \leq b\epsilon|u| \quad \text{and} \quad |u - v| \leq b\epsilon|v|; \quad (36)$$

$$u \sim v \quad (\epsilon) \quad \text{implies} \quad |u - v| \leq b\epsilon|u| \quad \text{or} \quad |u - v| \leq b\epsilon|v|. \quad (37)$$

Let $\epsilon_0 = b^{1-p}$ be one ulp. The derivation of (17) establishes the inequality $|x - \text{round}(x)| = |\rho(x)| < \frac{1}{2}\epsilon_0 \min(|x|, |\text{round}(x)|)$, hence

$$x \approx \text{round}(x) \quad (\frac{1}{2}\epsilon_0); \quad (38)$$

it follows that $u \oplus v \approx u + v$ ($\frac{1}{2}\epsilon_0$), etc. The approximate associative law for multiplication derived above can be recast as follows: We have

$$|(u \otimes v) \otimes w - u \otimes (v \otimes w)| < \frac{2\epsilon_0}{(1 - \frac{1}{2}\epsilon_0)^2} |u \otimes (v \otimes w)|$$

by (19), and the same inequality is valid with $(u \otimes v) \otimes w$ and $u \otimes (v \otimes w)$ interchanged. Hence by (34),

$$(u \otimes v) \otimes w \approx u \otimes (v \otimes w) \quad (\epsilon) \quad (39)$$

whenever $\epsilon \geq 2\epsilon_0/(1 - \frac{1}{2}\epsilon_0)^2$. For example, if $b = 10$ and $p = 8$ we may take $\epsilon = 0.00000021$.

The relations \prec , \sim , \succ , and \approx are useful within numerical algorithms, and it is therefore a good idea to provide routines for comparing floating point numbers as well as for doing arithmetic on them.

Let us now shift our attention back to the question of finding exact relations that are satisfied by the floating point operations. It is interesting to note that floating point addition and subtraction are not completely intractable from an axiomatic standpoint, since they do satisfy the nontrivial identities stated in the following theorems.

Theorem A. *Let u and v be normalized floating point numbers. Then*

$$((u \oplus v) \ominus u) + ((u \oplus v) \ominus ((u \oplus v) \ominus u)) = u \oplus v, \quad (40)$$

provided that no exponent overflow or underflow occurs.

This rather cumbersome-looking identity can be rewritten in a simpler manner:
Let

$$\begin{aligned} u' &= (u \oplus v) \ominus v, & v' &= (u \oplus v) \ominus u; \\ u'' &= (u \oplus v) \ominus v', & v'' &= (u \oplus v) \ominus u'. \end{aligned} \quad (41)$$

Intuitively, u' and u'' should be approximations to u , and v' and v'' should be approximations to v . Theorem A tells us that

$$u \oplus v = u' + v'' = u'' + v'. \quad (42)$$

This is a stronger statement than the identity

$$u \oplus v = u' \oplus v'' = u'' \oplus v', \quad (43)$$

which follows by rounding (42).

Proof. Let us say that t is a *tail* of x modulo b^e if

$$t \equiv x \pmod{b^e}, \quad |t| \leq \frac{1}{2}b^e; \quad (44)$$

thus, $x - \text{round}(x)$ is always a tail of x . The proof of Theorem A rests largely on the following simple fact proved in exercise 11:

Lemma T. *If t is a tail of the floating point number x , then $x \ominus t = x - t$.* ■

Let $w = u \oplus v$. Theorem A holds trivially when $w = 0$. By multiplying all variables by a suitable power of b , we may assume without loss of generality that $e_w = p$. Then $u + v = w + r$, where r is a tail of $u + v$ modulo 1. Furthermore $u' = \text{round}(w - v) = \text{round}(u - r) = u - r - t$, where t is a tail of $u - r$ modulo b^e and $e = e_{u'} - p$.

If $e \leq 0$, then $t \equiv u - r \equiv -v \pmod{b^e}$, hence t is a tail of $-v$ and $v'' = \text{round}(w - u') = \text{round}(v + t) = v + t$; this proves (40). If $e > 0$, then $|u - r| \geq b^p - \frac{1}{2}$; and since $|r| \leq \frac{1}{2}$, we have $|u| \geq b^p - 1$. It follows that r is a tail of v modulo 1. If $u' = u$, we have $v'' = \text{round}(w - u) = \text{round}(v - r) = v - r$. Otherwise the relation $\text{round}(u - r) \neq u$ implies that $|u| = b^p - 1$, $|r| = \frac{1}{2}$, $|u'| = b^p$; we have $v'' = \text{round}(w - u') = \text{round}(v + r) = v + r$, because r is also a tail of $-v$ in this case. ■

Theorem A exhibits a regularity property of floating point addition, but it doesn't seem to be an especially useful result. The following identity is more significant:

Theorem B. *Under the hypotheses of Theorem A and (41),*

$$u + v = (u \oplus v) + ((u \ominus u') \oplus (v \ominus v')). \quad (45)$$

Proof. In fact, we can show that $u \ominus u' = u - u'$, $v \ominus v'' = v - v''$, and $(u - u') \oplus (v - v'') = (u - u') + (v - v'')$, hence (45) will follow from Theorem A.

Using the notation of the preceding proof, these relations are respectively equivalent to

$$\text{round}(t + r) = t + r, \quad \text{round}(t) = t, \quad \text{round}(r) = r. \quad (46)$$

Exercise 12 establishes the theorem in the special case $|e_u - e_v| \geq p$. Otherwise $u + v$ has at most $2p$ significant digits and it is easy to see that $\text{round}(r) = r$. If now $e > 0$, the proof of Theorem A shows that $t = -r$ or $t = r = \pm \frac{1}{2}$. If $e \leq 0$ we have $t + r \equiv u$ and $t \equiv -v$ (modulo b^e); this is enough to prove that $t + r$ and r round to themselves, provided that $e_u \geq e$ and $e_v \geq e$. But either $e_u < 0$ or $e_v < 0$ would contradict our hypothesis that $|e_u - e_v| < p$, since $e_w = p$. ■

Theorem B gives an explicit formula for the difference between $u + v$ and $u \oplus v$, in terms of quantities that can be calculated directly using five operations of floating point arithmetic. If the radix b is 2 or 3, we can improve on this result, obtaining the exact value of the correction term with only two floating point operations and one (fixed point) comparison of absolute values:

Theorem C. *If $b \leq 3$ and $|u| \geq |v|$, then*

$$u + v = (u \oplus v) + (u \ominus (u \oplus v)) \oplus v. \quad (47)$$

Proof. Following the conventions of preceding proofs again, we wish to show that $v \ominus v' = r$. It suffices to show that $v' = w - u$, because (46) will then yield $v \ominus v' = \text{round}(v - v') = \text{round}(u + v - w) = \text{round}(r) = r$.

We shall in fact prove (47) whenever $b \leq 3$ and $e_u \geq e_v$. If $e_u \geq p$, then r is a tail of v modulo 1, hence $v' = w \ominus u = v \ominus r = v - r = w - u$ as desired. If $e_u < p$, then we must have $e_u = p - 1$, and $w - u$ is a multiple of b^{-1} ; it will therefore round to itself if its magnitude is less than $b^{p-1} + b^{-1}$. Since $b \leq 3$, we have indeed $|w - u| \leq |w - u - v| + |v| \leq \frac{1}{2} + (b^{p-1} - b^{-1}) < b^{p-1} + b^{-1}$. This completes the proof. ■

The proofs of Theorems A, B, and C do not rely on the precise definitions of $\text{round}(x)$ in the ambiguous cases when x is exactly midway between consecutive floating point numbers; any way of resolving the ambiguity will suffice for the validity of everything we have proved so far.

No rounding rule can be best for every application. For example, we generally want a special rule when computing our income tax. But for most numerical calculations the best policy appears to be the rounding scheme specified in Algorithm 4.2.1N, which insists that the least significant digit should always be made even (or always odd) when an ambiguous value is rounded. This is not a trivial technicality, of interest only to nit-pickers; it is an important practical consideration, since the ambiguous case arises surprisingly often and a biased rounding rule produces significantly poor results. For example, consider decimal arithmetic and assume that remainders of 5 are always rounded upwards. Then if $u = 1.0000000$ and $v = 0.55555555$ we have $u \oplus v = 1.5555556$; and if

we floating-subtract v from this result we get $u' = 1.0000001$. Adding and subtracting v from u' gives 1.0000002 , and the next time we get 1.0000003 , etc.; the result keeps growing although we are adding and subtracting the same value.

This phenomenon, called *drift*, will not occur when we use a stable rounding rule based on the parity of the least significant digit. More precisely:

Theorem D. $((u \oplus v) \ominus v) \oplus v = (u \oplus v) \ominus v$.

For example, if $u = 1.2345679$ and $v = -0.23456785$, we find $u \oplus v = 1.0000000$, $(u \oplus v) \ominus v = 1.2345678$, $((u \oplus v) \ominus v) \oplus v = 0.99999995$, and $((((u \oplus v) \ominus v) \oplus v) \ominus v = 1.2345678$. The proof for general u and v seems to require a case analysis even more detailed than that in the above theorems; see the references at the end of this section. ■

Theorem D is valid both for “round to even” and “round to odd”; how should we choose between these possibilities? When the radix b is odd, ambiguous cases never arise except during floating point division, and the rounding in such cases is comparatively unimportant. For even radices, there is reason to prefer the following rule: “Round to even when $b/2$ is odd, round to odd when $b/2$ is even.” The least significant digit of a floating point fraction occurs frequently as a remainder to be rounded off in subsequent calculations, and this rule avoids generating the digit $b/2$ in the least significant position whenever possible; its effect is to provide some memory of an ambiguous rounding so that subsequent rounding will tend to be unambiguous. For example, if we were to round to odd in the decimal system, repeated rounding of the number 2.44445 to one less place each time leads to the sequence 2.4445, 2.445, 2.45, 2.5, 3; but if we round to even, such situations do not occur. [Roy A. Keir, *Inf. Proc. Letters* 3 (1975), 188–189.] On the other hand, some people prefer rounding to even in all cases, so that the remainder will tend to be 0 more often. Neither alternative conclusively dominates the other; fortunately the base is usually $b = 2$ or $b = 10$, when everyone agrees that round-to-even is best.

A reader who has checked some of the details of the above proofs will realize the immense simplification that has been afforded by the simple rule $u \oplus v = \text{round}(u + v)$. If our floating point addition routine would fail to give this result even in a few rare cases, the proofs would become enormously more complicated and perhaps they would even break down completely.

Theorem B fails if truncation arithmetic is used in place of rounding, i.e., if we let $u \oplus v = \text{trunc}(u + v)$ and $u \ominus v = \text{trunc}(u - v)$, where $\text{trunc}(x)$ takes all positive real x into the largest floating point number $\leq x$. An exception to Theorem B would then occur for cases such as $(20, +.10000001) \oplus (10, -.10000001) = (20, +.10000000)$, when the difference between $u + v$ and $u \oplus v$ cannot be expressed exactly as a floating point number; and also for cases such as $12345678 \oplus .012345678$, when it can be.

Many people feel that, since floating point arithmetic is inexact by nature, there is no harm in making it just a little bit less exact in certain rather rare cases, if it is convenient to do so. This policy saves a few cents in the design of computer

hardware, or a small percentage of the average running time of a subroutine. But the above discussion shows that such a policy is mistaken. We could save about five percent of the running time of the FADD subroutine, Program 4.2.1A, and about 25 percent of its space, if we took the liberty of rounding incorrectly in a few cases, but we are much better off leaving it as it is. The reason is not to glorify “bit chasing”; a more fundamental issue is at stake here: *Numerical subroutines should deliver results that satisfy simple, useful mathematical laws whenever possible.* The crucial formula $u \oplus v = \text{round}(u + v)$ is a “regularity” property that makes a great deal of difference between whether mathematical analysis of computational algorithms is worth doing or worth avoiding. Without any underlying symmetry properties, the job of proving interesting results becomes extremely unpleasant. *The enjoyment of one's tools is an essential ingredient of successful work.*

B. Unnormalized floating point arithmetic. The policy of normalizing all floating point numbers may be construed in two ways: We may look on it favorably by saying that it is an attempt to get the maximum possible accuracy obtainable with a given degree of precision, or we may consider it to be potentially dangerous since it tends to imply that the results are more accurate than they really are. When we normalize the result of $(1, +.31428571) \ominus (1, +.31415927)$ to $(-2, +.12644000)$, we are suppressing information about the possibly greater inaccuracy of the latter quantity. Such information would be retained if the answer were left as $(1, +.00012644)$.

The input data to a problem is frequently not known as precisely as the floating point representation allows. For example, the values of Avogadro's number and Planck's constant are not known to eight significant digits, and it might be more appropriate to denote them, respectively, by

$$(27, +.00060225) \quad \text{and} \quad (-23, +.00010545)$$

instead of by $(24, +.60225200)$ and $(-26, +.10545000)$. It would be nice if we could give our input data for each problem in an unnormalized form that expresses how much precision is assumed, and if the output would indicate just how much precision is known in the answer. Unfortunately, this is a terribly difficult problem, although the use of unnormalized arithmetic can help to give some indication. For example, we can say with a fair degree of certainty that the product of Avogadro's number by Planck's constant is $(0, +.00063507)$, and that their sum is $(27, +.00060225)$. (The purpose of this example is not to suggest that any important physical significance should be attached to the sum and product of these fundamental constants; the point is that it is possible to preserve a little of the information about precision in the result of calculations with imprecise quantities, when the original operands are independent of each other.)

The rules for unnormalized arithmetic are simply this: Let l_u be the number of leading zeros in the fraction part of $u = (e_u, f_u)$, so that l_u is the largest integer $\leq p$ with $|f_u| < b^{-l_u}$. Then addition and subtraction are performed

just as in Algorithm 4.2.1A, except that all scaling to the left is suppressed. Multiplication and division are performed as in Algorithm 4.2.1M, except that the answer is scaled right or left so that precisely $\max(l_u, l_v)$ leading zeros appear. Essentially the same rules have been used in manual calculation for many years.

It follows that, for unnormalized computations,

$$e_{u \oplus v}, e_{u \ominus v} = \max(e_u, e_v) + (0 \text{ or } 1) \quad (48)$$

$$e_{u \otimes v} = e_u + e_v - q - \min(l_u, l_v) - (0 \text{ or } 1) \quad (49)$$

$$e_{u \oslash v} = e_u - e_v + q - l_u + l_v + \max(l_u, l_v) + (0 \text{ or } 1). \quad (50)$$

When the result of a calculation is zero, an unnormalized zero (often called an "order of magnitude zero") is given as the answer; this indicates that the answer may not truly be zero, we just don't know any of its significant digits.

Error analysis takes a somewhat different form with unnormalized floating point arithmetic. Let us define

$$\delta_u = \frac{1}{2}b^{e_u-q-p} \quad \text{if } u = (e_u, f_u). \quad (51)$$

This quantity depends on the representation of u , not just on the value $b^{e_u-q}f_u$. Our rounding rule tells us that

$$|u \oplus v - (u + v)| \leq \delta_{u \oplus v}, \quad |u \ominus v - (u - v)| \leq \delta_{u \ominus v},$$

$$|u \otimes v - (u \times v)| \leq \delta_{u \otimes v}, \quad |u \oslash v - (u / v)| \leq \delta_{u \oslash v}.$$

These inequalities apply to normalized as well as unnormalized arithmetic; the main difference between the two types of error analysis is the definition of the exponent of the result of each operation (Eqs. (48) to (50)).

We have remarked that the relations \prec , \sim , \succ , and \approx defined earlier in this section are valid and meaningful for unnormalized numbers as well as for normalized numbers. As an example of the use of these relations, let us prove an approximate associative law for unnormalized addition, analogous to (39):

$$(u \oplus v) \oplus w \approx u \oplus (v \oplus w) \quad (\epsilon), \quad (52)$$

for suitable ϵ . We have

$$\begin{aligned} |(u \oplus v) \oplus w - (u + v + w)| &\leq |(u \oplus v) \oplus w - ((u \oplus v) + w)| \\ &\quad + |u \oplus v - (u + v)| \\ &\leq \delta_{(u \oplus v) \oplus w} + \delta_{u \oplus v} \\ &\leq 2\delta_{(u \oplus v) \oplus w}. \end{aligned}$$

A similar formula holds for $|u \oplus (v \oplus w) - (u + v + w)|$. Now since $e_{(u \oplus v) \oplus w} = \max(e_u, e_v, e_w) + (0, 1, \text{ or } 2)$, we have $\delta_{(u \oplus v) \oplus w} \leq b^2 \delta_{u \oplus (v \oplus w)}$. Therefore we

find that (52) is valid when $\epsilon \geq 2b^{2-p}$; unnormalized addition is not as erratic as normalized addition with respect to the associative law.

It should be emphasized that unnormalized arithmetic is by no means a panacea. There are examples where it indicates greater accuracy than is present (e.g., addition of a great many small quantities of about the same magnitude, or evaluation of x^n for large n); and there are many more examples when it indicates poor accuracy while normalized arithmetic actually does produce good results. There is an important reason why no straightforward one-operation-at-a-time method of error analysis can be completely satisfactory, namely the fact that operands are usually not independent of each other. This means that errors tend to cancel or reinforce each other in strange ways. For example, suppose that x is approximately $\frac{1}{2}$, and suppose that we have an approximation $y = x + \delta$ with absolute error δ . If we now wish to compute $x(1 - x)$, we can form $y(1 - y)$; if $x = \frac{1}{2} + \epsilon$ we find $y(1 - y) = x(1 - x) - 2\epsilon\delta - \delta^2$, so the error has decreased substantially: it has been multiplied by a factor of $2\epsilon + \delta$. This is just one case where multiplication of imprecise quantities can lead to a quite accurate result when the operands are not independent of each other. A more obvious example is the computation of $x \ominus x$, which can be obtained with perfect accuracy regardless of how bad an approximation to x we begin with.

The extra information that unnormalized arithmetic gives us can often be more important than the information it destroys during an extended calculation, but (as usual) we must use it with care. Examples of the proper use of unnormalized arithmetic are discussed by R. L. Ashenhurst and N. Metropolis in *Computers and Computing, AMM Slaught Memorial Papers* **10** (February, 1965), 47–59; by N. Metropolis in *Numer. Math.* **7** (1965), 104–112; and by R. L. Ashenhurst in *Error in Digital Computation* **2**, ed. by L. B. Rall (New York: Wiley, 1965), 3–37. Appropriate methods for computing standard mathematical functions with both input and output in unnormalized form are given by R. L. Ashenhurst in *JACM* **11** (1964), 168–187. An extension of unnormalized arithmetic, which remembers that certain values are known to be exact, has been discussed by N. Metropolis in *IEEE Trans. C-22* (1973), 573–576.

C. Interval arithmetic. Another approach to the problem of error determination is the so-called interval or range arithmetic, in which upper and lower bounds on each number are maintained during the calculations. Thus, for example, if we know that $u_0 \leq u \leq u_1$ and $v_0 \leq v \leq v_1$, we represent this by the interval notation $u = [u_0, u_1]$, $v = [v_0, v_1]$. The sum $u \oplus v$ is $[u_0 \nabla v_0, u_1 \Delta v_1]$, where ∇ denotes “lower floating point addition,” the greatest representable number less than or equal to the true sum, and Δ is defined similarly (see exercise 4.2.1–13). Furthermore $u \ominus v = [u_0 \nabla v_1, u_1 \Delta v_0]$; and if u_0 and v_0 are positive, we have $u \otimes v = [u_0 \nabla v_0, u_1 \Delta v_1]$, $u \oslash v = [u_0 \nabla v_1, u_1 \Delta v_0]$. For example, we might represent Avogadro’s number and Planck’s constant as

$$N = [(24, +.60222400), (24, +.60228000)],$$

$$\hbar = [(-26, +.10544300), (-26, +.10545700)];$$

their sum and product would then turn out to be

$$N \oplus h = [(24, +.60222400), (24, +.60228001)],$$

$$N \otimes h = [(-3, +.63500305), (-3, +.63514642)].$$

If we try to divide by $[v_0, v_1]$ when $v_0 < 0 < v_1$, there is a possibility of division by zero. Since the philosophy underlying interval arithmetic is to provide rigorous error estimates, a divide-by-zero error should be signalled in this case. However, overflow and underflow need not be treated as errors in interval arithmetic, if special conventions are introduced as discussed in exercise 24.

Interval arithmetic takes only about twice as long as ordinary arithmetic, and it provides truly reliable error estimates. Considering the difficulty of mathematical error analyses, this is indeed a small price to pay. Since the intermediate values in a calculation often depend on each other, as explained above, the final estimates obtained with interval arithmetic will tend to be pessimistic; and iterative numerical methods often have to be redesigned if we want to deal with intervals. The prospects for effective use of interval arithmetic look very good, however, and efforts should be made to increase its availability.

D. History and bibliography. Jules Tannery's classic treatise on decimal calculations, *Leçons d'Arithmétique* (Paris: Colin, 1894), stated that positive numbers should be rounded upwards if the first discarded digit is 5 or more; since exactly half of the decimal digits are 5 or more, he felt that this rule would round upwards exactly half of the time, on the average, so it would produce compensating errors. The idea of "round to even" in the ambiguous cases seems to have been mentioned first by James B. Scarborough in the first edition of his pioneering book *Numerical Mathematical Analysis* (Baltimore: Johns Hopkins Press, 1930), p. 2; in the second (1950) edition he amplified his earlier remarks, stating that "It should be obvious to any thinking person that when a 5 is cut off, the preceding digit should be increased by 1 in only *half* the cases," and he recommended round-to-even in order to achieve this.

The first analysis of floating point arithmetic was given by F. L. Bauer and K. Samelson, *Zeitschrift für angewandte Math. und Physik* 4 (1953), 312–316. The next publication was not until over five years later: J. W. Carr III, *CACM* 2, 5 (May 1959), 10–15. See also P. C. Fischer, *Proc. ACM Nat. Meeting* 13 (Urbana, Illinois, 1958), paper 39. The book *Rounding Errors in Algebraic Processes* (Englewood Cliffs: Prentice-Hall, 1963), by J. H. Wilkinson, shows how to apply error analysis of the individual arithmetic operations to the error analysis of large-scale problems; see also his treatise on *The Algebraic Eigenvalue Problem* (Oxford: Clarendon Press, 1965).

More recent work on floating point accuracy is summarized in two important papers that can be especially recommended for further study: W. M. Kahan, *Proc. IFIP Congress* (1971), 2, 1214–1239; R. P. Brent, *IEEE Trans. C-22* (1973), 601–607. Both papers include useful theory and demonstrate that it pays off in practice.

The relations \prec , \sim , \succ , \approx introduced in this section are similar to ideas published by A. van Wijngaarden in *BIT* 6 (1966), 66–81. Theorems A and B above were inspired by some related work of Ole Møller, *BIT* 5 (1965), 37–50, 251–255; Theorem C is due to T. J. Dekker, *Numer. Math.* 18 (1971), 224–242. Extensions and refinements of all three theorems have been published by S. Linnainmaa, *BIT* 14 (1974), 167–202. W. M. Kahan introduced Theorem D in some unpublished notes; for a complete proof and further commentary, see J. F. Reiser and D. E. Knuth, *Inf. Proc. Letters* 3 (1975), 84–87, 164.

Unnormalized floating point arithmetic was recommended by F. L. Bauer and K. Samelson in the article cited above, and it was independently used by J. W. Carr III at the University of Michigan in 1953. Several years later, the MANIAC III computer was designed to include both kinds of arithmetic in its hardware; see R. L. Ashenhurst and N. Metropolis, *JACM* 6 (1959), 415–428, *IEEE Trans. EC-12* (1963), 896–901; R. L. Ashenhurst, *Proc. Spring Joint Computer Conf.* 21 (1962), 195–202. See also H. L. Gray and C. Harrison, Jr., *Proc. Eastern Joint Computer Conf.* 16 (1959), 244–248, and W. G. Wadey, *JACM* 7 (1960), 129–139, for further early discussions of unnormalized arithmetic.

For early developments in interval arithmetic, and some modifications, see A. Gibb, *CACM* 4 (1961), 319–320; B. A. Chartres, *JACM* 13 (1966), 386–403; and the book *Interval Analysis* by Ramon E. Moore (Prentice Hall, 1966). The subsequent flourishing of this subject is described in Moore's later book, *Methods and Applications of Interval Analysis* (SIAM, 1979).

The book *Grundlagen des Numerischen Rechnens: Mathematische Begründung der Rechenarithmetik* by Ulrich Kulisch (Mannheim: Bibl. Inst., 1976) is entirely devoted to the study of floating point arithmetic systems; see also Kulisch's article in *IEEE Trans. C-26* (1977), 610–621, and his more recent book written jointly with W. L. Miranker, entitled *Computer Arithmetic in Theory and Practice* (New York: Academic Press, 1980).

EXERCISES

Note: Normalized floating point arithmetic is assumed unless the contrary is specified.

1. [M18] Prove that identity (7) is a consequence of (2) through (6).
2. [M20] Use identities (2) through (8) to prove that $(u \oplus x) \oplus (v \oplus y) \geq u \oplus v$ whenever $x \geq 0$ and $y \geq 0$.
3. [M20] Find eight-digit floating decimal numbers u , v , and w such that

$$u \otimes (v \otimes w) \neq (u \otimes v) \otimes w,$$

and such that no exponent overflow or underflow occurs during the computations.

4. [10] Is it possible to have floating point numbers u , v , and w for which exponent overflow occurs during the calculation of $u \otimes (v \otimes w)$ but not during the calculation of $(u \otimes v) \otimes w$?

5. [M20] Is $u \oslash v = u \otimes (1 \oslash v)$ an identity, for all floating point numbers u and $v \neq 0$ such that no exponent overflow or underflow occurs?
6. [M22] Are either of the following two identities valid for all floating point numbers u ? (a) $0 \ominus (0 \ominus u) = u$; (b) $1 \oslash (1 \oslash u) = u$.
7. [M21] Let u^{\circledast} stand for $u \otimes u$. Find floating binary numbers u and v such that $2(u^{\circledast} + v^{\circledast}) < (u \oplus v)^{\circledast}$.
- 8. [20] Let $\epsilon = 0.0001$; which of the relations

$$u \prec v \quad (\epsilon), \quad u \sim v \quad (\epsilon), \quad u \succ v \quad (\epsilon), \quad u \approx v \quad (\epsilon)$$

hold for the following pairs of base 10, excess 0, eight-digit floating point numbers?

- a) $u = (1, +.31415927)$, $v = (1, +.31416000)$;
 b) $u = (0, +.99997000)$, $v = (1, +.10000039)$;
 c) $u = (24, +.60225200)$, $v = (27, +.00060225)$;
 d) $u = (24, +.60225200)$, $v = (31, +.00000006)$;
 e) $u = (24, +.60225200)$, $v = (32, +.00000000)$.
9. [M22] Prove (33), and explain why the conclusion cannot be strengthened to the relation $u \approx w (\epsilon_1 + \epsilon_2)$.
- 10. [M25] (W. M. Kahan.) A certain computer performs floating point arithmetic without proper rounding, and, in fact, its floating point multiplication routine ignores all but the first p most significant digits of the $2p$ -digit product $f_u f_v$. (Thus when $f_u f_v < 1/b$, the least-significant digit of $u \otimes v$ always comes out to be zero, due to subsequent normalization.) Show that this causes the monotonicity of multiplication to fail; i.e., there are positive normalized floating point numbers u , v , w such that $u < v$ but $u \otimes w > v \otimes w$.
11. [M20] Prove Lemma T.
12. [M24] Carry out the proof of Theorem B and (46) when $|e_u - e_v| \geq p$.
- 13. [M25] Some programming languages (and even some computers) make use of floating point arithmetic only, with no provision for exact calculations with integers. If operations on integers are desired, we can, of course, represent an integer as a floating point number; and when the floating point operations satisfy the basic definitions in (9), we know that all floating point operations will be exact, provided that the operands and the answer can each be represented exactly with p significant digits. Therefore—so long as we know that the numbers aren't too large—we can add, subtract, or multiply integers with no inaccuracy due to rounding errors.

But suppose that a programmer wants to determine if m is an exact multiple of n , when m and $n \neq 0$ are integers. Suppose further that a subroutine is available to calculate the quantity $\text{round}(u \bmod 1) = u \bmod 1$ for any given floating point number u , as in exercise 4.2.1–15. One good way to determine whether or not m is a multiple of n might be to test whether or not $(m \oslash n) \bmod 1 = 0$, using the assumed subroutine; but perhaps rounding errors in the floating point calculations will invalidate this test in certain cases.

Find suitable conditions on the range of integer values $n \neq 0$ and m , such that m is a multiple of n if and only if $(m \oslash n) \bmod 1 = 0$. In other words, show that if m and n are not too large, this test is valid.

14. [M27] Find a suitable ϵ such that $(u \otimes v) \otimes w \approx u \otimes (v \otimes w)$ (ϵ), when unnormalized multiplication is being used. (This generalizes (39), since unnormalized multiplication is exactly the same as normalized multiplication when the input operands u , v , and w are normalized.)

► 15. [M24] (H. Björk.) Does the computed midpoint of an interval always lie between the endpoints? (In other words, does $u \leq v$ imply that $u \leq (u \oplus v) \oslash 2 \leq v$?)

16. [M28] (a) What is $(\cdots((x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_n)$ when $n = 10^6$ and $x_k = 1.1111111$ for all k , using eight-digit floating decimal arithmetic? (b) What happens when Eq. (14) is used to calculate the standard deviation of these particular values x_k ? What happens when Eqs. (15) and (16) are used instead? (c) Prove that $S_k \geq 0$ in (16), for all choices of x_1, \dots, x_k .

17. [28] Write a MIX subroutine, FCMP, that compares the floating point number u in location ACC with the floating point number v in register A, and that sets the comparison indicator to LESS, EQUAL, or GREATER, according as $u < v$, $u \sim v$, or $u > v$ (ϵ); here ϵ is stored in location EPSILON as a nonnegative fixed point quantity with the decimal point assumed at the left of the word. Assume normalized inputs.

18. [M40] In unnormalized arithmetic is there a suitable number ϵ such that

$$u \otimes (v \oplus w) \approx (u \otimes v) \oplus (u \otimes w) \quad (\epsilon)?$$

► 19. [M30] (W. M. Kahan.) Consider the following procedure for floating point summation of x_1, \dots, x_n :

$$s_0 = c_0 = 0;$$

$$y_k = x_k \ominus c_{k-1}, \quad s_k = s_{k-1} \oplus y_k, \quad c_k = (s_k \ominus s_{k-1}) \ominus y_k, \quad \text{for } 1 \leq k \leq n.$$

Let the relative errors in these operations be defined by the equations

$$\begin{aligned} y_k &= (x_k - c_{k-1})(1 + \eta_k), & s_k &= (s_{k-1} + y_k)(1 + \sigma_k), \\ c_k &= ((s_k - s_{k-1})(1 + \gamma_k) - y_k)(1 + \delta_k), \end{aligned}$$

where $|\eta_k|, |\sigma_k|, |\gamma_k|, |\delta_k| \leq \epsilon$. Prove that $s_n = \sum_{1 \leq k \leq n} (1 + \theta_k) x_k$, where $|\theta_k| \leq 2\epsilon + O(n\epsilon^2)$. [Theorem C says that if $b = 2$ and $|s_{k-1}| \geq |y_k|$ we have $s_{k-1} + y_k = s_k - c_k$ exactly. But in this exercise we want to obtain an estimate that is valid even when floating point operations are not carefully rounded, assuming only that each operation has bounded relative error.]

20. [25] (S. Linnainmaa.) Find all u, v for which $|u| \geq |v|$ and (47) fails.

21. [M35] (T. J. Dekker.) Theorem C shows how to do exact addition of floating binary numbers. Explain how to do exact multiplication: Express the product uv in the form $w + w'$, where w and w' are computed from two given floating binary numbers u and v , using only the operations \oplus , \ominus , and \otimes .

22. [M30] Can drift occur in floating point multiplication/division? Consider the sequence $x_0 = u$, $x_{2n+1} = x_{2n} \otimes v$, $x_{2n+2} = x_{2n+1} \oslash v$, given u and v ; what is the largest subscript k such that $x_k \neq x_{k+2}$ is possible?

► 23. [M26] Prove or disprove: $u \ominus (u \text{ mod } 1) = \lfloor u \rfloor$, for all floating point u .

- 24.** [M27] Consider the set of all intervals $[u_l, u_r]$, where u_l and u_r are either nonzero floating point numbers or the special symbols $+0$, -0 , $+\infty$, $-\infty$; each interval must have $u_l \leq u_r$, and $u_l = u_r$ is allowed only when u_l is finite and nonzero. The interval $[u_l, u_r]$ stands for all floating point x such that $u_l \leq x \leq u_r$, where we regard

$$-\infty < -x < -0 < +0 < +x < +\infty$$

for all positive x . (Thus, $[1, 2]$ means $1 \leq x \leq 2$; $[+0, 1]$ means $0 < x \leq 1$; $[-0, 1]$ means $0 \leq x \leq 1$; $[-0, +0]$ denotes the single value 0; and $[-\infty, +\infty]$ stands for everything.) Show how to define appropriate arithmetic operations on all such intervals, without resorting to “overflow” or “underflow” or other anomalous indications except when dividing by an interval that includes zero.

- 25.** [15] When people speak about inaccuracy in floating point arithmetic they often ascribe errors to “cancellation” that occurs during the subtraction of nearly equal quantities. But when u and v are approximately equal, the difference $u \ominus v$ is obtained exactly, with no error. What do these people really mean?

- 26.** [HM30] (H. G. Diamond.) Suppose $f(x)$ is a strictly increasing function on some interval $[x_0, x_1]$, and let $g(x)$ be the inverse function. (For example, f and g might be “exp” and “ln”, or “tan” and “arctan”.) If x is a floating point number such that $x_0 \leq x \leq x_1$, let $\hat{f}(x) = \text{round}(f(x))$, and if y is a floating point number such that $f(x_0) \leq y \leq f(x_1)$, let $\hat{g}(y) = \text{round}(g(y))$; furthermore, let $h(x) = \hat{g}(\hat{f}(x))$, whenever this is defined. Although $h(x)$ won’t always be equal to x , due to rounding, we expect $h(x)$ to be “near” x .

- Prove that if the precision b^p is at least 3, and if f is strictly concave or strictly convex (i.e., $f''(x) < 0$ or $f''(x) > 0$ for all x in $[x_0, x_1]$), then repeated application of h will be *stable* in the sense that

$$h(h(h(x))) = h(h(x)),$$

for all x such that both sides of this equation are defined. In other words, there will be no “drift” if the subroutines are properly implemented.

- 27.** [M25] (W. M. Kahan.) Let $f(x) = 1 + x + \dots + x^{106} = (1 - x^{107})/(1 - x)$ for $x < 1$, and let $g(y) = f((\frac{1}{3} - y^2)(3 + 3.45y^2))$ for $0 < y < 1$. Evaluate $g(y)$ on one or more “pocket calculators,” for $y = 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$, and explain all inaccuracies in the results you obtain. (Since most present-day calculators do not round correctly, the results are often surprising. Note that $g(\epsilon) = 107 - 10491.35\epsilon^2 + O(\epsilon^4)$.)

*4.2.3. Double-Precision Calculations

Up to now we have considered “single-precision” floating point arithmetic, which essentially means that the floating point values we have dealt with can be stored in a single machine word. When single-precision floating point arithmetic does not yield sufficient accuracy for a given application, the precision can be increased by suitable programming techniques that use two or more words of memory to represent each number.

Although we shall discuss the general question of high-precision calculations in Section 4.3, it is appropriate to give a separate discussion of double-precision

here. Special techniques apply to double precision that are comparatively inappropriate for higher precisions; and double precision is a reasonably important topic in its own right, since it is the first step beyond single precision and it is applicable to many problems that do not require extremely high precision.

Double-precision calculations are almost always required for floating point rather than fixed point arithmetic, except perhaps in statistical work where fixed point double-precision is commonly used to calculate sums of squares and cross products; since fixed point versions of double-precision arithmetic are simpler than floating point versions, we shall confine our discussion here to the latter.

Double precision is quite frequently desired not only to extend the precision of the fraction parts of floating point numbers, but also to increase the range of the exponent part. Thus we shall deal in this section with the following two-word format for double-precision floating point numbers in the MIX computer:



Here two bytes are used for the exponent and eight bytes are used for the fraction. The exponent is “excess $b^2/2$,” where b is the byte size. The sign will appear in the most significant word; it is convenient to ignore the sign of the other word completely.

Our discussion of double-precision arithmetic will be quite machine-oriented, because it is only by studying the problems involved in coding these routines that a person can properly appreciate the subject. A careful study of the MIX programs below is therefore essential to the understanding of the material.

In this section we shall depart from the idealistic goals of accuracy stated in the previous two sections; our double-precision routines will not round their results, and a little bit of error will sometimes be allowed to creep in. Users dare not trust these routines too much. There was ample reason to squeeze out every possible drop of accuracy in the single-precision case, but now we face a different situation: (a) The extra programming required to ensure true double-precision rounding in all cases is considerable; fully accurate routines would take, say, twice as much space and half again as much more time. It was comparatively easy to make our single-precision routines perfect, but double precision brings us face to face with our machine’s limitations. (A similar situation occurs with respect to other floating point subroutines; we can’t expect the cosine routine to compute $\text{round}(\cos x)$ exactly for all x , since that turns out to be virtually impossible. Instead, the cosine routine should provide the best relative error it can achieve with reasonable speed, for all reasonable values of x . Of course, the designer of the routine should try to make the computed function satisfy simple mathematical laws whenever possible (e.g., $(\cos)(-x) = (\cos)x$, $|(\cos)x| \leq 1$, and $(\cos)x \geq (\cos)y$ for $0 \leq x \leq y < \pi$). (b) Single-precision arithmetic is a “staple food” that everybody who wants to employ floating point arithmetic must use, but double precision is usually for situations where such clean results aren’t as important. The difference between seven- and eight-place accuracy can be noticeable, but we rarely care about the difference between 15- and 16-place

accuracy. Double precision is most often used for intermediate steps during the calculation of single-precision results; its full potential isn't needed. (c) It will be instructive for us to analyze these procedures in order to see how inaccurate they can be, since they typify the types of short cuts generally taken in bad single-precision routines (see exercises 7 and 8).

Let us now consider addition and subtraction operations from this standpoint. Subtraction is, of course, converted to addition by changing the sign of the second operand. Addition is performed by separately adding together the least-significant halves and the most-significant halves, propagating "carries" appropriately.

A difficulty arises, however, since we are doing signed-magnitude arithmetic: it is possible to add the least-significant halves and to get the wrong sign (namely, when the signs of the operands are opposite and the least-significant half of the smaller operand is bigger than the least-significant half of the larger operand). The simplest solution is to anticipate the correct sign; so in step A2 (cf. Algorithm 4.2.1A), we not only assume that $e_u \geq e_v$, we also assume that $|u| \geq |v|$. This means we can be sure that the final sign will be the sign of u . In other respects, double-precision addition is very much like its single-precision counterpart, only everything is done twice.

Program A (*Double-precision addition*). The subroutine DFADD adds a double-precision floating point number v , having the form (1), to a double-precision floating point number u , assuming that v is initially in rAX (i.e., registers A and X), and that u is initially stored in locations ACC and ACCX. The answer appears both in rAX and in (ACC, ACCX). The subroutine DFSUB subtracts v from u under the same conventions.

Both input operands are assumed to be normalized, and the answer is normalized. The last portion of this program is a double-precision normalization procedure that is used by other subroutines of this section. Exercise 5 shows how to improve the program significantly.

01	ABS	EQU 1:5	Field definition for absolute value
02	SIGN	EQU 0:0	Field definition for sign
03	EXPD	EQU 1:2	Double-precision exponent field
04	DFSUB	STA TEMP	Double-precision subtraction:
05		LDAN TEMP	Change sign of v .
06	DFADD	STJ EXITDF	Double-precision addition:
07		CMPA ACC(ABS)	Compare $ u $ with $ v $.
08		JG 1F	
09		JL 2F	
10		CMPX ACCX(ABS)	
11		JLE 2F	
12	1H	STA ARG	If $ u < v $, interchange $u \leftrightarrow v$.
13		STX ARGX	
14		LDA ACC	
15		LDX ACCX	
16		ENT1 ACC	(ACC and ACCX are in consecutive
17		MOVE ARG(2)	locations.)

18	2H	STA TEMP	Now ACC has the sign of the answer.
19		LD1N TEMP(EXPD)	$rI1 \leftarrow -e_v$.
20		LD2 ACC(EXPD)	$rI2 \leftarrow e_u$.
21		INC1 0,2	$rI1 \leftarrow e_u - e_v$.
22		SLAX 2	Remove exponent.
23		SRAZ 1,1	Scale right.
24		STA ARG	$0 v_1 v_2 v_3 v_4$
25		STX ARGX	$v_5 v_6 v_7 v_8 v_9$
26		STA ARGX(SIGN)	Store true sign in both halves.
27		LDA ACC	
28		LDX ACCX	
29		SLAX 2	Remove exponent.
30		STA ACC	$u_1 u_2 u_3 u_4 u_5$
31		SLAX 4	
32		ENTX 1	
33		STX EXP0	$EXP0 \leftarrow 1$ (see below).
34		SRC 1	$1 u_5 u_6 u_7 u_8$
35		STA 1F(SIGN)	A trick, see comments in text.
36		ADD ARGX(0:4)	Add $0 v_5 v_6 v_7 v_8$.
37		SRAZ 4	
38	1H	DECA 1	Recover from inserted 1. (Sign varies)
39		ADD ACC(0:4)	Add most significant halves.
40		ADD ARG	(Overflow cannot occur)
41	DNORM	JANZ 1F	Normalization routine:
42		JXNZ 1F	f_w in rAX, $e_w = EXP0 + rI2$.
43	DZERO	STA ACC	If $f_w = 0$, set $e_w \leftarrow 0$.
44		JMP 9F	
45	2H	SLAX 1	Normalize to left.
46		DEC2 1	
47	1H	CMPA =0=(1:1)	Is the leading byte zero?
48		JE 2B	
49		SRAZ 2	(Rounding omitted)
50		STA ACC	
51		LDA EXP0	Compute final exponent.
52		INCA 0,2	
53		JAN EXPUND	Is it negative?
54		STA ACC(EXPD)	
55		CMPA =1(3:3)=	Is it more than two bytes?
56		JL 8F	
57	EXPOVD	HLT 20	
58	EXPUND	HLT 10	
59	8H	LDA ACC	Bring answer into rA.
60	9H	STX ACCX	
61	EXITDF	JMP *	Exit from subroutine.
62	ARG	CON 0	
63	ARGX	CON 0	
64	ACC	CON 0	floating point accumulator
65	ACCX	CON 0	
66	EXP0	CON 0	Part of "raw exponent" ■

When the least-significant halves are added together in this program, an extra digit "1" is inserted at the left of the word that is known to have the correct sign. After the addition, this byte can be 0, 1, or 2, depending on the circumstances, and all three cases are handled simultaneously in this way. (Compare this with the rather cumbersome method of complementation that is used in Program 4.2.1A.)

It is worth noting that register A can be zero after the instruction on line 40 has been performed; and, because of the way MIX defines the sign of a zero result, the accumulator contains the correct sign that is to be attached to the result if register X is nonzero. If lines 39 and 40 were interchanged, the program would be incorrect, even though both instructions are "ADD"!

Now let us consider double-precision multiplication. The product has four components, shown schematically in Fig. 4. Since we need only the leftmost eight bytes, it is convenient to work only with the digits to the left of the vertical line in the diagram, and this means in particular that we need not even compute the product of the two least-significant halves.

Program M (Double-precision multiplication). The input and output conventions for this subroutine are the same as for Program A.

01	BYTE	EQU	1(4:4)	Byte size
02	QQ	EQU	BYTE*BYTE/2	Excess of double-precision exponent
03	DFMUL	STJ	EXITDF	Double-precision multiplication:
04	STA	TEMP		
05	SLAX	2		Remove exponent.
06	STA	ARG		v_m
07	STX	ARGX		v_l
08	LDA	TEMP(EXPD)		
09	ADD	ACC(EXPD)		
10	STA	EXPO		$EXPO \leftarrow e_u + e_v.$
11	ENT2	-QQ		$rI2 \leftarrow -QQ.$
12	LDA	ACC		
13	LDX	ACCX		
14	SLAX	2		Remove exponent.
15	STA	ACC		u_m
16	STX	ACCX		u_l
17	MUL	ARGX		$u_m \times v_l$
18	STA	TEMP		
19	LDA	ARG(ABS)		
20	MUL	ACCX(ABS)		$ v_m \times u_l $
21	SRA	1		0 x x x x
22	ADD	TEMP(1:4)		(Overflow cannot occur)
23	STA	TEMP		
24	LDA	ARG		
25	MUL	ACC		$v_m \times u_m$
26	STA	TEMP(SIGN)		Store true sign of result.
27	STA	ACC		Now prepare to add all the
28	STX	ACCX		partial products together.

$$\begin{array}{r|rr}
 & u & u & u & u & u & 0 & 0 = u_m + \epsilon u_l \\
 & v & v & v & v & v & 0 & 0 = v_m + \epsilon v_l \\
 \hline
 x & x & x & x & x & x & 0 & 0 & 0 = \epsilon^2 u_l \times v_l \\
 x & x & x & x & x & x & 0 & 0 & = \epsilon u_m \times v_l \\
 x & x & x & x & x & x & 0 & 0 & = \epsilon u_l \times v_m \\
 \hline
 w & w & w & w & w & w & w & w & w = u_m \times v_m
 \end{array}$$

Fig. 4. Double-precision multiplication of eight-byte fraction parts.

```

29      LDA ACCX(0:4)    0 x x x x
30      ADD TEMP        (Overflow cannot occur)
31      SRAX 4
32      ADD ACC         (Overflow cannot occur)
33      JMP DNORM       Normalize and exit. ■

```

Note the careful treatment of signs in this program, and note also the fact that the range of exponents makes it impossible to compute the final exponent using an index register. Program M is perhaps too slipshod in accuracy, since it throws away all the information to the right of the vertical line in Fig. 4; this can make the least significant byte as much as 2 in error. A little more accuracy can be achieved as discussed in exercise 4.

Double-precision floating division is the most difficult routine, or at least the most frightening prospect we have encountered so far in this chapter. Actually, it is not terribly complicated, once we see how to do it; let us write the numbers to be divided in the form $(u_m + \epsilon u_l)/(v_m + \epsilon v_l)$, where ϵ is the reciprocal of the word size of the computer, and where v_m is assumed to be normalized. The fraction can now be expanded as follows:

$$\begin{aligned}
 \frac{u_m + \epsilon u_l}{v_m + \epsilon v_l} &= \frac{u_m + \epsilon u_l}{v_m} \left(\frac{1}{1 + \epsilon(v_l/v_m)} \right) \\
 &= \frac{u_m + \epsilon u_l}{v_m} \left(1 - \epsilon \left(\frac{v_l}{v_m} \right) + \epsilon^2 \left(\frac{v_l}{v_m} \right)^2 - \dots \right). \tag{2}
 \end{aligned}$$

Since $0 \leq |v_l| < 1$ and $1/b \leq |v_m| < 1$, we have $|v_l/v_m| < b$, and the error from dropping terms involving ϵ^2 can be disregarded. Our method therefore is to compute $w_m + \epsilon w_l = (u_m + \epsilon u_l)/v_m$, and then to subtract ϵ times $w_m v_l/v_m$ from the result.

In the following program, lines 27–32 do the lower half of a double-precision addition, using another method for forcing the appropriate sign as an alternative to the trick of Program A.

Program D (Double-precision division). This program adheres to the same conventions as Programs A and M.

01	DFDIV	STJ EXITDF	Double-precision division:
02		JOV OFLO	Ensure overflow is off.
03		STA TEMP	
04		SLAX 2	Remove exponent.
05		STA ARG	v_m
06		STX ARGX	v_l
07		LDA ACC(EXPD)	
08		SUB TEMP(EXPD)	
09		STA EXP0	$\text{EXP0} \leftarrow e_u - e_v.$
10		ENT2 QQ+1	$rI2 \leftarrow QQ + 1.$
11		LDA ACC	
12		LDX ACCX	
13		SLAX 2	Remove exponent.
14		SRAZ 1	(Cf. Algorithm 4.2.1M)
15		DIV ARG	If overflow, it is detected below.
16		STA ACC	w_m
17		SLAX 5	Use remainder in further division.
18		DIV ARG	
19		STA ACCX	$\pm w_l$
20		LDA ARGX(1:4)	
21		ENTX 0	
22		DIV ARG(ABS)	$rA \leftarrow \lfloor b^4 v_l / v_m \rfloor / b^5.$
23		JOV DVZROD	Did division cause overflow?
24		MUL ACC(ABS)	$rAX \leftarrow w_m v_l / b v_m $, approximately.
25		SRAZ 4	Multiply by b , and save
26		SLC 5	the leading byte in rX.
27		SUB ACCX(ABS)	Subtract $ w_l $.
28		DECA 1	Force minus sign.
29		SUB WM1	
30		JOV *+2	If no overflow, carry one more
31		INCX 1	to upper half.
32		SLC 5	(Now $rA \leq 0$)
33		ADD ACC(ABS)	$rA \leftarrow w_m - rA $.
34		STA ACC(ABS)	(Now $rA \geq 0$)
35		LDA ACC	w_m with correct sign
36		JMP DNORM	Normalize and exit.
37	DVZROD	HLT 30	Unnormalized or zero divisor
38	1H	EQU 1(1:1)	
39	WM1	CON 1B-1,BYTE-1(1:1)	Word size minus one ■

Here is a table of the approximate average computation times for these double-precision subroutines, compared to the single-precision subroutines that appear in Section 4.2.1:

	Single precision	Double precision
Addition	45.5u	84u
Subtraction	49.5u	88u
Multiplication	48u	109u
Division	52u	126.5u

For extension of the methods of this section to triple-precision floating point fraction parts, see Y. Ikebe, CACM 8 (1965), 175–177.

EXERCISES

1. [16] Try the double-precision division technique by hand, with $\epsilon = \frac{1}{1000}$, when dividing 180000 by 314159. (Thus, let $(u_m, u_l) = (.180, .000)$ and $(v_m, v_l) = (.314, .159)$, and find the quotient using the method suggested in the text following (2).)

2. [20] Would it be a good idea to insert the instruction “ENTX 0” between lines 30 and 31 of Program M, in order to keep unwanted information left over in register X from interfering with the accuracy of the results?

3. [M20] Explain why overflow cannot occur during Program M.

4. [22] How should Program M be changed so that extra accuracy is achieved, essentially by moving the vertical line in Fig. 4 over to the right one position? Specify all changes that are required, and determine the difference in execution time caused by these changes.

► 5. [24] How should Program A be changed so that extra accuracy is achieved, essentially by working with a nine-byte accumulator instead of an eight-byte accumulator to the right of the decimal point? Specify all changes that are required, and determine the difference in execution time caused by these changes.

6. [23] Assume that the double-precision subroutines of this section and the single-precision subroutines of Section 4.2.1 are being used in the same main program. Write a subroutine that converts a single-precision floating point number into double-precision form (1), and write another subroutine that converts a double-precision floating point number into single-precision form (reporting exponent overflow or underflow if the conversion is impossible).

► 7. [M30] Estimate the accuracy of the double-precision subroutines in this section, by finding bounds δ_1 , δ_2 , and δ_3 on the relative errors

$$\begin{aligned} |((u \oplus v) - (u + v))/(u + v)|, \quad & |((u \otimes v) - (u \times v))/(u \times v)|, \\ |((u \oslash v) - (u/v))/(u/v)|. \end{aligned}$$

8. [M28] Estimate the accuracy of the “improved” double-precision subroutines of exercises 4 and 5, in the sense of exercise 7.

9. [M42] T. J. Dekker [Numer. Math. 18 (1971), 224–242] has suggested an alternative approach to double precision, based entirely on single-precision floating binary calculations. For example, Theorem 4.2.2C states that $u+v=w+r$, where $w=u\oplus v$ and $r=(u\ominus w)\oplus v$, if $|u|\geq|v|$ and the radix is 2; here $|r|\leq|w|/2^p$, so the pair (w,r) may be considered a double-precision version of $u+v$. To add two such pairs $(u,u')\oplus(v,v')$, where $|u'|\leq|u|/2^p$ and $|v'|\leq|v|/2^p$ and $|u|\geq|v|$, Dekker suggests computing $u+v=w+r$ (exactly), then $s=(r\oplus v')\oplus u'$ (an approximate remainder), and finally returning the value $(w\oplus s,(w\ominus(w\oplus s))\oplus s)$.

Study the accuracy and efficiency of this approach when it is used recursively to produce quadruple-precision calculations.

Table 1
EMPIRICAL DATA FOR OPERAND ALIGNMENTS BEFORE ADDITION

$ e_u - e_v $	$b = 2$	$b = 10$	$b = 16$	$b = 64$
0	0.33	0.47	0.47	0.56
1	0.12	0.23	0.26	0.27
2	0.09	0.11	0.10	0.04
3	0.07	0.03	0.02	0.02
4	0.07	0.01	0.01	0.02
5	0.04	0.01	0.02	0.00
over 5	0.28	0.13	0.11	0.09
average	3.1	0.9	0.8	0.5

4.2.4. Distribution of Floating Point Numbers

In order to analyze the average behavior of floating point arithmetic algorithms (and in particular to determine their average running time), we need some statistical information that allows us to determine how often various cases arise. The purpose of this section is to discuss the empirical and theoretical properties of the distribution of floating point numbers.

A. Addition and subtraction routines. The execution time for a floating point addition or subtraction depends largely on the initial difference of exponents, and also on the number of normalization steps required (to the left or to the right). No way is known to give a good theoretical model that tells what characteristics to expect, but extensive empirical investigations have been made by D. W. Sweeney [*IBM Systems J.* 4 (1965), 31–42].

By means of a special tracing routine, Sweeney ran six “typical” large-scale numerical programs, selected from several different computing laboratories, and examined each floating addition or subtraction operation very carefully. Over 250,000 floating point addition-subtractions were involved in gathering this data. About one out of every ten instructions executed by the tested programs was either FADD or FSUB.

Let us consider subtraction to be addition preceded by negating the second operand; therefore we may give all the statistics as if we were merely doing addition. Sweeney’s results can be summarized as follows:

One of the two operands to be added was found to be equal to zero about 9 percent of the time, and this was usually the accumulator (ACC). The other 91 percent of the cases split about equally between operands of the same or of opposite signs, and about equally between cases where $|u| \leq |v|$ or $|v| \leq |u|$. The computed answer was zero about 1.4 percent of the time.

The difference between exponents had a behavior approximately given by the probabilities shown in Table 1, for various radices b . (The “over 5” line of that table includes essentially all of the cases when one operand was zero, but the “average” line does not include these cases.)

Table 2
EMPIRICAL DATA FOR NORMALIZATION AFTER ADDITION

	$b = 2$	$b = 10$	$b = 16$	$b = 64$
Shift right 1	0.20	0.07	0.06	0.03
No shift	0.59	0.80	0.82	0.87
Shift left 1	0.07	0.08	0.07	0.06
Shift left 2	0.03	0.02	0.01	0.01
Shift left 3	0.02	0.00	0.01	0.00
Shift left 4	0.02	0.01	0.00	0.01
Shift left >4	0.06	0.02	0.02	0.02

When u and v have the same sign and are normalized, then $u + v$ either requires one shift to the *right* (for fraction overflow), or no normalization shifts whatever. When u and v have opposite signs, we have zero or more *left* shifts during the normalization. Table 2 gives the observed number of shifts required; the last line of that table includes all cases where the result was zero. The average number of left shifts per normalization was about 0.9 when $b = 2$; about 0.2 when $b = 10$ or 16; and about 0.1 when $b = 64$.

B. The fraction parts. Further analysis of floating point routines can be based on the *statistical distribution of the fraction parts* of randomly chosen normalized floating point numbers. In this case the facts are quite surprising, and there is an interesting theory that accounts for the unusual phenomena that are observed.

For convenience let us temporarily assume that we are dealing with floating decimal (i.e., radix 10) arithmetic; modifications of the following discussion to any other positive integer base b will be very straightforward. Suppose we are given a “random” positive normalized number $(e, f) = 10^e \cdot f$. Since f is normalized, we know that its leading digit is 1, 2, 3, 4, 5, 6, 7, 8, or 9, and it seems natural to assume that each of these nine possible leading digits will occur about one-ninth of the time. But, in fact, the behavior in practice is quite different. For example, the leading digit tends to be equal to 1 over 30 percent of the time!

One way to test the assertion just made is to take a table of physical constants (e.g., the speed of light, the acceleration of gravity) from some standard reference. If we look at the *Handbook of Mathematical Functions* (U.S. Dept of Commerce, 1964), for example, we find that 8 of the 28 different physical constants given in Table 2.3, roughly 29 percent, have leading digit equal to 1. The decimal values of $n!$ for $1 \leq n \leq 100$ include exactly 30 entries beginning with 1; so do the decimal values of 2^n and of F_n , for $1 \leq n \leq 100$. We might also try looking at census reports, or a Farmer’s Almanack (but not a telephone directory).

In the days before pocket calculators, the pages in well-used tables of logarithms tended to get quite dirty in the front, while the last pages stayed relatively clean and neat. This phenomenon was apparently first mentioned in print by the American astronomer Simon Newcomb [*Amer. J. Math.* 4 (1881), 39–40], who

gave good grounds for believing that the leading digit d occurs with probability $\log_{10}(1 + 1/d)$. The same distribution was discovered empirically, many years later, by Frank Benford, who reported the results of 20,229 observations taken from different sources [Proc. Amer. Philosophical Soc. 78 (1938), 551–572].

In order to account for this leading-digit law, let's take a closer look at the way we write numbers in floating point notation. If we take any positive number u , its leading digits are determined by the value $(\log_{10} u) \bmod 1$: The leading digit is less than d if and only if

$$(\log_{10} u) \bmod 1 < \log_{10} d, \quad (1)$$

since $10f_u = 10^{(\log_{10} u) \bmod 1}$.

Now if we have a “random” positive number U , chosen from some reasonable distribution that might occur in nature, we might expect that $(\log_{10} U) \bmod 1$ would be uniformly distributed between zero and one, at least to a very good approximation. (Similarly, we expect $U \bmod 1$, $U^2 \bmod 1$, $\sqrt{U + \pi} \bmod 1$, etc., to be uniformly distributed. We expect a roulette wheel to be unbiased, for essentially the same reason.) Therefore by (1) the leading digit will be 1 with probability $\log_{10} 2 \approx 30.103$ percent; it will be 2 with probability $\log_{10} 3 - \log_{10} 2 \approx 17.609$ percent; and, in general, if r is any real value between 1 and 10, we ought to have $10f_U \leq r$ approximately $\log_{10} r$ of the time.

Another way to explain this law is to say that a random value U should appear at a random point on a slide rule, according to the uniform distribution, since the distance from the left end of a slide rule to the position of U is proportional to $(\log_{10} U) \bmod 1$. The analogy between slide rules and floating point calculation is very close when multiplication and division are being considered.

The fact that leading digits tend to be small is important to keep in mind; it makes the most obvious techniques of “average error” estimation for floating point calculations invalid. The relative error due to rounding is usually a little more than expected.

Of course, it may justly be said that the heuristic argument above does not prove the stated law. It merely shows us a plausible reason why the leading digits behave the way they do. An interesting approach to the analysis of leading digits has been suggested by R. Hamming: Let $p(r)$ be the probability that $10f_U \leq r$, where $1 \leq r \leq 10$ and f_U is the normalized fraction part of a random normalized floating point number U . If we think of random quantities in the real world, we observe that they are measured in terms of arbitrary units; and if we were to change the definition of a meter or a gram, many of the fundamental physical constants would have different values. Suppose then that all of the numbers in the universe are suddenly multiplied by a constant factor c ; our universe of random floating point quantities should be essentially unchanged by this transformation, so $p(r)$ should not be affected.

Multiplying everything by c has the effect of transforming $(\log_{10} U) \bmod 1$ into $(\log_{10} U + \log_{10} c) \bmod 1$. It is now time to set up formulas that describe the desired behavior; we may assume that $1 \leq c \leq 10$. By definition,

$$p(r) = \text{probability that } (\log_{10} U) \bmod 1 \leq \log_{10} r.$$

By our assumption, we should also have

$$\begin{aligned}
 p(r) &= \text{probability that } (\log_{10} U + \log_{10} c) \bmod 1 \leq \log_{10} r \\
 &= \begin{cases} \text{probability that } (\log_{10} U \bmod 1) \leq \log_{10} r - \log_{10} c \\ \quad \text{or } (\log_{10} U \bmod 1) \geq 1 - \log_{10} c, & \text{if } c \leq r; \\ \text{probability that } (\log_{10} U \bmod 1) \leq \log_{10} r + 1 - \log_{10} c \\ \quad \text{and } (\log_{10} U \bmod 1) \geq 1 - \log_{10} c, & \text{if } c \geq r; \end{cases} \\
 &= \begin{cases} p(r/c) + 1 - p(10/c), & \text{if } c \leq r; \\ p(10r/c) - p(10/c), & \text{if } c \geq r. \end{cases} \tag{2}
 \end{aligned}$$

Let us now extend the function $p(r)$ to values outside the range $1 \leq r \leq 10$, by defining $p(10^n r) = p(r) + n$; then if we replace $10/c$ by d , the last equation of (2) may be written

$$p(rd) = p(r) + p(d). \tag{3}$$

If our assumption about invariance of the distribution under multiplication by a constant factor is valid, then Eq. (3) must hold for all $r > 0$ and $1 \leq d \leq 10$. The facts that $p(1) = 0$, $p(10) = 1$ now imply that

$$1 = p(10) = p((\sqrt[10]{10})^n) = p(\sqrt[10]{10}) + p((\sqrt[10]{10})^{n-1}) = \dots = np(\sqrt[10]{10});$$

hence we deduce that $p(10^{m/n}) = m/n$ for all positive integers m and n . If we now decide to require that p is continuous, we are forced to conclude that $p(r) = \log_{10} r$, and this is the desired law.

Although this argument may be more convincing than the first one, it doesn't really hold up under scrutiny if we stick to conventional notions of probability. The traditional way to make the above argument rigorous is to assume that there is some underlying distribution of numbers $F(u)$ such that a given positive number U is $\leq u$ with probability $F(u)$; then the probability of concern to us is

$$p(r) = \sum_m (F(10^m r) - F(10^m)), \tag{4}$$

summed over all values $-\infty < m < \infty$. Our assumptions about scale invariance and continuity have led us to conclude that

$$p(r) = \log_{10} r.$$

Using the same argument, we could "prove" that

$$\sum_m (F(b^m r) - F(b^m)) = \log_b r, \tag{5}$$

for each integer $b \geq 2$, when $1 \leq r \leq b$. But there is no distribution function F that satisfies this equation for all such b and r ! (See exercise 7.)

One way out of the difficulty is to regard the logarithm law $p(r) = \log_{10} r$ as only a very close *approximation* to the true distribution. The true distribution itself may perhaps be changing as the universe expands, becoming a better and better approximation as time goes on; and if we replace 10 by an arbitrary base b , the approximation might be less accurate (at any given time) as b gets larger. Another rather appealing way to resolve the dilemma, by abandoning the traditional idea of a distribution function, has been suggested by R. A. Raimi, *AMM* 76 (1969), 342–348.

The hedging in the last paragraph is probably a very unsatisfactory explanation, and so the following further calculation (which sticks to rigorous mathematics and avoids any intuitive, yet paradoxical, notions of probability) should be welcome. Let us consider the distribution of the leading digits of the *positive integers*, instead of the distribution for some imagined set of real numbers. The investigation of this topic is quite interesting, not only because it sheds some light on the probability distributions of floating point data, but also because it makes a particularly instructive example of how to combine the methods of discrete mathematics with the methods of infinitesimal calculus.

In the following discussion, let r be a fixed real number, $1 \leq r \leq 10$; we will attempt to make a reasonable definition of $p(r)$, the “probability” that the representation $10^{e_N} \cdot f_N$ of a “random” positive integer N has $10f_N < r$, assuming infinite precision.

To start, let us try to find the probability using a limiting method like the definition of “Pr” in Section 3.5. One nice way to rephrase that definition is to define

$$P_0(n) = \begin{cases} 1, & \text{if } n = 10^e \cdot f \text{ where } 10f < r, \\ & \quad \text{i.e., if } (\log_{10} n) \bmod 1 < \log_{10} r; \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

Now $P_0(1), P_0(2), \dots$ is an infinite sequence of zeros and ones, with ones to represent the cases that contribute to the probability we are seeking. We can try to “average out” this sequence, by defining

$$P_1(n) = \frac{1}{n} \sum_{1 \leq k \leq n} P_0(k). \quad (7)$$

Thus if we generate a random integer between 1 and n using the techniques of Chapter 3, and convert it to floating decimal form (e, f) , the probability that $10f < r$ is exactly $P_1(n)$. It is natural to let $\lim_{n \rightarrow \infty} P_1(n)$ be the “probability” $p(r)$ we are after, and that is just what we did in Section 3.5.

But in this case the limit does not exist: For example, let us consider the subsequence

$$P_1(s), P_1(10s), P_1(100s), \dots, P_1(10^n s), \dots,$$

where s is a real number, $1 \leq s \leq 10$. If $s \leq r$, we find that

$$\begin{aligned}
P_1(10^n s) &= \frac{1}{10^n s} ([r] - 1 + [10r] - 10 + \cdots + [10^{n-1}r] \\
&\quad - 10^{n-1} + [10^n s] + 1 - 10^n) \\
&= \frac{1}{10^n s} (r(1 + 10 + \cdots + 10^{n-1}) + O(n) \\
&\quad + [10^n s] - 1 - 10 - \cdots - 10^n) \\
&= \frac{1}{10^n s} (\frac{1}{9}(10^n r - 10^{n+1}) + [10^n s] + O(n)). \tag{8}
\end{aligned}$$

As $n \rightarrow \infty$, $P_1(10^n s)$ therefore approaches the limiting value $1 + (r - 10)/9s$. The above calculation for the case $s \leq r$ can be modified so that it is valid for $s > r$ if we replace $[10^n s] + 1$ by $[10^n r]$; when $s \geq r$, we therefore obtain the limiting value $10(r - 1)/9s$. [See J. Franel, *Naturforschende Gesellschaft, Vierteljahrsschrift* **62** (Zürich, 1917), 286–295.]

In other words, the sequence $\langle P_1(n) \rangle$ has subsequences $\langle P_1(10^n s) \rangle$ whose limit goes from $(r - 1)/9$ up to $10(r - 1)/9r$ and down again to $(r - 1)/9$, as s goes from 1 to r to 10. We see that $P_1(n)$ has no limit as $n \rightarrow \infty$; and the values of $P_1(n)$ for large n are not particularly good approximations to our conjectured limit $\log_{10} r$ either!

Since $P_1(n)$ doesn't approach a limit, we can try to use the same idea as (7) once again, to "average out" the anomalous behavior. In general, let

$$P_{m+1}(n) = \frac{1}{n} \sum_{1 \leq k \leq n} P_m(k). \tag{9}$$

Then $P_{m+1}(n)$ will tend to be a more well-behaved sequence than $P_m(n)$. Let us try to confirm this with quantitative calculations; our experience with the special case $m = 0$ indicates that it might be worthwhile to consider the subsequence $P_{m+1}(10^n s)$. The following results can, in fact, be derived:

Lemma Q. For any integer $m \geq 1$ and any real number $\epsilon > 0$, there are functions $Q_m(s)$, $R_m(s)$ and an integer $N_m(\epsilon)$, such that whenever $n > N_m(\epsilon)$ and $1 \leq s \leq 10$, we have

$$\begin{aligned}
|P_m(10^n s) - Q_m(s)| &< \epsilon, & \text{if } s \leq r; \\
|P_m(10^n s) - (Q_m(s) + R_m(s))| &< \epsilon, & \text{if } s > r. \tag{10}
\end{aligned}$$

Furthermore the functions $Q_m(s)$ and $R_m(s)$ satisfy the relations

$$\begin{aligned}
Q_m(s) &= \frac{1}{s} \left(\frac{1}{9} \int_1^{10} Q_{m-1}(t) dt + \int_1^s Q_{m-1}(t) dt + \frac{1}{9} \int_r^{10} R_{m-1}(t) dt \right); \\
R_m(s) &= \frac{1}{s} \int_r^s R_{m-1}(t) dt; \\
Q_0(s) &= 1, \quad R_0(s) = -1. \tag{11}
\end{aligned}$$

Proof. Consider the functions $Q_m(s)$ and $R_m(s)$ defined by (11), and let

$$S_m(t) = \begin{cases} Q_m(t), & t \leq r, \\ Q_m(t) + R_m(t), & t > r. \end{cases} \quad (12)$$

We will prove the lemma by induction on m .

First note that $Q_1(s) = (1 + (s - 1) + (r - 10)/9)/s = 1 + (r - 10)/9s$, and $R_1(s) = (r - s)/s$. From (8) we find that $|P_1(10^n s) - S_1(s)| = O(n)/10^n$; this establishes the lemma when $m = 1$.

Now for $m > 1$, we have

$$P_m(10^n s) = \frac{1}{s} \left(\sum_{0 \leq j < n} \frac{1}{10^{n-j}} \sum_{10^j \leq k < 10^{j+1}} \frac{1}{10^j} P_{m-1}(k) + \sum_{10^n \leq k \leq 10^n s} \frac{1}{10^n} P_{m-1}(k) \right),$$

and we want to approximate this quantity. By induction, the difference

$$\left| \sum_{10^j \leq k \leq 10^j q} \frac{1}{10^j} P_{m-1}(k) - \sum_{10^j \leq k \leq 10^j q} \frac{1}{10^j} S_{m-1}\left(\frac{k}{10^j}\right) \right| \quad (13)$$

is less than $q\epsilon$ when $1 \leq q \leq 10$ and $j > N_{m-1}(\epsilon)$. Since $S_{m-1}(t)$ is continuous, it is a Riemann-integrable function; and the difference

$$\left| \sum_{10^j \leq k \leq 10^j q} \frac{1}{10^j} S_{m-1}\left(\frac{k}{10^j}\right) - \int_1^q S_{m-1}(t) dt \right| \quad (14)$$

is less than ϵ for all j greater than some number N , independent of q , by the definition of integration. We may choose N to be $> N_{m-1}(\epsilon)$. Therefore for $n > N$, the difference

$$\left| P_m(10^n s) - \frac{1}{s} \left(\sum_{0 \leq j < n} \frac{1}{10^{n-j}} \int_1^{10} S_{m-1}(t) dt + \int_1^s S_{m-1}(t) dt \right) \right| \quad (15)$$

is bounded by $\sum_{0 \leq j \leq N} (M/10^{n-j}) + \sum_{N < j < n} (11\epsilon/10^{n-j}) + 11\epsilon$, if M is an upper bound for (13) + (14) that is valid for all positive integers j . Finally, the sum $\sum_{0 \leq j < n} (1/10^{n-j})$, which appears in (15), is equal to $(1 - 1/10^n)/9$; so

$$\left| P_m(10^n s) - \frac{1}{s} \left(\frac{1}{9} \int_1^{10} S_{m-1}(t) dt + \int_1^s S_{m-1}(t) dt \right) \right|$$

can be made smaller than, say, 20ϵ , if n is taken large enough. Comparing this with (10) and (11) completes the proof. ■

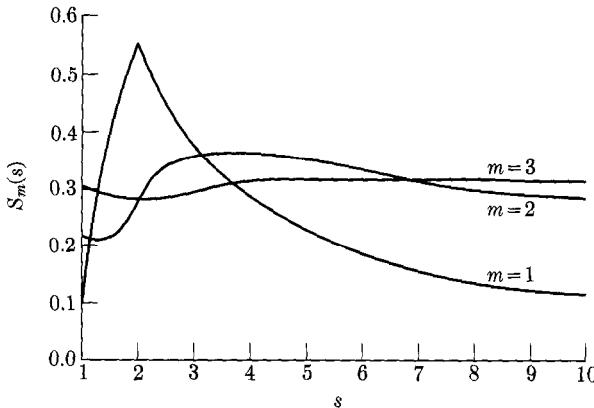


Fig. 5. The probability that the leading digit is 1.

The gist of Lemma Q is that we have the limiting relationship

$$\lim_{n \rightarrow \infty} P_m(10^n s) = S_m(s). \quad (16)$$

Also, since $S_m(s)$ is not constant as s varies, the limit

$$\lim_{n \rightarrow \infty} P_m(n)$$

(which would be our desired “probability”) does not exist for any m . The situation is shown in Fig. 5, which shows the values of $S_m(s)$ when m is small and $r = 2$.

Even though $S_m(s)$ is not a constant, so that we do not have a definite limit for $P_m(n)$, note that already for $m = 3$ in Fig. 5 the value of $S_m(s)$ stays very close to $\log_{10} 2 = 0.30103\dots$. Therefore we have good reason to suspect that $S_m(s)$ is very close to $\log_{10} r$ for all large m , and, in fact, that the sequence of functions $\{S_m(s)\}$ converges uniformly to the constant function $\log_{10} r$.

It is interesting to prove this conjecture by explicitly calculating $Q_m(s)$ and $R_m(s)$ for all m , as in the proof of the following theorem:

Theorem F. *Let $S_m(s)$ be the limit defined in (16). For all $\epsilon > 0$, there exists a number $N(\epsilon)$ such that*

$$|S_m(s) - \log_{10} r| < \epsilon, \quad \text{for } 1 \leq s \leq 10, \quad (17)$$

whenever $m > N(\epsilon)$.

Proof. In view of Lemma Q, we can prove this result if we can show that there is a number M depending on ϵ such that, for $1 \leq s \leq 10$ and for all $m > M$, we have

$$|Q_m(s) - \log_{10} r| < \epsilon \quad \text{and} \quad |R_m(s)| < \epsilon. \quad (18)$$

It is not difficult to solve the recurrence formula (11) for R_m : We have $R_0(s) = -1$, $R_1(s) = -1 + r/s$, $R_2(s) = -1 + (r/s)(1 + \ln(s/r))$, and in general

$$R_m(s) = -1 + \frac{r}{s} \left(1 + \frac{1}{1!} \ln \left(\frac{s}{r} \right) + \cdots + \frac{1}{(m-1)!} \left(\ln \left(\frac{s}{r} \right) \right)^{m-1} \right). \quad (19)$$

For the stated range of s , this converges uniformly to

$$-1 + (r/s) \exp(\ln(s/r)) = 0.$$

The recurrence (11) for Q_m takes the form

$$Q_m(s) = \frac{1}{s} \left(c_m + 1 + \int_1^s Q_{m-1}(t) dt \right), \quad (20)$$

where

$$c_m = \frac{1}{9} \left(\int_1^{10} Q_{m-1}(t) dt + \int_r^{10} R_{m-1}(t) dt \right) - 1. \quad (21)$$

The solution to recurrence (20) is easily found by trying out the first few cases and guessing at a formula that can be proved by induction; we find that

$$Q_m(s) = 1 + \frac{1}{s} \left(c_m + \frac{1}{1!} c_{m-1} \ln s + \cdots + \frac{1}{(m-1)!} c_1 (\ln s)^{m-1} \right). \quad (22)$$

It remains for us to calculate the coefficients c_m , which by (19), (21), and (22) satisfy the relations

$$\begin{aligned} c_1 &= (r - 10)/9; \\ c_{m+1} &= \frac{1}{9} \left(c_m \ln 10 + \frac{1}{2!} c_{m-1} (\ln 10)^2 + \cdots + \frac{1}{m!} c_1 (\ln 10)^m \right. \\ &\quad \left. + r \left(1 + \frac{1}{1!} \ln \frac{10}{r} + \cdots + \frac{1}{m!} \left(\ln \frac{10}{r} \right)^m \right) - 10 \right). \end{aligned} \quad (23)$$

This sequence appears at first to be very complicated, but actually we can analyze it without difficulty with the help of generating functions. Let

$$C(z) = c_1 z + c_2 z^2 + c_3 z^3 + \cdots;$$

then since $10^z = 1 + z \ln 10 + (1/2!)(z \ln 10)^2 + \cdots$, we deduce that

$$\begin{aligned} c_{m+1} &= \frac{1}{10} c_{m+1} + \frac{9}{10} c_{m+1} \\ &= \frac{1}{10} \left(c_{m+1} + c_m \ln 10 + \cdots + \frac{1}{m!} c_1 (\ln 10)^m \right) \\ &\quad + \frac{r}{10} \left(1 + \cdots + \frac{1}{m!} \left(\ln \frac{10}{r} \right)^m \right) - 1 \end{aligned}$$

is the coefficient of z^{m+1} in the function

$$\frac{1}{10}C(z)10^z + \frac{r}{10}\left(\frac{10}{r}\right)^z\left(\frac{z}{1-z}\right) - \frac{z}{1-z}. \quad (24)$$

This condition holds for all values of m , so (24) must equal $C(z)$, and we obtain the explicit formula

$$C(z) = \frac{-z}{1-z} \left(\frac{(10/r)^{z-1} - 1}{10^{z-1} - 1} \right). \quad (25)$$

We want to study asymptotic properties of the coefficients of $C(z)$, to complete our analysis. The large parenthesized factor in (25) approaches $\ln(10/r)/\ln 10 = 1 - \log_{10} r$ as $z \rightarrow 1$, so we see that

$$C(z) + \frac{1 - \log_{10} r}{1-z} = R(z) \quad (26)$$

is an analytic function of the complex variable z in the circle

$$|z| < \left| 1 + \frac{2\pi i}{\ln 10} \right|.$$

In particular, $R(z)$ converges for $z = 1$, so its coefficients approach zero. This proves that the coefficients of $C(z)$ behave like those of $(\log_{10} r - 1)/(1-z)$, that is,

$$\lim_{m \rightarrow \infty} c_m = \log_{10} r - 1.$$

Finally, we may combine this with (22), to show that $Q_m(s)$ approaches

$$1 + \frac{\log_{10} r - 1}{s} \left(1 + \ln s + \frac{1}{2!} (\ln s)^2 + \dots \right) = \log_{10} r$$

uniformly for $1 \leq s \leq 10$. ■

Therefore we have established the logarithmic law for integers by direct calculation, at the same time seeing that it is an extremely good approximation to the average behavior although it is never precisely achieved.

The above proofs of Lemma Q and Theorem F are slight simplifications and amplifications of methods due to B. J. Flehinger, *AMM* 73 (1966), 1056–1061. Many authors have written about the distribution of initial digits, showing that the logarithmic law is a good approximation for many underlying distributions; see the survey by Ralph A. Raimi, *AMM* 83 (1976), 521–538, for a comprehensive review of the literature. Another interesting (and different) treatment of floating point distribution has been given by Alan G. Konheim, *Math. Comp.* 19 (1965), 143–144.

Exercise 17 discusses an approach to the definition of probability under which the logarithmic law holds exactly, over the integers. Furthermore, exercise 18 demonstrates that any reasonable definition of probability over the integers must lead to the logarithmic law, if it assigns a value to the probability of leading digits.

EXERCISES

1. [13] Given that u and v are nonzero floating decimal numbers with the same sign, what is the approximate probability that fraction overflow occurs during the calculation of $u \oplus v$, according to Tables 1 and 2?
2. [42] Make further tests of floating point addition and subtraction, to confirm or improve on the accuracy of Tables 1 and 2.
3. [15] What is the probability that the two leading digits of a floating decimal number are "23", according to the logarithmic law?
4. [M18] The text points out that the front pages of a well-used table of logarithms get dirtier than the back pages do. What if we had an antilogarithm table instead, i.e., a table giving the value of x when $\log_{10} x$ is given; which pages of such a table would be the dirtiest?
- 5. [M20] Let U be a random real number that is uniformly distributed in the interval $0 < U < 1$. What is the distribution of the leading digits of U ?
6. [23] If we have binary computer words containing $n + 1$ bits, we might use p bits for the fraction part of floating binary numbers, one bit for the sign, and $n - p$ bits for the exponent. This means that the range of values representable, i.e., the ratio of the largest positive normalized value to the smallest, is essentially 2^{2n-p} . The same computer word could be used to represent floating hexadecimal numbers, i.e., floating point numbers with radix 16, with $p + 2$ bits for the fraction part (($p + 2)/4$ hexadecimal digits) and $n - p - 2$ bits for the exponent; then the range of values would be $16^{2n-p-2} = 2^{2n-p}$, the same as before, and with more bits in the fraction part. This may sound as if we are getting something for nothing, but the normalization condition for base 16 is weaker in that there may be up to three leading zero bits in the fraction part; thus not all of the $p + 2$ bits are "significant."
- On the basis of the logarithmic law, what are the probabilities that the fraction part of a positive normalized radix 16 floating point number has exactly 0, 1, 2, and 3 leading zero bits? Discuss the desirability of hexadecimal versus binary.
7. [HM28] Prove that there is no distribution function $F(u)$ that satisfies (5) for each integer $b \geq 2$, and for all real values r in the range $1 \leq r \leq b$.
8. [HM23] Does (10) hold when $m = 0$ for suitable $N_0(\epsilon)$?
9. [HM24] (P. Diaconis.) Let $P_1(n), P_2(n), \dots$ be any sequence of functions defined by repeatedly averaging a given function $P_0(n)$ according to Eq. (9). Prove that $\lim_{m \rightarrow \infty} P_m(n) = P_0(1)$ for all fixed n .
- 10. [HM28] The text shows that $c_m = \log_{10} r - 1 + \epsilon_m$, where ϵ_m approaches zero as $m \rightarrow \infty$. Obtain the next term in the asymptotic expansion of c_m .
11. [M15] Given that U is a random variable distributed according to the logarithmic law, prove that $1/U$ is also.
12. [HM25] (R. W. Hamming.) The purpose of this exercise is to show that the result of floating point multiplication tends to obey the logarithmic law more perfectly than the operands do. Let U and V be random, normalized, positive floating point numbers, whose fraction parts are independently distributed with the respective density functions $f(x)$ and $g(x)$. Thus, $f_u \leq r$ and $f_v \leq s$ with probability $\int_{1/b}^r \int_{1/b}^s f(x)g(y) dy dx$, for $1/b \leq r, s \leq 1$. Let $h(x)$ be the density function of the fraction part of $U \times V$ (unrounded). Define the abnormality $A(f)$ of a density function f to be the maximum

relative error,

$$A(f) = \max_{1/b \leq x \leq 1} \left| \frac{f(x) - l(x)}{l(x)} \right|,$$

where $l(x) = 1/(x \ln b)$ is the density of the logarithmic distribution.

Prove that $A(h) \leq \min(A(f), A(g))$. (In particular, if either factor has logarithmic distribution the product does also.)

► 13. [M20] The floating point multiplication routine, Algorithm 4.2.1M, requires zero or one left shifts during normalization, depending on whether $f_u f_v \geq 1/b$ or not. Assuming that the input operands are independently distributed according to the logarithmic law, what is the probability that no left shift is needed for normalization of the result?

► 14. [HM30] Let U and V be random, normalized, positive floating point numbers whose fraction parts are independently distributed according to the logarithmic law, and let p_k be the probability that the difference in their exponents is k . Assuming that the distribution of the exponents is independent of the fraction parts, give an equation for the probability that “fraction overflow” occurs during the floating point addition of $U \oplus V$, in terms of the base b and the quantities p_0, p_1, p_2, \dots . Compare this result with exercise 1. (Ignore rounding.)

15. [HM28] Let U, V, p_0, p_1, \dots be as in exercise 14, and assume that radix 10 arithmetic is being used. Show that regardless of the values of p_0, p_1, p_2, \dots , the sum $U \oplus V$ will not obey the logarithmic law exactly, and in fact the probability that $U \oplus V$ has leading digit 1 is always strictly less than $\log_{10} 2$.

16. [HM28] (P. Diaconis.) Let $P_0(n)$ be 0 or 1 for each n , and define “probabilities” $P_{m+1}(n)$ by repeated averaging, as in (9). Show that if $\lim_{n \rightarrow \infty} P_1(n)$ does not exist, neither does $\lim_{n \rightarrow \infty} P_m(n)$ for any m . [Hint: Prove that $a_n \rightarrow 0$ whenever we have $(a_1 + \dots + a_n)/n \rightarrow 0$ and $a_{n+1} \leq a_n + M/n$, for some fixed constant $M > 0$.]

► 17. [HM25] (R. L. Duncan.) Another way to define the value of $\Pr(S(n))$ is to evaluate the quantity $\lim_{n \rightarrow \infty} ((\sum_{S(k) \text{ and } 1 \leq k \leq n} 1/k)/H_n)$; it can be shown that this “harmonic probability” exists and is equal to $\Pr(S(n))$, whenever the latter exists according to Definition 3.5A. Prove that the harmonic probability of the statement “ $(\log_{10} n) \bmod 1 < r$ ” exists and equals r . (Thus, initial digits of integers exactly satisfy the logarithmic law in this sense.)

► 18. [HM30] Let $P(S)$ be any real-valued function defined on sets S of positive integers, but not necessarily on all such sets, satisfying the following rather weak axioms:

- i) If $P(S)$ and $P(T)$ are defined and $S \cap T = \emptyset$, then $P(S \cup T) = P(S) + P(T)$.
- ii) If $P(S)$ is defined, then $P(S+1) = P(S)$, where $S+1 = \{n+1 \mid n \in S\}$.
- iii) If $P(S)$ is defined, then $P(2S) = \frac{1}{2}P(S)$, where $2S = \{2n \mid n \in S\}$.
- iv) If S is the set of all positive integers, then $P(S) = 1$.
- v) If $P(S)$ is defined, then $P(S) \geq 0$.

Assume furthermore that $P(L_a)$ is defined for all positive integers a , where L_a is the set of all integers whose decimal representation begins with a :

$$L_a = \{n \mid 10^m a \leq n < 10^{m+1} a \text{ for some integer } m\}.$$

(In this definition, m may be negative; for example, 1 is an element of L_{10} , but not of L_{11} .) Prove that $P(L_a) = \log_{10}(1 + 1/a)$ for all integers $a \geq 1$.

4.3. MULTIPLE-PRECISION ARITHMETIC

LET US NOW consider operations on numbers that have arbitrarily high precision. For simplicity in exposition, we shall assume that we are working with integers, instead of with numbers that have an embedded radix point.

4.3.1. The Classical Algorithms

In this section we shall discuss algorithms for

- a) addition or subtraction of n -place integers, giving an n -place answer and a carry;
- b) multiplication of an n -place integer by an m -place integer, giving an $(m + n)$ -place answer;
- c) division of an $(m + n)$ -place integer by an n -place integer, giving an $(m + 1)$ -place quotient and an n -place remainder.

These may be called "the classical algorithms," since the word "algorithm" was used only in connection with these processes for several centuries. The term " n -place integer" means any integer less than b^n , where b is the radix of ordinary positional notation in which the numbers are expressed; such numbers can be written using at most n "places" in this notation.

It is a straightforward matter to apply the classical algorithms for integers to numbers with embedded radix points or to extended-precision floating point numbers, in the same way that arithmetic operations defined for integers in MIX are applied to these more general problems.

In this section we shall study algorithms that do operations (a), (b), and (c) above for integers expressed in radix b notation, where b is any given integer ≥ 2 . Thus the algorithms are quite general definitions of arithmetic processes, and as such they are unrelated to any particular computer. But the discussion in this section will also be somewhat machine-oriented, since we are chiefly concerned with efficient methods for doing high-precision calculations by computer. Although our examples are based on the mythical MIX, essentially the same considerations apply to nearly every other machine. For convenience, we shall assume first that we have a computer (like MIX) that uses the signed-magnitude representation for numbers; suitable modifications for complement notations are discussed near the end of this section.

The most important fact to understand about extended-precision numbers is that they may be regarded as numbers written in radix w notation, where w is the computer's word size. For example, an integer that fills 10 words on a computer whose word size is $w = 10^{10}$ has 100 decimal digits; but we will consider it to be a 10-place number to the base 10^{10} . This viewpoint is justified for the same reason that we may convert, say, from binary to octal notation, simply by grouping the bits together. (See Eq. 4.1-5.)

In these terms, we are given the following primitive operations to work with:

- a₀) addition or subtraction of one-place integers, giving a one-place answer and a carry;

- b₀) multiplication of a one-place integer by another one-place integer, giving a two-place answer;
- c₀) division of a two-place integer by a one-place integer, provided that the quotient is a one-place integer, and yielding also a one-place remainder.

By adjusting the word size, if necessary, nearly all computers will have these three operations available; so we will construct algorithms (a), (b), and (c) mentioned above in terms of the primitive operations (a₀), (b₀), and (c₀).

Since we are visualizing extended-precision integers as base b numbers, it is sometimes helpful to think of the situation when $b = 10$, and to imagine that we are doing the arithmetic by hand. Then operation (a₀) is analogous to memorizing the addition table; (b₀) is analogous to memorizing the multiplication table; and (c₀) is essentially memorizing the multiplication table in reverse. The more complicated operations (a), (b), (c) on high-precision numbers can now be done using the simple addition, subtraction, multiplication, and long-division procedures we are taught in elementary school. In fact, most of the algorithms we shall discuss in this section are essentially nothing more than mechanizations of familiar pencil-and-paper operations. Of course, we must state the algorithms much more precisely than they have ever been stated in the fifth grade, and we should also attempt to minimize computer memory and running time requirements.

To avoid a tedious discussion and cumbersome notations, let us assume that all numbers we deal with are *nonnegative*. The additional work of computing the signs, etc., is quite straightforward, and the reader will find it easy to fill in any details of this sort.

First comes addition, which of course is very simple, but it is worth studying since the same ideas occur in the other algorithms also:

Algorithm A (Addition of nonnegative integers). Given nonnegative n -place integers $(u_1 u_2 \dots u_n)_b$ and $(v_1 v_2 \dots v_n)_b$, this algorithm forms their radix- b sum, $(w_0 w_1 w_2 \dots w_n)_b$. (Here w_0 is the “carry,” and it will always be equal to 0 or 1.)

- A1.** [Initialize.] Set $j \leftarrow n$, $k \leftarrow 0$. (The variable j will run through the various digit positions, and the variable k keeps track of carries at each step.)
- A2.** [Add digits.] Set $w_j \leftarrow (u_j + v_j + k) \bmod b$, and $k \leftarrow \lfloor (u_j + v_j + k)/b \rfloor$. (In other words, k is set to 1 or 0, depending on whether a “carry” occurs or not, i.e., whether $u_j + v_j + k \geq b$ or not. At most one carry is possible during the two additions, since we always have

$$u_j + v_j + k \leq (b-1) + (b-1) + 1 < 2b,$$

by induction on the computation.)

- A3.** [Loop on j .] Decrease j by one. Now if $j > 0$, go back to step A2; otherwise set $w_0 \leftarrow k$ and terminate the algorithm. ■

For a formal proof that Algorithm A is a valid, see exercise 4.

A MIX program for this addition process might take the following form:

Program A (Addition of nonnegative integers). Let $\text{LOC}(u_j) \equiv U + j$, $\text{LOC}(v_j) \equiv V + j$, $\text{LOC}(w_j) \equiv W + j$, $\text{rI1} \equiv j$, $\text{rA} \equiv k$, word size $\equiv b$, $N \equiv n$.

01	ENT1 N	1	<u>A1. Initialize.</u> $j \leftarrow n$.
02	JOV OFLO	1	Ensure overflow is off.
03	1H ENTA 0	$N + 1 - K$	$k \leftarrow 0$.
04	J1Z 3F	$N + 1 - K$	To A3 if $j = 0$.
05	2H ADD U,1	N	<u>A2. Add digits.</u>
06	ADD V,1	N	
07	STA W,1	N	
08	DEC1 1	N	<u>A3. Loop on j.</u>
09	JNOV 1B	N	If no overflow, set $k \leftarrow 0$.
10	ENTA 1	K	Otherwise, set $k \leftarrow 1$.
11	J1P 2B	K	To A2 if $j \neq 0$.
12	3H STA W	1	Store final carry in w_0 . ■

The running time for this program is $10N + 6$ cycles, independent of the number of carries, K . The quantity K is analyzed in detail at the close of this section.

Many modifications of Algorithm A are possible, and only a few of these are mentioned in the exercises below. A chapter on generalizations of this algorithm might be entitled “How to design addition circuits for a digital computer.”

The problem of subtraction is similar to addition, but the differences are worth noting:

Algorithm S (Subtraction of nonnegative integers). Given nonnegative n -place integers $(u_1 u_2 \dots u_n)_b \geq (v_1 v_2 \dots v_n)_b$, this algorithm forms their nonnegative radix- b difference, $(w_1 w_2 \dots w_n)_b$.

S1. [Initialize.] Set $j \leftarrow n$, $k \leftarrow 0$.

S2. [Subtract digits.] Set $w_j \leftarrow (u_j - v_j + k) \bmod b$, and $k \leftarrow \lfloor (u_j - v_j + k)/b \rfloor$. (In other words, k is set to -1 or 0 , depending on whether a “borrow” occurs or not, i.e., whether $u_j - v_j + k < 0$ or not. In the calculation of w_j , note that we must have $-b = 0 - (b - 1) + (-1) \leq u_j - v_j + k \leq (b - 1) - 0 + 0 < b$; hence $0 \leq u_j - v_j + k + b < 2b$, and this suggests the method of computer implementation explained below.)

S3. [Loop on j .] Decrease j by one. Now if $j > 0$, go back to step S2; otherwise terminate the algorithm. (When the algorithm terminates, we should have $k = 0$; the condition $k = -1$ will occur if and only if $v_1 \dots v_n > u_1 \dots u_n$, and this is contrary to the given assumptions. See exercise 12.) ■

In a MIX program to implement subtraction, it is most convenient to retain the value $1 + k$ instead of k throughout the algorithm, so that we can calculate $u_j - v_j + (1 + k) + (b - 1)$ in step S2. (Recall that b is the word size.) This is illustrated in the following code.

Program S (*Subtraction of nonnegative integers*). This program is analogous to the code in Program A; we have $rI1 \equiv j$, $rA \equiv 1 + k$. Here, as in other programs of this section, location WM1 contains the constant $b - 1$, the largest possible value that can be stored in a MIX word; cf. Program 4.2.3D, lines 38–39.

01	ENT1	N	1	<u>S1. Initialize.</u> $j \leftarrow n$.
02	JOV	OFLO	1	Ensure overflow is off.
03	1H	J1Z	DONE $K + 1$	Terminate if $j = 0$.
04	ENTA	1	K	Set $k \leftarrow 0$.
05	2H	ADD	U, 1 N	<u>S2. Subtract digits.</u>
06	SUB	V, 1	N	Compute $u_j - v_j + k + b$.
07	ADD	WM1	N	
08	STA	W, 1	N	(May be minus zero)
09	DEC1	1	N	<u>S3. Loop on j.</u>
10	JOV	1B	N	If overflow, set $k \leftarrow 0$.
11	ENTA	0	$N - K$	Otherwise, set $k \leftarrow -1$.
12	J1P	2B	$N - K$	Back to S2.
13	HLT	5		(Error, $v > u$) ■

The running time for this program is $12N + 3$ cycles, slightly longer than the corresponding amount for Program A.

The reader may wonder if it would not be worthwhile to have a combined addition-subtraction routine in place of the two algorithms A and S. But an examination of the computer programs shows that it is generally better to use two different routines, so that the inner loops of the computations can be performed as rapidly as possible, since the programs are so short.

Our next problem is multiplication, and here we carry the ideas used in Algorithm A a little further:

Algorithm M (*Multiplication of nonnegative integers*). Given nonnegative integers $(u_1 u_2 \dots u_n)_b$ and $(v_1 v_2 \dots v_m)_b$, this algorithm forms their radix- b product $(w_1 w_2 \dots w_{m+n})_b$. (The conventional pencil-and-paper method is based on forming the partial products $(u_1 u_2 \dots u_n) \times v_j$ first, for $1 \leq j \leq m$, and then adding these products together with appropriate scale factors; but in a computer it is best to do the addition concurrently with the multiplication, as described in this algorithm.)

M1. [Initialize.] Set $w_{m+1}, w_{m+2}, \dots, w_{m+n}$ all to zero. Set $j \leftarrow m$. (If w_{m+1}, \dots, w_{m+n} were not cleared to zero in this step, it turns out that the steps below would set

$$(w_1 \dots w_{m+n})_b \leftarrow (u_1 \dots u_n)_b \times (v_1 \dots v_m)_b + (w_{m+1} \dots w_{m+n})_b.$$

This more general operation is sometimes useful.)

M2. [Zero multiplier?] If $v_j = 0$, set $w_j \leftarrow 0$ and go to step M6. (This test saves a good deal of time if there is a reasonable chance that v_j is zero, but otherwise it may be omitted without affecting the validity of the algorithm.)

Table 1

MULTIPLICATION OF 914 BY 84.

Step	i	j	u_i	v_j	t	w_1	w_2	w_3	w_4	w_5
M5	3	2	4	4	16	x	x	0	0	6
M5	2	2	1	4	05	x	x	0	5	6
M5	1	2	9	4	36	x	x	6	5	6
M6	0	2	x	4	36	x	3	6	5	6
M5	3	1	4	8	37	x	3	6	7	6
M5	2	1	1	8	17	x	3	7	7	6
M5	1	1	9	8	76	x	6	7	7	6
M6	0	1	x	8	76	7	6	7	7	6

M3. [Initialize i .] Set $i \leftarrow n$, $k \leftarrow 0$.**M4.** [Multiply and add.] Set $t \leftarrow u_i \times v_j + w_{i+j} + k$; then set $w_{i+j} \leftarrow t \bmod b$ and $k \leftarrow \lfloor t/b \rfloor$. (Here the “carry” k will always be in the range $0 \leq k < b$; see below.)**M5.** [Loop on i .] Decrease i by one. Now if $i > 0$, go back to step M4; otherwise set $w_j \leftarrow k$.**M6.** [Loop on j .] Decrease j by one. Now if $j > 0$, go back to step M2; otherwise the algorithm terminates. ■

Algorithm M is illustrated in Table 1, assuming that $b = 10$, by showing the states of the computation at the beginning of steps M5 and M6. A proof of Algorithm M appears in the answer to exercise 14.

The two inequalities

$$0 \leq t < b^2, \quad 0 \leq k < b \quad (1)$$

are crucial for an efficient implementation of this algorithm, since they point out how large a register is needed for the computations. These inequalities may be proved by induction as the algorithm proceeds, for if we have $k < b$ at the start of step M4, we have

$$u_i \times v_j + w_{i+j} + k \leq (b-1) \times (b-1) + (b-1) + (b-1) = b^2 - 1 < b^2.$$

The following MIX program shows the considerations that are necessary when Algorithm M is implemented on a computer. The coding for step M4 would be a little simpler if our computer had a “multiply-and-add” instruction, or if it had a double-length accumulator for addition.

Program M. (*Multiplication of nonnegative integers*). This program is analogous to Program A. $rI1 \equiv i$, $rI2 \equiv j$, $rI3 \equiv i + j$, $\text{CONTENTS(CARRY)} \equiv k$.

01	ENT1 N	1	<u>M1. Initialize.</u>
02	JOV OFLO	1	Ensure overflow is off.
03	STZ W+M, 1	N	$w_{m+i} \leftarrow 0$.
04	DEC1 1	N	
05	J1P *-2	N	Repeat for $n \geq i > 0$.

06	ENT2 M	1	$j \leftarrow m$.
07	1H LDX V,2	M	<u>M2. Zero multiplier?</u>
08	JXZ 8F	M	If $v_j = 0$, set $w_j \leftarrow 0$ and go to M6.
09	ENT1 N	M - Z	<u>M3. Initialize i.</u>
10	ENT3 N,2	M - Z	$i \leftarrow n$, $(i + j) \leftarrow (n + j)$.
11	ENTX 0	M - Z	$k \leftarrow 0$.
12	2H STX CARRY	$(M - Z)N$	<u>M4. Multiply and add.</u>
13	LDA U,1	$(M - Z)N$	
14	MUL V,2	$(M - Z)N$	$rAX \leftarrow u_i \times v_j$.
15	SLC 5	$(M - Z)N$	Interchange $rA \leftrightarrow rX$.
16	ADD W,3	$(M - Z)N$	Add w_{i+j} to lower half.
17	JNOV **2	$(M - Z)N$	Did overflow occur?
18	INCX 1	K	If so, carry 1 into upper half.
19	ADD CARRY	$(M - Z)N$	Add k to lower half.
20	JNOV **2	$(M - Z)N$	Did overflow occur?
21	INCX 1	K'	If so, carry 1 into upper half.
22	STA W,3	$(M - Z)N$	$w_{i+j} \leftarrow t \bmod b$.
23	DEC1 1	$(M - Z)N$	<u>M5. Loop on i.</u>
24	DEC3 1	$(M - Z)N$	Decrease i and $(i + j)$ by 1.
25	J1P 2B	$(M - Z)N$	Back to M4 if $i > 0$; $rX = \lfloor t/b \rfloor$.
26	8H STX W,2	M	Set $w_j \leftarrow k$.
27	DEC2 1	M	<u>M6. Loop on j.</u>
28	J2P 1B	M	Repeat until $j = 0$. ■

The execution time of Program M depends on the number of places, M , in the multiplier v ; the number of places, N , in the multiplicand u ; the number of zeros, Z , in the multiplier; and the number of carries, K and K' , that occur during the addition to the lower half of the product in the computation of t . If we approximate both K and K' by the reasonable (although somewhat pessimistic) values $\frac{1}{2}(M - Z)N$, we find that the total running time comes to $28MN + 10M + 4N + 3 - Z(28N + 3)$ cycles. If step M2 were deleted, the running time would be $28MN + 7M + 4N + 3$ cycles, so this step is not advantageous unless the density of zero positions within the multiplier is $Z/M > 3/(28N + 3)$. If the multiplier is chosen completely at random, this ratio Z/M is expected to be only about $1/b$, which is extremely small; so step M2 is usually *not* worthwhile, unless b is small.

Algorithm M is not the fastest way to multiply when m and n are large, although it has the advantage of simplicity. Speedier methods are discussed in Section 4.3.3; even when $m = n = 4$, it is possible to multiply numbers in a little less time than is required by Algorithm M.

The final algorithm of concern to us in this section is long division, in which we want to divide $(n+m)$ -place integers by n -place integers. Here the ordinary pencil-and-paper method involves a certain amount of guesswork and ingenuity on the part of the person doing the division; we must either eliminate this guesswork from the algorithm or develop some theory to explain it more carefully.

A moment's reflection about the ordinary process of long division shows that the general problem breaks down into simpler steps, each of which is the division

Fig. 6. Wanted: a way to determine q rapidly.

$$\begin{array}{c} q \\ \overline{v_1 v_2 \dots v_n) u_0 u_1 u_2 \dots u_n} \\ \xleftarrow{\quad qv \quad} \\ \xleftarrow{\quad r \quad} \end{array}$$

of an $(n+1)$ -place number u by the n -place divisor v , where $0 \leq u/v < b$; the remainder r after each step is less than v , so we may use the quantity $rb +$ (next place of dividend) as the new u in the succeeding step. For example, if we are asked to divide 3142 by 47, we first divide 314 by 47, getting 6 and a remainder of 32; then we divide 322 by 47, getting 6 and a remainder of 40; thus we have a quotient of 66 and a remainder of 40. It is clear that this same idea works in general, and so our search for an appropriate division algorithm reduces to the following problem (Fig. 6):

Let $u = (u_0 u_1 \dots u_n)_b$ and $v = (v_1 v_2 \dots v_n)_b$ be nonnegative integers in radix- b notation, such that $u/v < b$. Find an algorithm to determine $q = \lfloor u/v \rfloor$.

We may observe that the condition $u/v < b$ is equivalent to the condition that $u/b < v$; i.e., $\lfloor u/b \rfloor < v$. This is simply the condition that $(u_0 u_1 \dots u_{n-1})_b < (v_1 v_2 \dots v_n)_b$. Furthermore, if we write $r = u - qv$, then q is the unique integer such that $0 \leq r < v$.

The most obvious approach to this problem is to make a guess about q , based on the most significant digits of u and v . It isn't obvious that such a method will be reliable enough, but it is worth investigating; let us therefore set

$$\hat{q} = \min \left(\left\lfloor \frac{u_0 b + u_1}{v_1} \right\rfloor, b - 1 \right). \quad (2)$$

This formula says \hat{q} is obtained by dividing the two leading digits of u by the leading digit of v ; and if the result is b or more we can replace it by $(b-1)$.

It is a remarkable fact, which we will now investigate, that this value \hat{q} is always a very good approximation to the desired answer q , so long as v_1 is reasonably large. In order to analyze how close \hat{q} comes to q , we will first prove that \hat{q} is never too small.

Theorem A. In the notation above, $\hat{q} \geq q$.

Proof. Since $q \leq b-1$, the theorem is certainly true if $\hat{q} = b-1$. Otherwise we have $\hat{q} = \lfloor (u_0 b + u_1)/v_1 \rfloor$, hence $\hat{q}v_1 \geq u_0 b + u_1 - v_1 + 1$. It follows that

$$\begin{aligned} u - \hat{q}v &\leq u - \hat{q}v_1 b^{n-1} \leq u_0 b^n + \dots + u_n \\ &\quad - (u_0 b^n + u_1 b^{n-1} - v_1 b^{n-1} + b^{n-1}) \\ &= u_2 b^{n-2} + \dots + u_n - b^{n-1} + v_1 b^{n-1} < v_1 b^{n-1} \leq v. \end{aligned}$$

Since $u - \hat{q}v < v$, we must have $\hat{q} \geq q$. ■

We will now prove that \hat{q} cannot be much larger than q in practical situations. Assume that $\hat{q} \geq q + 3$. We have

$$\hat{q} \leq \frac{u_0 b + u_1}{v_1} = \frac{u_0 b^n + u_1 b^{n-1}}{v_1 b^{n-1}} \leq \frac{u}{v_1 b^{n-1}} < \frac{u}{v - b^{n-1}}.$$

(The case $v = b^{n-1}$ is impossible, for if $v = (100\dots0)_b$ then $q = \hat{q}$.) Furthermore, the relation $q > (u/v) - 1$ implies that

$$3 \leq \hat{q} - q < \frac{u}{v - b^{n-1}} - \frac{u}{v} + 1 = \frac{u}{v} \left(\frac{b^{n-1}}{v - b^{n-1}} \right) + 1.$$

Therefore

$$\frac{u}{v} > 2 \left(\frac{v - b^{n-1}}{b^{n-1}} \right) \geq 2(v_1 - 1).$$

Finally, since $b - 4 \geq \hat{q} - 3 \geq q = \lfloor u/v \rfloor \geq 2(v_1 - 1)$, we have $v_1 < \lfloor b/2 \rfloor$. This proves the result we seek:

Theorem B. *If $v_1 \geq \lfloor b/2 \rfloor$, then $\hat{q} - 2 \leq q \leq \hat{q}$.* ■

The most important part of this theorem is that the conclusion is independent of b ; no matter how large b is, the trial quotient \hat{q} will never be more than 2 in error.

The condition that $v_1 \geq \lfloor b/2 \rfloor$ is very much like a normalization requirement; in fact, it is exactly the condition of floating-binary normalization in a binary computer. One simple way to ensure that v_1 is sufficiently large is to multiply both u and v by $\lfloor b/(v_1 + 1) \rfloor$; this does not change the value of u/v , nor does it increase the number of places in v , and exercise 23 proves that it will always make the new value of v_1 large enough. (Note: Another way to normalize the divisor is discussed in exercise 28.)

Now that we have armed ourselves with all of these facts, we are in a position to write the desired long-division algorithm. This algorithm uses a slightly improved choice of \hat{q} in step D3, which guarantees that $q = \hat{q}$ or $\hat{q} - 1$; in fact, the improved choice of \hat{q} made here is almost always accurate.

Algorithm D (Division of nonnegative integers). Given nonnegative integers $u = (u_1 u_2 \dots u_{m+n})_b$ and $v = (v_1 v_2 \dots v_n)_b$, where $v_1 \neq 0$ and $n > 1$, we form the radix- b quotient $\lfloor u/v \rfloor = (q_0 q_1 \dots q_m)_b$ and the remainder $u \bmod v = (r_1 r_2 \dots r_n)_b$. (This notation is slightly different from that used in the above proofs. When $n = 1$, the simpler algorithm of exercise 16 should be used.)

D1. [Normalize.] Set $d \leftarrow \lfloor b/(v_1 + 1) \rfloor$. Then set $(u_0 u_1 u_2 \dots u_{m+n})_b$ equal to $(u_1 u_2 \dots u_{m+n})_b$ times d , and set $(v_1 v_2 \dots v_n)_b$ equal to $(v_1 v_2 \dots v_n)_b$ times d . (Note the introduction of a new digit position u_0 at the left of u_1 ; if $d = 1$, all we need to do in this step is to set $u_0 \leftarrow 0$. On a binary computer it may be preferable to choose d to be a power of 2 instead of using the value suggested here; any value of d that results in $v_1 \geq \lfloor b/2 \rfloor$ will suffice. See also exercise 37.)

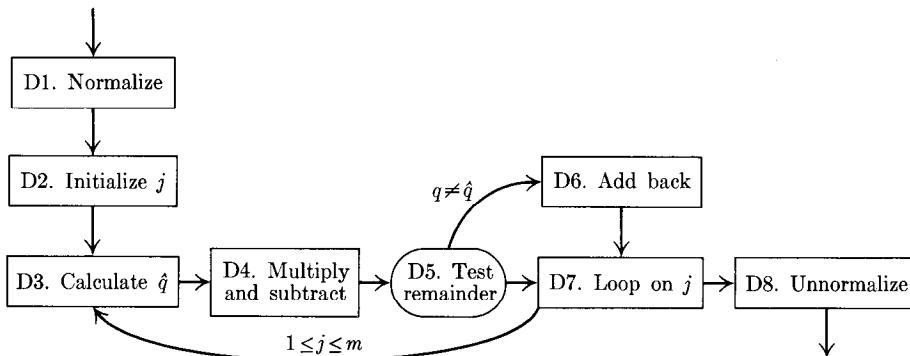


Fig. 7. Long division.

D2. [Initialize j .] Set $j \leftarrow 0$. (The loop on j , steps D2 through D7, will be essentially a division of $(u_j u_{j+1} \dots u_{j+n})_b$ by $(v_1 v_2 \dots v_n)_b$ to get a single quotient digit q_j ; cf. Fig. 6.)

D3. [Calculate \hat{q} .] If $u_j = v_1$, set $\hat{q} \leftarrow b - 1$; otherwise set $\hat{q} \leftarrow \lfloor (u_j b + u_{j+1}) / v_1 \rfloor$. Now test if $v_2 \hat{q} > (u_j b + u_{j+1} - \hat{q} v_1) b + u_{j+2}$; if so, decrease \hat{q} by 1 and repeat this test. (The latter test determines at high speed most of the cases in which the trial value \hat{q} is one too large, and it eliminates all cases where \hat{q} is two too large; see exercises 19, 20, 21.)

D4. [Multiply and subtract.] Replace $(u_j u_{j+1} \dots u_{j+n})_b$ by $(u_j u_{j+1} \dots u_{j+n})_b$ minus \hat{q} times $(v_1 v_2 \dots v_n)_b$. This step (analogous to steps M3, M4, and M5 of Algorithm M) consists of a simple multiplication by a one-place number, combined with a subtraction. The digits $(u_j, u_{j+1}, \dots, u_{j+n})$ should be kept positive; if the result of this step is actually negative, $(u_j u_{j+1} \dots u_{j+n})_b$ should be left as the true value plus b^{n+1} , i.e., as the b 's complement of the true value, and a “borrow” to the left should be remembered.

D5. [Test remainder.] Set $q_j \leftarrow \hat{q}$. If the result of step D4 was negative, go to step D6; otherwise go on to step D7.

D6. [Add back.] (The probability that this step is necessary is very small, on the order of only $2/b$, as shown in exercise 21; test data that activates this step should therefore be specifically contrived when debugging.) Decrease q_j by 1, and add $(0 v_1 v_2 \dots v_n)_b$ to $(u_j u_{j+1} u_{j+2} \dots u_{j+n})_b$. (A carry will occur to the left of u_j , and it should be ignored since it cancels with the “borrow” that occurred in D4.)

D7. [Loop on j .] Increase j by one. Now if $j \leq m$, go back to D3.

D8. [Unnormalize.] Now $(q_0 q_1 \dots q_m)_b$ is the desired quotient, and the desired remainder may be obtained by dividing $(u_{m+1} \dots u_{m+n})_b$ by d . ■

The representation of Algorithm D as a MIX program has several points of interest:

Program D (*Division of nonnegative integers*). The conventions of this program are analogous to Program A; rI1 $\equiv i$, rI2 $\equiv j - m$, rI3 $\equiv i + j$.

001	D1	JOV	OFLO	1	<u>D1. Normalize.</u> (See exercise 25)
...					
039	D2	ENN2	M	1	<u>D2. Initialize j.</u> Set $v_0 \leftarrow 0$, for convenience in D4.
040		STZ	V	1	
041	D3	LDA	U+M,2(1:5)	$M + 1$	<u>D3. Calculate \hat{q}.</u>
042		LDX	U+M+1,2	$M + 1$	$rAX \leftarrow u_j b + u_{j+1}$.
043		DIV	V+1	$M + 1$	$rA \leftarrow \lfloor rAX/v_1 \rfloor$.
044		JOV	1F	$M + 1$	Jump if quotient = b .
045		STA	QHAT	$M + 1$	$\hat{q} \leftarrow rA$.
046		STX	RHAT	$M + 1$	$\hat{r} \leftarrow u_j b + u_{j+1} - \hat{q}v_1$
047		JMP	2F	$M + 1$	$= (u_j b + u_{j+1}) \bmod v_1$.
048	1H	LDX	WM1		$rX \leftarrow b - 1$.
049		LDA	U+M+1,2		$rA \leftarrow u_{j+1}$. (Here $u_j = v_1$.)
050		JMP	4F		
051	3H	LDX	QHAT	E	
052		DECX	1	E	Decrease \hat{q} by one.
053		LDA	RHAT	E	Adjust \hat{r} accordingly:
054	4H	STX	QHAT	E	$\hat{q} \leftarrow rX$.
055		ADD	V+1	E	$rA \leftarrow rA + v_1$.
056		JOV	D4	E	(If \hat{r} will be $\geq b$, $v_2 \hat{q}$ will be $< \hat{r}b$.)
057		STA	RHAT	E	$\hat{r} \leftarrow rA$.
058		LDA	QHAT	E	
059	2H	MUL	V+2	$M + E + 1$	
060		CMPA	RHAT	$M + E + 1$	Test if $v_2 \hat{q} \leq \hat{r}b + u_{j+2}$.
061		JL	D4	$M + E + 1$	
062		JG	3B	E	
063		CMPX	U+M+2,2		If not, \hat{q} is too large.
064		JG	3B		<u>D4. Multiply and subtract.</u>
065	D4	ENTX	1	$M + 1$	
066		ENT1	N	$M + 1$	$i \leftarrow n$.
067		ENT3	M+N,2	$M + 1$	$(i + j) \leftarrow (n + j)$.
068	2H	STX	CARRY	$(M + 1)(N + 1)$	(Here $1 - b < rX \leq +1$.)
069		LDAN	V,1	$(M + 1)(N + 1)$	$rAX \leftarrow -\hat{q}v_i$.
070		MUL	QHAT	$(M + 1)(N + 1)$	Interchange $rA \leftrightarrow rX$.
071		SLC	5	$(M + 1)(N + 1)$	Add the contribution from the
072		ADD	CARRY	$(M + 1)(N + 1)$	digit to the right, plus 1.
073		JNOV	**2	$(M + 1)(N + 1)$	If sum is $\leq -b$, carry -1 .
074		DECX	1	K	Add u_{i+j} .
075		ADD	U,3	$(M + 1)(N + 1)$	Add $b - 1$ to force + sign.
076		ADD	WM1	$(M + 1)(N + 1)$	If no overflow, carry -1 .
077		JNOV	**2	$(M + 1)(N + 1)$	$rX \equiv$ carry + 1.
078		INCX	1	K'	$u_{i+j} \leftarrow rA$ (may be minus zero).
079		STA	U,3	$(M + 1)(N + 1)$	
080		DEC1	1	$(M + 1)(N + 1)$	
081		DEC3	1	$(M + 1)(N + 1)$	
082		J1NN	2B	$(M + 1)(N + 1)$	Repeat for $n \geq i \geq 0$.

083	D5	LDA	QHAT	$M + 1$	<u>D5. Test remainder.</u>
084		STA	Q+M,2	$M + 1$	Set $q_j \leftarrow \hat{q}$.
085		JXP	D7	$M + 1$	(Here $rX = 0$ or 1, since $v_0 = 0$.)
086	D6	DECA	1		<u>D6. Add back.</u>
087		STA	Q+M,2		Set $q_j \leftarrow \hat{q} - 1$.
088		ENT1	N		$i \leftarrow n$.
089		ENT3	M+N,2		$(i + j) \leftarrow (n + j)$.
090	1H	ENTA	0		(This is essentially Program A.)
091	2H	ADD	U,3		
092		ADD	V,1		
093		STA	U,3		
094		DEC1	1		
095		DEC3	1		
096		JNOV	1B		
097		ENTA	1		
098		J1NN	2B		
099	D7	INC2	1	$M + 1$	<u>D7. Loop on j.</u>
100		J2NP	D3	$M + 1$	Repeat for $0 \leq j \leq m$.
101	D8	...			(See exercise 26) ■

Note how easily the rather complex-appearing calculations and decisions of step D3 can be handled inside the machine. Note also that the program for step D4 is analogous to Program M, except that the ideas of Program S have also been incorporated.

The running time for Program D can be estimated by considering the quantities M , N , E , K , and K' shown in the program. (These quantities ignore several situations that occur only with very low probability; for example, we may assume that lines 048–050, 063–064, and step D6 are never executed.) Here $M + 1$ is the number of words in the quotient; N is the number of words in the divisor; E is the number of times \hat{q} is adjusted downwards in step D3; K and K' are the number of times certain “carry” adjustments are made during the multiply-subtract loop. If we assume that $K + K'$ is approximately $(N + 1)(M + 1)$, and that E is approximately $\frac{1}{2}M$, we get a total running time of approximately

$$30MN + 30N + 89M + 111$$

cycles, plus $67N + 235M + 4$ more if $d > 1$. (The program segments of exercises 25 and 26 are included in these totals.) When M and N are large, this is only about seven percent longer than the time Program M takes to multiply the quotient by the divisor.

Further commentary on Algorithm D appears in the exercises at the close of this section.

It is possible to debug programs for multiple-precision arithmetic by using the multiplication and addition routines to check the result of the division routine, etc. The following type of test data is occasionally useful:

$$(t^m - 1)(t^n - 1) = t^{m+n} - t^n - t^m + 1.$$

If $m < n$, this number has the radix- t expansion

$$\underbrace{(t-1) \dots (t-1)}_{m-1 \text{ places}} \quad \underbrace{(t-2) \dots (t-1)}_{n-m \text{ places}} \quad \underbrace{0 \dots 0}_{m-1 \text{ places}} \quad 1;$$

for example, $(10^3 - 1)(10^5 - 1) = 99899001$. In the case of Program D, it is also necessary to find some test cases that cause the rarely executed parts of the program to be exercised; some portions of that program would probably never get tested even if a million random test cases were tried.

Now that we have seen how to operate with signed-magnitude numbers, let us consider what approach should be taken to the same problems when a computer with complement notation is being used. For two's complement and ones' complement notations, it is usually best to let the radix b be one half of the word size; thus for a 32-bit computer word we would use $b = 2^{31}$ in the above algorithms. The sign bit of all but the most significant word of a multiple-precision number will be zero, so that no anomalous sign correction takes place during the computer's multiplication and division operations. In fact, the basic meaning of complement notation requires that we consider all but the most significant word to be nonnegative. For example, assuming an 8-bit word, the two's complement number

11011111 1111110 1101011

(where the sign bit is shown only in the most significant word) is properly thought of as

$$-2^{21} + (1011111)_2 \cdot 2^{14} + (1111110)_2 \cdot 2^7 + (1101011)_2.$$

Addition of signed numbers is slightly easier when complement notations are being used, since the routine for adding n -place nonnegative integers can be used for arbitrary n -place integers; the sign appears only in the first word, so the less significant words may be added together irrespective of the actual sign. (Special attention must be given to the leftmost carry when ones' complement notation is being used, however; it must be added into the least significant word, and possibly propagated further to the left.) Similarly, we find that subtraction of signed numbers is slightly simpler with complement notation. On the other hand, multiplication and division seem to be done most easily by working with nonnegative quantities and doing suitable complementation operations beforehand to make sure that both operands are nonnegative; it may be possible to avoid this complementation by devising some tricks for working directly with negative numbers in a complement notation, and it is not hard to see how this could be done in double-precision multiplication, but care should be taken not to slow down the inner loops of the subroutines when high precision is required. Note that the product of two m -place numbers in two's complement notation may require $2m + 1$ places: the square of $(-b^m)$ is b^{2m} .

Let us now turn to an analysis of the quantity K that arises in Program A, i.e., the number of carries that occur when two n -place numbers are being added together. Although K has no effect on the total running time of Program A, it does affect the running time of the Program A's counterparts that deal with complement notations, and its analysis is interesting in itself as a significant application of generating functions.

Suppose that u and v are independent random n -place integers, uniformly distributed in the range $0 \leq u, v < b^n$. Let p_{nk} be the probability that exactly k carries occur in the addition of u to v , and that one of these carries occurs in the most significant position (so that $u + v \geq b^n$). Similarly, let q_{nk} be the probability that exactly k carries occur, but that there is no carry in the most significant position. Then it is not hard to see that

$$\begin{aligned} p_{0k} &= 0, & q_{0k} &= \delta_{0k}, & \text{for all } k; \\ p_{(n+1)(k+1)} &= \frac{b+1}{2b} p_{nk} + \frac{b-1}{2b} q_{nk}, \\ q_{(n+1)k} &= \frac{b-1}{2b} p_{nk} + \frac{b+1}{2b} q_{nk}; \end{aligned} \quad (3)$$

this happens because $(b-1)/2b$ is the probability that $u_1 + v_1 \geq b$ and $(b+1)/2b$ is the probability that $u_1 + v_1 + 1 \geq b$, when u_1 and v_1 are independently and uniformly distributed integers in the range $0 \leq u_1, v_1 < b$.

To obtain further information about these quantities p_{nk} and q_{nk} , we may set up the generating functions

$$P(z, t) = \sum_{k,n} p_{nk} z^k t^n, \quad Q(z, t) = \sum_{k,n} q_{nk} z^k t^n. \quad (4)$$

From (3) we have the basic relations

$$\begin{aligned} P(z, t) &= zt \left(\frac{b+1}{2b} P(z, t) + \frac{b-1}{2b} Q(z, t) \right), \\ Q(z, t) &= 1 + t \left(\frac{b-1}{2b} P(z, t) + \frac{b+1}{2b} Q(z, t) \right). \end{aligned}$$

These two equations are readily solved for $P(z, t)$ and $Q(z, t)$; and if we let

$$G(z, t) = P(z, t) + Q(z, t) = \sum_n G_n(z) t^n,$$

where $G_n(z)$ is the generating function for the total number of carries when n -place numbers are added, we find that

$$G(z, t) = (b - zt)/p(z, t), \quad \text{where } p(z, t) = b - \frac{1}{2}(1+b)(1+z)t + zt^2. \quad (5)$$

Note that $G(1, t) = 1/(1-t)$, and this checks with the fact that $G_n(1)$ must equal 1 (it is the sum of all the possible probabilities). Taking partial derivatives

of (5) with respect to z , we find that

$$\begin{aligned}\frac{\partial G}{\partial z} &= \sum_n G'_n(z)t^n = \frac{-t}{p(z,t)} + \frac{t(b-zt)(b+1-2t)}{2p(z,t)^2}; \\ \frac{\partial^2 G}{\partial z^2} &= \sum_n G''_n(z)t^n = \frac{-t^2(b+1-2t)}{p(z,t)^2} + \frac{t^2(b-zt)(b+1-2t)}{p(z,t)^3}.\end{aligned}$$

Now let us put $z = 1$ and expand in partial fractions:

$$\begin{aligned}\sum_n G'_n(1)t^n &= \frac{t}{2} \left(\frac{1}{(1-t)^2} - \frac{1}{(b-1)(1-t)} + \frac{1}{(b-1)(b-t)} \right), \\ \sum_n G''_n(1)t^n &= \frac{t^2}{2} \left(\frac{1}{(1-t)^3} - \frac{1}{(b-1)^2(1-t)} + \frac{1}{(b-1)^2(b-t)} \right. \\ &\quad \left. + \frac{1}{(b-1)(b-t)^2} \right).\end{aligned}$$

It follows that the average number of carries, i.e., the mean value of K , is

$$G'_n(1) = \frac{1}{2} \left(n - \frac{1}{b-1} \left(1 - \left(\frac{1}{b} \right)^n \right) \right); \quad (6)$$

the variance is

$$\begin{aligned}G''_n(1) + G'_n(1) - G'_n(1)^2 &= \frac{1}{4} \left(n + \frac{2n}{b-1} - \frac{2b+1}{(b-1)^2} + \frac{2b+2}{(b-1)^2} \left(\frac{1}{b} \right)^n - \frac{1}{(b-1)^2} \left(\frac{1}{b} \right)^{2n} \right). \quad (7)\end{aligned}$$

So the number of carries is just slightly less than $\frac{1}{2}n$ under these assumptions.

History and bibliography. The early history of the classical algorithms described in this section is left as an interesting project for the reader, and only the history of their implementation on computers will be traced here.

The use of 10^n as an assumed radix when multiplying large numbers on a desk calculator was discussed by D. N. Lehmer and J. P. Ballantine, *AMM* 30 (1923), 67–69.

Double-precision arithmetic on digital computers was first treated by J. von Neumann and H. H. Goldstine in their introductory notes on programming, originally published in 1947 [J. von Neumann, *Collected Works* 5, 142–151]. Theorems A and B above are due to D. A. Pope and M. L. Stein [*CACM* 3 (1960), 652–654], whose paper also contains a bibliography of earlier work on double-precision routines. Other ways of choosing the trial quotient \hat{q} have been discussed by A. G. Cox and H. A. Luther, *CACM* 4 (1961), 353 [divide by $v_1 + 1$ instead of v_1], and by M. L. Stein, *CACM* 7 (1964), 472–474 [divide by v_1 or

$v_1 + 1$ according to the magnitude of v_2]; E. V. Krishnamurthy [CACM 8 (1965), 179–181] showed that examination of the single-precision remainder in the latter method leads to an improvement over Theorem B. Krishnamurthy and Nandi [CACM 10 (1967), 809–813] suggested a way to replace the normalization and unnormalization operations of Algorithm D by a calculation of \hat{q} based on several leading digits of the operands. G. E. Collins and D. R. Musser have carried out an interesting analysis of the original Pope and Stein algorithm [Inf. Proc. Letters 6 (1977), 151–155].

Several alternative methods for division have also been suggested:

1) “Fourier division” [J. Fourier, *Analyse des équations déterminées* (Paris, 1831), §2.21]. This method, which was often used on desk calculators, essentially obtains each new quotient digit by increasing the precision of the divisor and the dividend at each step. Some rather extensive tests by the author have indicated that such a method is inferior to the divide-and-correct technique above, but there may be some applications in which Fourier division is practical. See D. H. Lehmer, AMM 33 (1926), 198–206; J. V. Uspensky, *Theory of Equations* (New York: McGraw-Hill, 1948), 159–164.

2) “Newton’s method” for evaluating the reciprocal of a number was extensively used in early computers when there was no single-precision division instruction. The idea is to find some initial approximation x_0 to the number $1/v$, then to let $x_{n+1} = 2x_n - vx_n^2$. This method converges rapidly to $1/v$, since $x_n = (1-\epsilon)/v$ implies that $x_{n+1} = (1-\epsilon^2)/v$. Convergence to third order, i.e., with ϵ replaced by $O(\epsilon^3)$ at each step, can be obtained using the formula

$$\begin{aligned} x_{n+1} &= x_n + x_n(1 - vx_n) + x_n(1 - vx_n)^2 \\ &= x_n(1 + (1 - vx_n)(1 + (1 - vx_n))), \end{aligned}$$

and similar formulas hold for fourth-order convergence, etc.; see P. Rabinowitz, CACM 4 (1961), 98. For calculations on extremely large numbers, Newton’s second-order method and subsequent multiplication by u can actually be considerably faster than Algorithm D, if we increase the precision of x_n at each step and if we also use the fast multiplication routines of Section 4.3.3. (See Algorithm 4.3.3D for details.) Some related iterative schemes have been discussed by E. V. Krishnamurthy, IEEE Trans. C-19 (1970), 227–231.

3) Division methods have also been based on the evaluation of

$$\frac{u}{v+\epsilon} = \frac{u}{v} \left(1 - \left(\frac{\epsilon}{v} \right) + \left(\frac{\epsilon}{v} \right)^2 - \left(\frac{\epsilon}{v} \right)^3 + \dots \right).$$

See H. H. Laughlin, AMM 37 (1930), 287–293. We have used this idea in the double-precision case (Eq. 4.2.3–3).

Besides the references just cited, the following early articles concerning multiple-precision arithmetic are of interest: High-precision routines for floating point calculations using ones’ complement arithmetic are described by A. H.

Stroud and D. Secrest, *Comp. J.* **6** (1963), 62–66. Extended-precision subroutines for use in FORTRAN programs are described by B. I. Blum, *CACM* **8** (1965), 318–320; and for use in ALGOL by M. Tienari and V. Suokonautio, *BIT* **6** (1966), 332–338. Arithmetic on integers with *unlimited* precision, making use of linked memory allocation techniques, has been elegantly described by G. E. Collins, *CACM* **9** (1966), 578–589. For a much larger repertoire of operations, including logarithms and trigonometric functions, see R. P. Brent, *ACM Trans. Math. Software* **4** (1978), 57–81.

We have restricted our discussion in this section to arithmetic techniques for use in computer programming. There are many algorithms for *hardware* implementation of arithmetic operations that are very interesting, but they appear to be inapplicable to computer programs for high-precision numbers; for example, see G. W. Reitwiesner, “Binary Arithmetic,” *Advances in Computers* **1** (New York: Academic Press, 1960), 231–308; O. L. MacSorley, *Proc. IRE* **49** (1961), 67–91; G. Metze, *IRE Trans. EC-11* (1962), 761–764; H. L. Garner, “Number Systems and Arithmetic,” *Advances in Computers* **6** (New York: Academic Press, 1965), 131–194. The minimum achievable execution time for hardware addition and multiplication operations has been investigated by S. Winograd, *JACM* **12** (1965), 277–285, **14** (1967), 793–802; by R. P. Brent, *IEEE Trans. C-19* (1970), 758–759; and by R. W. Floyd, *IEEE Symp. Found. Comp. Sci.* **16** (1975), 3–5.

EXERCISES

1. [42] Study the early history of the classical algorithms for arithmetic by looking up the writings of, say, Sun Tsū, al-Khwārizmī, al-Uqlīdisī, Fibonacci, and Robert Recorde, and by translating their methods as faithfully as possible into precise algorithmic notation.
2. [15] Generalize Algorithm A so that it does “column addition,” i.e., obtains the sum of m nonnegative n -place integers. (Assume that $m \leq b$.)
3. [21] Write a MIX program for the algorithm of exercise 2, and estimate its running time as a function of m and n .
4. [M21] Give a formal proof of the validity of Algorithm A, using the method of “inductive assertions” explained in Section 1.2.1.
5. [21] Algorithm A adds the two inputs by going from right to left, but sometimes the data is more readily accessible from left to right. Design an algorithm that produces the same answer as Algorithm A, but that generates the digits of the answer from left to right, going back to change previous values if a carry occurs to make a previous value incorrect. [Note: Early Hindu and Arabic manuscripts dealt with addition from left to right in this way, probably because it was customary to work from left to right on an abacus; the right-to-left addition algorithm was a refinement due to al-Uqlīdisī, perhaps because Arabic is written from right to left.]

► 6. [22] Design an algorithm that adds from left to right (as in exercise 5), but your algorithm should not store a digit of the answer until this digit cannot possibly be affected by future carries; there is to be no changing of any answer digit once it has been stored. [Hint: Keep track of the number of consecutive $(b - 1)$'s that have not yet been stored in the answer.] This sort of algorithm would be appropriate, for example, in a situation where the input and output numbers are to be read and written from left to right on magnetic tapes, or if they appear in straight linear lists.

7. [M26] Determine the average number of times the algorithm of exercise 5 will find that a carry makes it necessary to go back and change k digits of the partial answer, for $k = 1, 2, \dots, n$. (Assume that both inputs are independently and uniformly distributed between 0 and $b^n - 1$.)

8. [M26] Write a MIX program for the algorithm of exercise 5, and determine its average running time based on the expected number of carries as computed in the text.

► 9. [21] Generalize Algorithm A to obtain an algorithm that adds two n -place numbers in a *mixed-radix* number system, with bases b_0, b_1, \dots (from right to left). Thus the least significant digits lie between 0 and $b_0 - 1$, the next digits lie between 0 and $b_1 - 1$, etc.; cf. Eq. 4.1-9.

10. [18] Would Program S work properly if the instructions on lines 06 and 07 were interchanged? If the instructions on lines 05 and 06 were interchanged?

11. [10] Design an algorithm that compares two nonnegative n -place integers $u = (u_1 u_2 \dots u_n)_b$ and $v = (v_1 v_2 \dots v_n)_b$, to determine whether $u < v$, $u = v$, or $u > v$.

12. [16] Algorithm S assumes that we know which of the two input operands is the larger; if this information is not known, we could go ahead and perform the subtraction anyway, and we would find that an extra "borrow" is still present at the end of the algorithm. Design another algorithm that could be used (if there is a "borrow" present at the end of Algorithm S) to complement $(w_1 w_2 \dots w_n)_b$ and therefore to obtain the absolute value of the difference of u and v .

13. [21] Write a MIX program that multiplies $(u_1 u_2 \dots u_n)_b$ by v , where v is a single-precision number (i.e., $0 \leq v < b$), producing the answer $(w_0 w_1 \dots w_n)_b$. How much running time is required?

► 14. [M24] Give a formal proof of the validity of Algorithm M, using the method of "inductive assertions" explained in Section 1.2.1.

15. [M20] If we wish to form the product of two n -place fractions, $(.u_1 u_2 \dots u_n)_b \times (.v_1 v_2 \dots v_n)_b$, and to obtain only an n -place approximation $(.w_1 w_2 \dots w_n)_b$ to the result, Algorithm M could be used to obtain a $2n$ -place answer that is subsequently rounded to the desired approximation. But this involves about twice as much work as is necessary for reasonable accuracy, since the products $u_i v_j$ for $i + j > n + 2$ contribute very little to the answer.

Give an estimate of the maximum error that can occur, if these products $u_i v_j$ for $i + j > n + 2$ are not computed during the multiplication, but are assumed to be zero.

► 16. [20] Design an algorithm that divides a nonnegative n -place integer $(u_1 u_2 \dots u_n)_b$ by v , where v is a single-precision number (i.e., $0 < v < b$), producing the quotient $(w_1 w_2 \dots w_n)_b$ and remainder r .

17. [M20] In the notation of Fig. 6, assume that $v_1 \geq \lfloor b/2 \rfloor$; show that if $u_0 = v_1$, we must have $q = b - 1$ or $b - 2$.

18. [M20] In the notation of Fig. 6, show that if $q' = \lfloor (u_0 b + u_1)/(v_1 + 1) \rfloor$, then $q' \leq q$.

► 19. [M21] In the notation of Fig. 6, let \hat{q} be an approximation to q , and let $\hat{r} = u_0 b + u_1 - \hat{q} v_1$. Assume that $v_1 > 0$. Show that if $v_2 \hat{q} > b\hat{r} + u_2$, then $q < \hat{q}$. [Hint: Strengthen the proof of Theorem A by examining the influence of v_2 .]

20. [M22] Using the notation and assumptions of exercise 19, show that if $v_2 \hat{q} \leq b\hat{r} + u_2$, then $\hat{q} = q$ or $q = \hat{q} - 1$.

► 21. [M23] Show that if $v_1 \geq \lfloor b/2 \rfloor$, and if $v_2 \hat{q} \leq b\hat{r} + u_2$ but $\hat{q} \neq q$ in the notation of exercises 19 and 20, then $u \bmod v \geq (1 - 2/b)v$. (The latter event occurs with approximate probability $2/b$, so that when b is the word size of a computer we must have $q_j = \hat{q}$ in Algorithm D except in very rare circumstances.)

► 22. [24] Find an example of a four-digit number divided by a three-digit number for which step D6 is necessary in Algorithm D, when the radix b is 10.

23. [M23] Given that v and b are integers, and that $1 \leq v < b$, prove that we always have $\lfloor b/2 \rfloor \leq v \lfloor b/(v+1) \rfloor < (v+1) \lfloor b/(v+1) \rfloor \leq b$.

24. [M20] Using the law of the distribution of leading digits explained in Section 4.2.4, give an approximate formula for the probability that $d = 1$ in Algorithm D. (When $d = 1$, it is, of course, possible to omit most of the calculation in steps D1 and D8.)

25. [26] Write a MIX routine for step D1, which is needed to complete Program D.

26. [21] Write a MIX routine for step D8, which is needed to complete Program D.

27. [M20] Prove that at the beginning of step D8 in Algorithm D, the unnormalized remainder $(u_{m+1}u_{m+2}\dots u_{m+n})_b$ is always an exact multiple of d .

28. [M30] (A. Svoboda, *Stroje na Zpracování Informací* 9 (1963), 25–32.) Let $v = (v_1 v_2 \dots v_n)_b$ be any radix b integer, where $v_1 \neq 0$. Perform the following operations:

N1. If $v_1 < b/2$, multiply v by $\lfloor (b+1)/(v_1+1) \rfloor$. Let the result of this step be $(v_0 v_1 v_2 \dots v_n)_b$.

N2. If $v_0 = 0$, set $v \leftarrow v + (1/b) \lfloor b(b-v_1)/(v_1+1) \rfloor v$; let the result of this step be $(v_0 v_1 v_2 \dots v_n \cdot v_{n+1} \dots)_b$. Repeat step N2 until $v_0 \neq 0$.

Prove that step N2 will be performed at most three times, and that we must always have $v_0 = 1$, $v_1 = 0$ at the end of the calculations.

[Note: If u and v are both multiplied by the above constants, we do not change the value of the quotient u/v , and the divisor has been converted into the form $(10v_2 \dots v_n \cdot v_{n+1} v_{n+2} v_{n+3})_b$. This form of the divisor is very convenient because, in the notation of Algorithm D, we may simply take $\hat{q} = u_j$ as a trial divisor at the beginning of step D3, or $\hat{q} = b - 1$ when $(u_{j-1}, u_j) = (1, 0)$.]

29. [15] Prove or disprove: At the beginning of step D7 of Algorithm D, we always have $u_j = 0$.

► 30. [22] If memory space is limited, it may be desirable to use the same storage locations for both input and output during the performance of some of the algorithms in this section. Is it possible to have w_1, \dots, w_n stored in the same respective locations as u_1, \dots, u_n or v_1, \dots, v_n during Algorithm A or S? Is it possible to have the quotient q_0, \dots, q_m occupy the same locations as u_0, \dots, u_m in Algorithm D? Is there any permissible overlap of memory locations between input and output in Algorithm M?

31. [28] Assume that $b = 3$ and that $u = (u_1 \dots u_{m+n})_3$, $v = (v_1 \dots v_n)_3$ are integers in balanced ternary notation (cf. Section 4.1), $v_1 \neq 0$. Design a long-division algorithm that divides u by v , obtaining a remainder whose absolute value does not exceed $\frac{1}{2}|v|$. Try to find an algorithm that would be efficient if incorporated into the arithmetic circuitry of a balanced ternary computer.

32. [M40] Assume that $b = 2i$ and that u and v are complex numbers expressed in the quater-imaginary number system. Design algorithms that divide u by v , perhaps obtaining a suitable remainder of some sort, and compare their efficiency. [References: M. Nadler, CACM 4 (1961), 192–193; Z. Pawlak and A. Wakulicz, Bull. de l'Acad. Polonaise des Sciences, Classe III, 5 (1957), 233–236 (see also pp. 803–804); and exercise 4.1–15.]

33. [M40] Design an algorithm for taking square roots, analogous to Algorithm D and to the traditional pencil-and-paper method for extracting square roots.

34. [40] Develop a set of computer subroutines for doing the four arithmetic operations on arbitrary integers, putting no constraint on the size of the integers except for the implicit assumption that the total memory capacity of the computer should not be exceeded. (Use linked memory allocation, so that no time is wasted in finding room to put the results.)

35. [40] Develop a set of computer subroutines for “decuple-precision floating point” arithmetic, using excess 0, base b , nine-place floating point number representation, where b is the computer word size, and allowing a full word for the exponent. (Thus each floating point number is represented in 10 words of memory, and all scaling is done by moving full words instead of by shifting within the words.)

36. [M42] Compute the values of the fundamental constants listed in Appendix B to much higher precision than the 40-place values listed there. [Note: The first 100,000 digits of the decimal expansion of π were published by D. Shanks and J. W. Wrench, Jr., in Math. Comp. 16 (1962), 76–99. One million digits of π were computed by Jean Guilloud and Martine Bouyer of the French Atomic Energy Commission in 1974.]

► 37. [20] (E. Salamin.) Explain how to avoid the normalization and unnormalization steps of Algorithm D, when d is a power of 2 on a binary computer, without changing the sequence of trial quotient digits computed by that algorithm. (How can \hat{q} be computed in step D3 if the normalization of step D1 hasn't been done?)

*4.3.2. Modular Arithmetic

Another interesting alternative is available for doing arithmetic on large integer numbers, based on some simple principles of number theory. The idea is to have several “moduli” m_1, m_2, \dots, m_r that contain no common factors, and to work indirectly with “residues” $u \bmod m_1, u \bmod m_2, \dots, u \bmod m_r$ instead of directly with the number u .

For convenience in notation throughout this section, let

$$u_1 = u \bmod m_1, \quad u_2 = u \bmod m_2, \quad \dots, \quad u_r = u \bmod m_r. \quad (1)$$

It is easy to compute (u_1, u_2, \dots, u_r) from an integer number u by means of division; and it is important to note that no information is lost in this process,

since we can recompute u from (u_1, u_2, \dots, u_r) provided that u is not too large. For example, if $0 \leq u < v \leq 1000$, it is impossible to have $(u \bmod 7, u \bmod 11, u \bmod 13)$ equal to $(v \bmod 7, v \bmod 11, v \bmod 13)$. This is a consequence of the "Chinese remainder theorem" stated below.

We may therefore regard (u_1, u_2, \dots, u_r) as a new type of internal computer representation, a "modular representation," of the integer u .

The advantages of a modular representation are that addition, subtraction, and multiplication are very simple:

$$(u_1, \dots, u_r) + (v_1, \dots, v_r) = ((u_1 + v_1) \bmod m_1, \dots, (u_r + v_r) \bmod m_r), \quad (2)$$

$$(u_1, \dots, u_r) - (v_1, \dots, v_r) = ((u_1 - v_1) \bmod m_1, \dots, (u_r - v_r) \bmod m_r), \quad (3)$$

$$(u_1, \dots, u_r) \times (v_1, \dots, v_r) = ((u_1 \times v_1) \bmod m_1, \dots, (u_r \times v_r) \bmod m_r). \quad (4)$$

To derive (4), for example, we need to show that

$$uv \bmod m_j = (u \bmod m_j)(v \bmod m_j) \bmod m_j$$

for each modulus m_j . But this is a basic fact of elementary number theory: $x \bmod m_j = y \bmod m_j$ if and only if $x \equiv y \pmod{m_j}$; furthermore if $x \equiv x'$ and $y \equiv y'$, then $xy \equiv x'y' \pmod{m_j}$; hence $(u \bmod m_j)(v \bmod m_j) \equiv uv \pmod{m_j}$.

The disadvantages of a modular representation are that it is comparatively difficult to test whether a number is positive or negative or to test whether or not (u_1, \dots, u_r) is greater than (v_1, \dots, v_r) . It is also difficult to test whether or not overflow has occurred as the result of an addition, subtraction, or multiplication, and it is even more difficult to perform division. When these operations are required frequently in conjunction with addition, subtraction, and multiplication, the use of modular arithmetic can be justified only if fast means of conversion into and out of the modular representation are available. Therefore conversion between modular and positional notation is one of the principal topics of interest to us in this section.

The processes of addition, subtraction, and multiplication using (2), (3), and (4) are called residue arithmetic or *modular arithmetic*. The range of numbers that can be handled by modular arithmetic is equal to $m = m_1 m_2 \dots m_r$, the product of the moduli. Therefore we see that the amount of time required to add, subtract, or multiply n -digit numbers using modular arithmetic is essentially proportional to n (not counting the time to convert in and out of modular representation). This is no advantage at all when addition and subtraction are considered, but it can be a considerable advantage with respect to multiplication since the conventional method of the preceding section requires an execution time proportional to n^2 .

Moreover, on a computer that allows many operations to take place simultaneously, modular arithmetic can be a significant advantage even for addition and subtraction; the operations with respect to different moduli can all be done at the same time, so we obtain a substantial increase in speed. The same kind of

decrease in execution time could not be achieved by the conventional techniques discussed in the previous section, since carry propagation must be considered. Perhaps some day highly parallel computers will make simultaneous operations commonplace, so that modular arithmetic will be of significant importance in “real-time” calculations when a quick answer to a single problem requiring high precision is needed. (With highly parallel computers, it is often preferable to run k separate programs simultaneously, instead of running a *single* program k times as fast, since the latter alternative is more complicated but does not utilize the machine any more efficiently. “Real-time” calculations are exceptions that make the inherent parallelism of modular arithmetic more significant.)

Now let us examine the basic fact that underlies the modular representation of numbers:

Theorem C (Chinese Remainder Theorem). *Let m_1, m_2, \dots, m_r be positive integers that are relatively prime in pairs, i.e.,*

$$\gcd(m_j, m_k) = 1 \quad \text{when } j \neq k. \quad (5)$$

Let $m = m_1 m_2 \dots m_r$, and let a, u_1, u_2, \dots, u_r be integers. Then there is exactly one integer u that satisfies the conditions

$$a \leq u < a + m, \quad \text{and} \quad u \equiv u_j \pmod{m_j} \text{ for } 1 \leq j \leq r. \quad (6)$$

Proof. If $u \equiv v \pmod{m_j}$ for $1 \leq j \leq r$, then $u - v$ is a multiple of m_j for all j , so (5) implies that $u - v$ is a multiple of $m = m_1 m_2 \dots m_r$. This argument shows that there is at most one solution of (6). To complete the proof we must now show the existence of *at least* one solution, and this can be done in two simple ways:

METHOD 1 (“Nonconstructive” proof). As u runs through the m distinct values $a \leq u < a + m$, the r -tuples $(u \pmod{m_1}, \dots, u \pmod{m_r})$ must also run through m distinct values, since (6) has at most one solution. But there are exactly $m_1 m_2 \dots m_r$ possible r -tuples (v_1, \dots, v_r) such that $0 \leq v_j < m_j$. Therefore each r -tuple must occur exactly once, and there must be some value of u for which $(u \pmod{m_1}, \dots, u \pmod{m_r}) = (u_1, \dots, u_r)$.

METHOD 2 (“Constructive” proof). We can find numbers M_j for $1 \leq j \leq r$ such that

$$M_j \equiv 1 \pmod{m_j} \quad \text{and} \quad M_j \equiv 0 \pmod{m_k} \text{ for } k \neq j. \quad (7)$$

This follows because (5) implies that m_j and m/m_j are relatively prime, so we may take

$$M_j = (m/m_j)^{\varphi(m_j)} \quad (8)$$

by Euler’s theorem (exercise 1.2.4-28). Now the number

$$u = a + ((u_1 M_1 + u_2 M_2 + \dots + u_r M_r - a) \pmod{m}) \quad (9)$$

satisfies all the conditions of (6). ■

A very special case of this theorem was stated by the Chinese mathematician Sun Tsü, who gave a rule called tái-yen (“great generalization”). The date of his writing is very uncertain; it is thought to be between 280 and 473 A.D. [See Joseph Needham, *Science and Civilization in China 3* (Cambridge University Press, 1959), 33–34, 119–120, for an interesting discussion.] Theorem C was apparently first stated and proved in its proper generality by Chhin Chiu Shao in his *Shu Shu Chiu Chang* (1247). Numerous early contributions to this theory have been summarized by L. E. Dickson in his *History of the Theory of Numbers* 2 (New York: Chelsea, 1952), 57–64.

As a consequence of Theorem C, we may use modular representation for numbers in any consecutive interval of $m = m_1 m_2 \dots m_r$ integers. For example, we could take $a = 0$ in (6), and work only with nonnegative integers u less than m . On the other hand, when addition and subtraction are being done, as well as multiplication, it is usually most convenient to assume that all the moduli m_1, m_2, \dots, m_r are odd numbers, so that $m = m_1 m_2 \dots m_r$ is odd, and to work with integers in the range

$$-\frac{m}{2} < u < \frac{m}{2}, \quad (10)$$

which is completely symmetrical about zero.

To perform the basic operations listed in (2), (3), and (4), we need to compute $(u_j + v_j) \bmod m_j$, $(u_j - v_j) \bmod m_j$, and $u_j v_j \bmod m_j$, when $0 \leq u_j, v_j < m_j$. If m_j is a single-precision number, it is most convenient to form $u_j v_j \bmod m_j$ by doing a multiplication and then a division operation. For addition and subtraction, the situation is a little simpler, since no division is necessary; the following formulas may conveniently be used:

$$(u_j + v_j) \bmod m_j = \begin{cases} u_j + v_j, & \text{if } u_j + v_j < m_j; \\ u_j + v_j - m_j, & \text{if } u_j + v_j \geq m_j. \end{cases} \quad (11)$$

$$(u_j - v_j) \bmod m_j = \begin{cases} u_j - v_j, & \text{if } u_j - v_j \geq 0; \\ u_j - v_j + m_j, & \text{if } u_j - v_j < 0. \end{cases} \quad (12)$$

(Cf. Section 3.2.1.1.) In this case, since we want m to be as large as possible, it is easiest to let m_1 be the largest odd number that fits in a computer word, to let m_2 be the largest odd number $< m_1$ that is relatively prime to m_1 , to let m_3 be the largest odd number $< m_2$ that is relatively prime to both m_1 and m_2 , and so on until enough m_j 's have been found to give the desired range m . Efficient ways to determine whether or not two integers are relatively prime are discussed in Section 4.5.2.

As a simple example, suppose that we have a decimal computer whose words hold only two digits, so that the word size is 100. Then the procedure described in the previous paragraph would give

$$m_1 = 99, \quad m_2 = 97, \quad m_3 = 95, \quad m_4 = 91, \quad m_5 = 89, \quad m_6 = 83, \quad (13)$$

and so on.

On binary computers it is sometimes desirable to choose the m_j in a different way, by selecting

$$m_j = 2^{e_j} - 1. \quad (14)$$

In other words, each modulus is one less than a power of 2. Such a choice of m_j often makes the basic arithmetic operations simpler, because it is relatively easy to work modulo $2^{e_j} - 1$, as in ones' complement arithmetic. When the moduli are chosen according to this strategy, it is helpful to relax the condition $0 \leq u_j < m_j$ slightly, so that we require only

$$0 \leq u_j < 2^{e_j}, \quad u_j \equiv u \pmod{2^{e_j} - 1}. \quad (15)$$

Thus, the value $u_j = m_j = 2^{e_j} - 1$ is allowed as an optional alternative to $u_j = 0$, since this does not affect the validity of Theorem C, and it means we are allowing u_j to be any e_j -bit binary number. Under this assumption, the operations of addition and multiplication modulo m_j become the following:

$$u_j \oplus v_j = \begin{cases} u_j + v_j, & \text{if } u_j + v_j < 2^{e_j}; \\ ((u_j + v_j) \bmod 2^{e_j}) + 1, & \text{if } u_j + v_j \geq 2^{e_j}. \end{cases} \quad (16)$$

$$u_j \otimes v_j = (u_j v_j \bmod 2^{e_j}) \oplus \lfloor u_j v_j / 2^{e_j} \rfloor. \quad (17)$$

(Here \oplus and \otimes refer to the operations done on the individual components of (u_1, \dots, u_r) and (v_1, \dots, v_r) when adding or multiplying, respectively, using the convention (15).) Equation (12) may be used for subtraction. These operations can be performed efficiently even when m_j is larger than the computer's word size, since it is a simple matter to compute the remainder of a positive number modulo a power of 2, or to divide a number by a power of 2. In (17) we have the sum of the "upper half" and the "lower half" of the product, as discussed in exercise 3.2.1.1-8.

If moduli of the form $2^{e_j} - 1$ are to be used, we must know under what conditions the number $2^e - 1$ is relatively prime to the number $2^f - 1$. Fortunately, there is a very simple rule,

$$\gcd(2^e - 1, 2^f - 1) = 2^{\gcd(e, f)} - 1, \quad (18)$$

which states in particular that $2^e - 1$ and $2^f - 1$ are relatively prime if and only if e and f are relatively prime. Equation (18) follows from Euclid's algorithm and the identity

$$(2^e - 1) \bmod (2^f - 1) = 2^{e \bmod f} - 1. \quad (19)$$

(See exercise 6.) Thus we could choose for example $m_1 = 2^{35} - 1$, $m_2 = 2^{34} - 1$, $m_3 = 2^{33} - 1$, $m_4 = 2^{31} - 1$, $m_5 = 2^{29} - 1$, if we had a computer with word size 2^{35} ; this would permit efficient addition, subtraction, and multiplication of integers in a range of size $m_1 m_2 m_3 m_4 m_5 > 2^{161}$.

As we have already observed, the operations of conversion to and from modular representation are very important. If we are given a number u , its modular representation (u_1, \dots, u_r) may be obtained by simply dividing u by m_1, \dots, m_r and saving the remainders. A possibly more attractive procedure, if $u = (v_m v_{m-1} \dots v_0)_b$, is to evaluate the polynomial

$$(\dots(v_m b + v_{m-1})b + \dots)b + v_0$$

using modular arithmetic. When $b = 2$ and when the modulus m_j has the special form $2^{e_j} - 1$, both of these methods reduce to quite a simple procedure: Consider the binary representation of u with blocks of e_j bits grouped together,

$$u = a_t A^t + a_{t-1} A^{t-1} + \dots + a_1 A + a_0, \quad (20)$$

where $A = 2^{e_j}$ and $0 \leq a_k < 2^{e_j}$ for $0 \leq k \leq t$. Then

$$u \equiv a_t + a_{t-1} + \dots + a_1 + a_0 \pmod{2^{e_j} - 1}, \quad (21)$$

since $A \equiv 1$, so we obtain u_j by adding the e_j -bit numbers $a_t \oplus \dots \oplus a_1 \oplus a_0$, using (16). This process is similar to the familiar device of “casting out nines” that determines $u \bmod 9$ when u is expressed in the decimal system.

Conversion back from modular form to positional notation is somewhat more difficult. It is interesting in this regard to make a few side remarks about the way computers make us change our viewpoint towards mathematical proofs: Theorem C tells us that the conversion from (u_1, \dots, u_r) to u is possible, and two proofs are given. The first proof we considered is a classical one that relies only on very simple concepts, namely the facts that

- i) any number that is a multiple of m_1 , of m_2, \dots , and of m_r , must be a multiple of $m_1 m_2 \dots m_r$ when the m_j 's are pairwise relatively prime; and
- ii) if m things are put into m boxes with no two things in the same box, then there must be one in each box.

By traditional notions of mathematical aesthetics, this is no doubt the nicest proof of Theorem C; but from a computational standpoint it is completely worthless. It amounts to saying, “Try $u = a, a + 1, \dots$ until you find a value for which $u \equiv u_1 \pmod{m_1}, \dots, u \equiv u_r \pmod{m_r}$.”

The second proof of Theorem C is more explicit; it shows how to compute r new constants M_1, \dots, M_r , and to get the solution in terms of these constants by formula (9). This proof uses more complicated concepts (for example, Euler's theorem), but it is much more satisfactory from a computational standpoint, since the constants M_1, \dots, M_r need to be determined only once. On the other hand, the determination of M_j by Eq. (8) is certainly not trivial, since the evaluation of Euler's φ -function requires, in general, the factorization of m_j into prime powers. Furthermore, M_j is likely to be a terribly large number, even if we compute only the quantity $M_j \bmod m$ (which will work just as well as M_j in (9)).

Since $M_j \bmod m$ is uniquely determined if (7) is to be satisfied (because of the Chinese remainder theorem), we can see that, in any event, Eq. (9) requires a lot of high-precision calculation, and such calculation is just what we wished to avoid by modular arithmetic in the first place.

So we need an even better proof of Theorem C if we are going to have a really usable method of conversion from (u_1, \dots, u_r) to u . Such a method was suggested by H. L. Garner in 1958; it can be carried out using $\binom{r}{2}$ constants c_{ij} for $1 \leq i < j \leq r$, where

$$c_{ij} m_i \equiv 1 \pmod{m_j}. \quad (22)$$

These constants c_{ij} are readily computed using Euclid's algorithm, since for any given i and j Algorithm 4.5.2X will determine a and b such that $am_i + bm_j = \gcd(m_i, m_j) = 1$, and we may take $c_{ij} = a$. When the moduli have the special form $2^{e_j} - 1$, a simple method of determining c_{ij} is given in exercise 6.

Once the c_{ij} have been determined satisfying (22), we can set

$$\begin{aligned} v_1 &\leftarrow u_1 \bmod m_1, \\ v_2 &\leftarrow (u_2 - v_1)c_{12} \bmod m_2, \\ v_3 &\leftarrow ((u_3 - v_1)c_{13} - v_2)c_{23} \bmod m_3, \\ &\vdots \\ v_r &\leftarrow (\dots((u_r - v_1)c_{1r} - v_2)c_{2r} - \dots - v_{r-1})c_{(r-1)r} \bmod m_r. \end{aligned} \quad (23)$$

Then

$$u = v_r m_{r-1} \dots m_2 m_1 + \dots + v_3 m_2 m_1 + v_2 m_1 + v_1 \quad (24)$$

is a number satisfying the conditions

$$0 \leq u < m, \quad u \equiv u_j \pmod{m_j} \quad \text{for } 1 \leq j \leq r. \quad (25)$$

(See exercise 8; another way of rewriting (23) that does not involve as many auxiliary constants is given in exercise 7.) Equation (24) is a *mixed-radix representation* of u , which may be converted to binary or decimal notation using the methods of Section 4.4. If $0 \leq u < m$ is not the desired range, an appropriate multiple of m can be added or subtracted after the conversion process.

The advantage of the computation shown in (23) is that the calculation of v_j can be done using only arithmetic $\bmod m_j$, which is already built into the modular arithmetic algorithms. Furthermore, (23) allows parallel computation: We can start with $(v_1, \dots, v_r) \leftarrow (u_1 \bmod m_1, \dots, u_r \bmod m_r)$, then at time j for $1 \leq j < r$ we simultaneously set $v_k \leftarrow (v_k - v_j)c_{jk} \bmod m_k$ for $j < k \leq r$. An alternative way to compute the mixed-radix representation, allowing similar possibilities for parallelism, has been discussed by A. S. Fraenkel, *Proc. ACM Nat. Conf.* 19 (Philadelphia, 1965), E1.4.

It is important to observe that the mixed-radix representation (24) is sufficient to compare the magnitudes of two modular numbers. For if we know that

$0 \leq u < m$ and $0 \leq u' < m$, then we can tell if $u < u'$ by first doing the conversion to (v_1, \dots, v_r) and (v'_1, \dots, v'_r) , then testing if $v_r < v'_r$, or if $v_r = v'_r$ and $v_{r-1} < v'_{r-1}$, etc. It is not necessary to convert all the way to binary or decimal notation if we only want to know whether (u_1, \dots, u_r) is less than (u'_1, \dots, u'_r) .

The operation of comparing two numbers, or of deciding if a modular number is negative, is intuitively very simple, so we would expect to have a much easier method for making this test than the conversion to mixed-radix form. But the following theorem shows that there is little hope of finding a substantially better method, since the range of a modular number depends essentially on all bits of all the residues (u_1, \dots, u_r) :

Theorem S (Nicholas Szabó, 1961). *In terms of the notation above, assume that $m_1 < \sqrt{m}$, and let L be any value in the range*

$$m_1 \leq L \leq m - m_1. \quad (26)$$

Let g be any function such that the set $\{g(0), g(1), \dots, g(m_1 - 1)\}$ contains fewer than m_1 values. Then there are numbers u and v such that

$$g(u \bmod m_1) = g(v \bmod m_1), \quad u \bmod m_j = v \bmod m_j \text{ for } 2 \leq j \leq r; \quad (27)$$

$$0 \leq u < L \leq v < m. \quad (28)$$

Proof. By hypothesis, there must exist numbers $u \neq v$ satisfying (27), since g must take on the same value for two different residues. Let (u, v) be a pair of values with $0 \leq u < v < m$ satisfying (27), for which u is a minimum. Since $u' = u - m_1$ and $v' = v - m_1$ also satisfy (27), we must have $u' < 0$ by the minimality of u . Hence $u < m_1 \leq L$; and if (28) does not hold, we must have $v < L$. But $v > u$, and $v - u$ is a multiple of $m_2 \dots m_r = m/m_1$, so $v \geq v - u \geq m/m_1 > m_1$. Therefore, if (28) does not hold for (u, v) , it will be satisfied for the pair $(u'', v'') = (v - m_1, u + m - m_1)$. ■

Of course, a similar result can be proved for any m_j in place of m_1 ; and we could also replace (28) by the condition " $a \leq u < a + L \leq v < a + m$ " with only minor changes in the proof. Therefore Theorem S shows that many simple functions cannot be used to determine the range of a modular number.

Let us now reiterate the main points of the discussion in this section: Modular arithmetic can be a significant advantage for applications in which the predominant calculations involve exact multiplication (or raising to a power) of large integers, combined with addition and subtraction, but where there is very little need to divide or compare numbers, or to test whether intermediate results "overflow" out of range. (It is important not to forget the latter restriction; methods are available to test for overflow, as in exercise 12, but they are in general so complicated that they nullify the advantages of modular arithmetic.) Several applications of modular computations have been discussed by H. Takahasi and Y. Ishibashi, *Information Proc. in Japan* 1 (1961), 28–42.

An example of such an application is the exact solution of linear equations with rational coefficients. For various reasons it is desirable in this case to assume that the moduli m_1, m_2, \dots, m_r are all large prime numbers; the linear equations can be solved independently modulo each m_j . A detailed discussion of this procedure has been given by I. Borosh and A. S. Fraenkel [*Math. Comp.* **20** (1966), 107–112]. By means of their method, the nine independent solutions of a system of 111 linear equations in 120 unknowns were obtained exactly in less than one hour's running time on a CDC 1604 computer. The same procedure is worthwhile also for solving simultaneous linear equations with floating point coefficients, when the matrix of coefficients is ill-conditioned. The modular technique (treating the given floating point coefficients as exact rational numbers) gives a method for obtaining the true answers in less time than conventional methods can produce reliable approximate answers! [See M. T. McClellan, *JACM* **20** (1973), 563–588, for further developments of this approach; and see also E. H. Bareiss, *J. Inst. Math. and Appl.* **10** (1972), 68–104, for a discussion of its limitations.]

The published literature concerning modular arithmetic is mostly oriented towards hardware design, since the carry-free properties of modular arithmetic make it attractive from the standpoint of high-speed operation. The idea was first published by A. Svoboda and M. Valach in the Czechoslovakian journal *Stroje na Zpracování Informací* **3** (1955), 247–295; then independently by H. L. Garner [*IRE Trans. EC-8* (1959), 140–147]. The use of moduli of the form $2^{e_j} - 1$ was suggested by A. S. Fraenkel [*JACM* **8** (1961), 87–96], and several advantages of such moduli were demonstrated by A. Schönhage [*Computing* **1** (1966), 182–196]. See the book *Residue Arithmetic and its Applications to Computer Technology* by N. S. Szabó and R. I. Tanaka (New York: McGraw-Hill, 1967), for additional information and a comprehensive bibliography of the subject. A Russian book published in 1968 by I. Īa. Akushskii and D. I. Īuditskii includes a chapter about complex moduli [cf. *Rev. Romaine des Math.* **15** (1970), 159–160].

Further discussion of modular arithmetic can be found in Section 4.3.3B.

EXERCISES

1. [20] Find all integers u that satisfy all of the following conditions: $u \bmod 7 = 1$, $u \bmod 11 = 6$, $u \bmod 13 = 5$, $0 \leq u < 1000$.
2. [M20] Would Theorem C still be true if we allowed a, u_1, u_2, \dots, u_r and u to be arbitrary real numbers (not just integers)?
- 3. [M26] (*Generalized Chinese Remainder Theorem.*) Let m_1, m_2, \dots, m_r be positive integers. Let m be the least common multiple of m_1, m_2, \dots, m_r , and let a, u_1, u_2, \dots, u_r be any integers. Prove that there is exactly one integer u that satisfies the conditions

$$a \leq u < a + m, \quad u \equiv u_j \pmod{m_j}, \quad 1 \leq j \leq r,$$

provided that

$$u_i \equiv u_j \pmod{\gcd(m_i, m_j)}, \quad 1 \leq i < j \leq r;$$

and there is no such integer u when the latter condition fails to hold.

4. [20] Continue the process shown in (13); what would m_7, m_8, m_9, \dots be?

► 5. [M23] Suppose that the method of (13) is continued until no more m_j can be chosen; does this method give the largest attainable value $m_1 m_2 \dots m_r$ such that the m_j are odd positive integers less than 100 that are relatively prime in pairs?

6. [M22] Let e, f, g be nonnegative integers. (a) Show that $2^e \equiv 2^f \pmod{2^g - 1}$ if and only if $e \equiv f \pmod{g}$. (b) Given that $e \pmod{f} = d$ and $ce \pmod{f} = 1$, prove that

$$((1 + 2^d + \dots + 2^{(c-1)d}) \cdot (2^e - 1)) \pmod{2^f - 1} = 1.$$

(Thus, we have a comparatively simple formula for the inverse of $2^e - 1$, modulo $2^f - 1$, as required in (22).)

► 7. [M21] Show that (23) can be rewritten as follows:

$$v_1 \leftarrow u_1 \pmod{m_1},$$

$$v_2 \leftarrow (u_2 - v_1)c_{12} \pmod{m_2},$$

$$v_3 \leftarrow (u_3 - (v_1 + m_1 v_2))c_{13}c_{23} \pmod{m_3},$$

⋮

$$v_r \leftarrow (u_r - (v_1 + m_1(v_2 + m_2(v_3 + \dots + m_{r-2}v_{r-1}) \dots)))c_{1r} \dots c_{(r-1)r} \pmod{m_r}.$$

If the formulas are rewritten in this way, we see that only $r - 1$ constants $C_j = c_{1j} \dots c_{(j-1)j} \pmod{m_j}$ are needed instead of $r(r - 1)/2$ constants c_{ij} as in (23). Discuss the relative merits of this version of the formula as compared to (23), from the standpoint of computer calculation.

8. [M21] Prove that the number u defined by (23) and (24) satisfies (25).

9. [M20] Show how to go from the values v_1, \dots, v_r of the mixed-radix notation (24) back to the original residues u_1, \dots, u_r , using only arithmetic mod m_j to compute the value of u_j .

10. [M25] An integer u that lies in the symmetrical range (10) might be represented by finding the numbers u_1, \dots, u_r such that $u \equiv u_j \pmod{m_j}$ and $-m_j/2 < u_j < m_j/2$, instead of insisting that $0 \leq u_j < m_j$ as in the text. Discuss the modular arithmetic procedures that would be appropriate in connection with such a symmetrical representation (including the conversion process, (23)).

11. [M23] Assume that all the m_j are odd, and that $u = (u_1, \dots, u_r)$ is known to be even, where $0 \leq u_j < m_j$. Find a reasonably fast method to compute $u/2$ using modular arithmetic.

12. [M10] Prove that, if $0 \leq u, v < m$, the modular addition of u and v causes overflow (i.e., is outside the range allowed by the modular representation) if and only if the sum is less than u . (Thus the overflow detection problem is equivalent to the comparison problem.)

► 13. [M25] (*Automorphic numbers.*) An n -place decimal number $x > 1$ is called an “automorph” by recreational mathematicians if the last n digits of x^2 are equal to x . For example, 9376 is a 4-place automorph, since $9376^2 = 87909376$. [See *Scientific American* 218 (January 1968), 125.]

- Prove that an n -place number $x > 1$ is an automorph if and only if $x \bmod 5^n = 0$ or 1 and $x \bmod 2^n = 1$ or 0, respectively. (Thus, if $m_1 = 2^n$ and $m_2 = 5^n$, the only two n -place automorphs are the numbers M_1 and M_2 in (7).)
- Prove that if x is an n -place automorph, then $(3x^2 - 2x^3) \bmod 10^{2n}$ is a $2n$ -place automorph.
- Given that $cx \equiv 1$ (modulo y), find a simple formula for a number c' depending on c and x but not on y , such that $c'x^2 \equiv 1$ (modulo y^2).

*4.3.3. How Fast Can We Multiply?

The conventional method for multiplication in positional number systems, Algorithm 4.3.1M, requires approximately cmn operations to multiply an m -digit number by an n -digit number, where c is a constant. In this section, let us assume for convenience that $m = n$, and let us consider the following question: *Does every general computer algorithm for multiplying two n -digit numbers require an execution time proportional to n^2 , as n increases?*

(In this question, a “general” algorithm means one that accepts, as input, the number n and two arbitrary n -digit numbers in positional notation, and whose output is their product in positional form. Certainly if we were allowed to choose a different algorithm for each value of n , the question would be of no interest, since multiplication could be done for any specific value of n by a “table-lookup” operation in some huge table. The term “computer algorithm” is meant to imply an algorithm that is suitable for implementation on a digital computer such as MIX, and the execution time is to be the time it takes to perform the algorithm on such a computer.)

A. Digital methods. The answer to the above question is, rather surprisingly, “No,” and, in fact, it is not very difficult to see why. For convenience, let us assume throughout this section that we are working with integers expressed in binary notation. If we have two $2n$ -bit numbers $u = (u_{2n-1} \dots u_1 u_0)_2$ and $v = (v_{2n-1} \dots v_1 v_0)_2$, we can write

$$u = 2^n U_1 + U_0, \quad v = 2^n V_1 + V_0, \quad (1)$$

where $U_1 = (u_{2n-1} \dots u_n)_2$ is the “most significant half” of the number u and $U_0 = (u_{n-1} \dots u_0)_2$ is the “least significant half”; similarly $V_1 = (v_{2n-1} \dots v_n)_2$ and $V_0 = (v_{n-1} \dots v_0)_2$. Now we have

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_0 - V_1) + (2^n + 1)U_0V_0. \quad (2)$$

This formula reduces the problem of multiplying $2n$ -bit numbers to three multiplications of n -bit numbers, namely U_1V_1 , $(U_1 - U_0)(V_0 - V_1)$, and U_0V_0 , plus some simple shifting and adding operations.

Formula (2) can be used for double-precision multiplication when we want a quadruple-precision result, and it will be just a little faster than the traditional method on many machines. But the main advantage of (2) is that we can use it to define a recursive process for multiplication that is significantly faster than the familiar order- n^2 method when n is large: If $T(n)$ is the time required to perform multiplication of n -bit numbers, we have

$$T(2n) \leq 3T(n) + cn \quad (3)$$

for some constant c , since the right-hand side of (2) uses just three multiplications plus some additions and shifts. Relation (3) implies by induction that

$$T(2^k) \leq c(3^k - 2^k), \quad k \geq 1, \quad (4)$$

if we choose c to be large enough so that this inequality is valid when $k = 1$; therefore we have

$$\begin{aligned} T(n) &\leq T(2^{\lceil \lg n \rceil}) \leq c(3^{\lceil \lg n \rceil} - 2^{\lceil \lg n \rceil}) \\ &< 3c \cdot 3^{\lg n} = 3cn^{\lg 3}. \end{aligned} \quad (5)$$

Relation (5) shows that the running time for multiplication can be reduced from order n^2 to order $n^{\lg 3} \approx n^{1.585}$, so the recursive method is much faster than the traditional method when n is large.

(A similar but more complicated method for doing multiplication with running time of order $n^{\lg 3}$ was apparently first suggested by A. Karatsuba in *Doklady Akad. Nauk SSSR* 145 (1962), 293–294 [English translation in *Soviet Physics-Doklady* 7 (1963), 595–596]. Curiously, this idea does not seem to have been discovered before 1962; none of the “calculating prodigies” who have become famous for their ability to multiply large numbers mentally have been reported to use any such method, although formula (2) adapted to decimal notation would seem to lead to a reasonably easy way to multiply eight-digit numbers in one’s head.)

The running time can be reduced still further, in the limit as n approaches infinity, if we observe that the method just used is essentially the special case $r = 1$ of a more general method that yields

$$T((r+1)n) \leq (2r+1)T(n) + cn \quad (6)$$

for any fixed r . This more general method can be obtained as follows: Let

$$u = (u_{(r+1)n-1} \dots u_1 u_0)_2 \quad \text{and} \quad v = (v_{(r+1)n-1} \dots v_1 v_0)_2$$

be broken into $r+1$ pieces,

$$u = U_r 2^{rn} + \dots + U_1 2^n + U_0, \quad v = V_r 2^{rn} + \dots + V_1 2^n + V_0, \quad (7)$$

where each U_j and each V_j is an n -bit number. Consider the polynomials

$$U(x) = U_r x^r + \cdots + U_1 x + U_0, \quad V(x) = V_r x^r + \cdots + V_1 x + V_0, \quad (8)$$

and let

$$W(x) = U(x)V(x) = W_{2r}x^{2r} + \cdots + W_1 x + W_0. \quad (9)$$

Since $u = U(2^n)$ and $v = V(2^n)$, we have $uv = W(2^n)$, so we can easily compute uv if we know the coefficients of $W(x)$. The problem is to find a good way to compute the coefficients of $W(x)$ by using only $2r+1$ multiplications of n -bit numbers plus some further operations that involve only an execution time proportional to n . This can be done by computing

$$U(0)V(0) = W(0), \quad U(1)V(1) = W(1), \quad \dots, \quad U(2r)V(2r) = W(2r). \quad (10)$$

The coefficients of a polynomial of degree $2r$ can be written as a linear combination of the values of that polynomial at $2r+1$ distinct points; computing such a linear combination requires an execution time at most proportional to n . (Actually, the products $U(j)V(j)$ are not strictly products of n -bit numbers, but they are products of at most $(n+t)$ -bit numbers, where t is a fixed value depending on r . It is easy to design a multiplication routine for $(n+t)$ -bit numbers that requires only $T(n) + c_1 n$ operations, where $T(n)$ is the number of operations needed for n -bit multiplications, since two products of t -bit by n -bit numbers can be done in $c_2 n$ operations when t is fixed.) Therefore we obtain a method of multiplication satisfying (6).

Relation (6) implies that $T(n) \leq c_3 n^{\log_{r+1}(2r+1)} < c_3 n^{1+\log_{r+1} 2}$, if we argue as in the derivation of (5), so we have now proved the following result:

Theorem A. Given $\epsilon > 0$, there exists a multiplication algorithm such that the number of elementary operations $T(n)$ needed to multiply two n -bit numbers satisfies

$$T(n) < c(\epsilon)n^{1+\epsilon}, \quad (11)$$

for some constant $c(\epsilon)$ independent of n . ■

This theorem is still not the result we are after. It is unsatisfactory for practical purposes in that the method becomes much more complicated as $\epsilon \rightarrow 0$ (and therefore as $r \rightarrow \infty$), causing $c(\epsilon)$ to grow so rapidly that extremely huge values of n are needed before we have any significant improvement over (5). And it is unsatisfactory for theoretical purposes because it does not make use of the full power of the polynomial method on which it is based. We can obtain a better result if we let r vary with n , choosing larger and larger values of r as n increases. This idea is due to A. L. Toom [*Doklady Akad. Nauk SSSR* 150 (1963), 496–498, English translation in *Soviet Mathematics* 3 (1963), 714–716], who used it to show that computer circuitry for multiplication of n -bit numbers can be constructed involving a fairly small number of components as n grows. S. A. Cook [On the minimum computation time of functions (Thesis, Harvard

University, 1966), 51–77] later showed how Toom’s method can be adapted to fast computer programs.

Before we discuss the Toom–Cook algorithm any further, let us study a small example of the transition from $U(x)$ and $V(x)$ to the coefficients of $W(x)$. This example will not demonstrate the efficiency of the method, since the numbers are too small, but it points out some useful simplifications that we can make in the general case. Suppose that we want to multiply $u = 1234$ times $v = 2341$; in binary notation this is $u = (0100\ 1101\ 0010)_2$ times $v = (1001\ 0010\ 0101)_2$. Let $r = 2$; the polynomials $U(x)$, $V(x)$ in (8) are

$$U(x) = 4x^2 + 13x + 2, \quad V(x) = 9x^2 + 2x + 5.$$

Hence we find, for $W(x) = U(x)V(x)$,

$$\begin{aligned} U(0) &= 2, & U(1) &= 19, & U(2) &= 44, & U(3) &= 77, & U(4) &= 118; \\ V(0) &= 5, & V(1) &= 16, & V(2) &= 45, & V(3) &= 92, & V(4) &= 157; \\ W(0) &= 10, & W(1) &= 304, & W(2) &= 1980, & W(3) &= 7084, & W(4) &= 18526. \end{aligned} \tag{12}$$

Our job now is to compute the five coefficients of $W(x)$ from the latter five values.

There is an attractive little algorithm that can be used to compute the coefficients of a polynomial $W(x) = W_{m-1}x^{m-1} + \cdots + W_1x + W_0$ when the values $W(0), W(1), \dots, W(m-1)$ are given: Let us first write

$$W(x) = \theta_{m-1}x^{m-1} + \theta_{m-2}x^{m-2} + \cdots + \theta_1x^1 + \theta_0, \tag{13}$$

where $x^k = x(x-1)\dots(x-k+1)$, and where the coefficients θ_j are unknown. The falling factorial powers have the important property that

$$W(x+1) - W(x) = (m-1)\theta_{m-1}x^{m-2} + (m-2)\theta_{m-2}x^{m-3} + \cdots + \theta_1;$$

hence by induction we find that, for all $k \geq 0$,

$$\begin{aligned} \frac{1}{k!} \left(W(x+k) - \binom{k}{1} W(x+k-1) + \binom{k}{2} W(x+k-2) - \cdots + (-1)^k W(x) \right) \\ = \binom{m-1}{k} \theta_{m-1} x^{m-1-k} + \binom{m-2}{k} \theta_{m-2} x^{m-2-k} + \cdots + \binom{k}{k} \theta_k. \end{aligned} \tag{14}$$

Denoting the left-hand side of (14) by $(1/k!) \Delta^k W(x)$, we see that

$$\frac{1}{k!} \Delta^k W(x) = \frac{1}{k} \left(\frac{1}{(k-1)!} \Delta^{k-1} W(x+1) - \frac{1}{(k-1)!} \Delta^{k-1} W(x) \right)$$

and $(1/k!) \Delta^k W(0) = \theta_k$. So the coefficients θ_j can be evaluated using a very simple method, illustrated here for the polynomial $W(x)$ in (12):

$$\begin{array}{rccccc} 10 & & 294 & & & & \\ 304 & 1676 & 1382/2 = 691 & 1023/3 = 341 & & & \\ 1980 & 5104 & 3428/2 = 1714 & 1455/3 = 485 & 144/4 = 36 & & \\ 7084 & 11442 & 6338/2 = 3169 & & & & \\ 18526 & & & & & & \end{array} \tag{15}$$

The leftmost column of this tableau is a listing of the given values of $W(0), W(1), \dots, W(4)$; the k th succeeding column is obtained by computing the difference between successive values of the preceding column and dividing by k . The coefficients θ_j appear at the top of the columns, so that $\theta_0 = 10, \theta_1 = 294, \dots, \theta_4 = 36$, and we have

$$\begin{aligned} W(x) &= 36x^4 + 341x^3 + 691x^2 + 294x^1 + 10 \\ &= ((36(x-3) + 341)(x-2) + 691)(x-1) + 294)x + 10. \end{aligned} \quad (16)$$

In general, we can write

$$W(x) = (\dots((\theta_{m-1}(x-m+2) + \theta_{m-2})(x-m+3) + \theta_{m-3})(x-m+4) + \dots + \theta_1)x + \theta_0,$$

and this formula shows how the coefficients W_{m-1}, \dots, W_1, W_0 can be obtained from the θ 's:

36				341		
				-3 · 36		
36	233		691			
	-2 · 36		-2 · 233			
36	161	225		294		
	-1 · 36	-1 · 161	-1 · 225			
36	125	64	69	10		

Here the numbers below the horizontal lines successively show the coefficients of the polynomials

$$\begin{aligned} \theta_{m-1}, \quad & \theta_{m-1}(x+m+2) + \theta_{m-2}, \\ & (\theta_{m-1}(x-m+2) + \theta_{m-2})(x-m+3) + \theta_{m-3}, \quad \text{etc.} \end{aligned}$$

From this tableau we have

$$W(x) = 36x^4 + 125x^3 + 64x^2 + 69x + 10,$$

so the answer to our original problem is $1234 \cdot 2341 = W(16)$, where $W(16)$ is obtained by adding and shifting. A generalization of this method for obtaining coefficients is discussed in Section 4.6.4.

The basic Stirling number identity,

$$x^n = \left\{ \begin{matrix} n \\ n \end{matrix} \right\} x^n + \dots + \left\{ \begin{matrix} n \\ 1 \end{matrix} \right\} x^1 + \left\{ \begin{matrix} n \\ 0 \end{matrix} \right\},$$

Eq. 1.2.6-41, shows that if the coefficients of $W(x)$ are nonnegative, so are the numbers θ_j , and in such a case all of the intermediate results in the above computation are nonnegative. This further simplifies the Toom-Cook multiplication algorithm, which we will now consider in detail.

Algorithm C (*High-precision multiplication of binary numbers*). Given a positive integer n and two nonnegative n -bit integers u and v , this algorithm forms their $2n$ -bit product, w . Four auxiliary stacks are used to hold the long numbers that are manipulated during the procedure:

Stacks U, V : Temporary storage of $U(j)$ and $V(j)$ in step C4.
 Stack C : Numbers to be multiplied, and control codes.
 Stack W : Storage of $W(j)$.

These stacks may contain either binary numbers or special control symbols called code-1, code-2, and code-3. The algorithm also constructs an auxiliary table of numbers q_k, r_k ; this table is maintained in such a manner that it may be stored as a linear list, where a single pointer that traverses the list (moving back and forth) may be used to access the current table entry of interest.

(Stacks C and W are used to control the recursive mechanism of this multiplication algorithm in a reasonably straightforward manner that is a special case of general procedures discussed in Chapter 8.)

C1. [Compute q, r tables.] Set stacks U, V, C , and W empty. Set

$$k \leftarrow 1, \quad q_0 \leftarrow q_1 \leftarrow 16, \quad r_0 \leftarrow r_1 \leftarrow 4, \quad Q \leftarrow 4, \quad R \leftarrow 2.$$

Now if $q_{k-1} + q_k < n$, set

$$k \leftarrow k + 1, \quad Q \leftarrow Q + R, \quad R \leftarrow \lfloor \sqrt{Q} \rfloor, \quad q_k \leftarrow 2^Q, \quad r_k \leftarrow 2^R,$$

and repeat this operation until $q_{k-1} + q_k \geq n$. (Note: The calculation of $R \leftarrow \lfloor \sqrt{Q} \rfloor$ does not require a square root to be taken, since we may simply set $R \leftarrow R + 1$ if $(R + 1)^2 \leq Q$ and leave R unchanged if $(R + 1)^2 > Q$; see exercise 2. In this step we build the sequences

$k =$	0	1	2	3	4	5	6	...
$q_k =$	2^4	2^4	2^6	2^8	2^{10}	2^{13}	2^{16}	...
$r_k =$	2^2	2^2	2^2	2^2	2^3	2^3	2^4	...

The multiplication of 70000-bit numbers would cause this step to terminate with $k = 6$, since $70000 < 2^{13} + 2^{16}$.)

C2. [Put u, v on stack.] Put code-1 on stack C , then place u and v onto stack C as numbers of exactly $q_{k-1} + q_k$ bits each.

C3. [Check recursion level.] Decrease k by 1. If $k = 0$, the top of stack C now contains two 32-bit numbers, u and v ; remove them, set $w \leftarrow uv$ using a built-in routine for multiplying 32-bit numbers, and go to step C10. If $k > 0$, set $r \leftarrow r_k$, $q \leftarrow q_k$, $p \leftarrow q_{k-1} + q_k$, and go on to step C4.

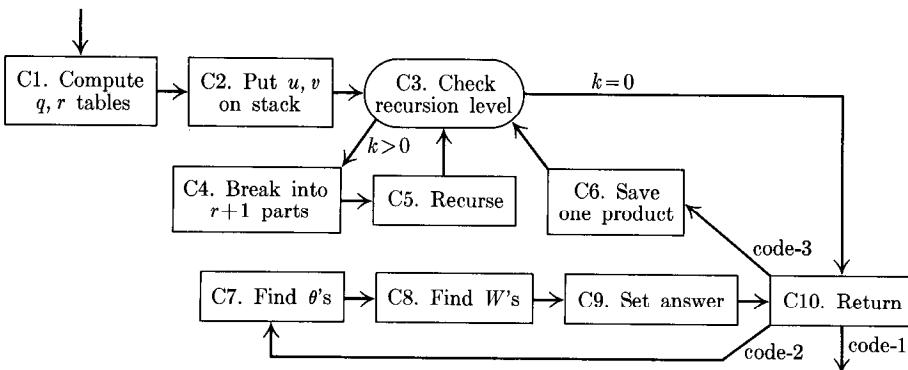


Fig. 8. Toom-Cook algorithm for high-precision multiplication.

- C4. [Break into $r + 1$ parts.] Let the number at the top of stack C be regarded as a list of $r + 1$ numbers with q bits each, $(U_r \dots U_1 U_0)_{2^q}$. (The top of stack C now contains an $(r + 1)q = (q_k + q_{k+1})$ -bit number.) For $j = 0, 1, \dots, 2r$, compute the p -bit numbers

$$(\dots(U_r j + U_{r-1})j + \dots + U_1)j + U_0 = U(j)$$

and successively put these values onto stack U . (The bottom of stack U now contains $U(0)$, then comes $U(1)$, etc., with $U(2r)$ on top. Note that

$$U(j) \leq U(2r) < 2^q((2r)^r + (2r)^{r-1} + \dots + 1) < 2^{q+1}(2r)^r \leq 2^p,$$

by exercise 3.) Then remove $U_r \dots U_1 U_0$ from stack C .

Now the top of stack C contains another list of $r + 1$ q -bit numbers, $V_r \dots V_1 V_0$, and the p -bit numbers

$$(\dots(V_r j + V_{r-1})j + \dots + V_1)j + V_0 = V(j)$$

should be put onto stack V in the same way. After this has been done, remove $V_r \dots V_1 V_0$ from stack C .

- C5. [Recurse.] Successively put the following items onto stack C , at the same time emptying stacks U and V :

$$\begin{aligned} \text{code-2, } &V(2r), U(2r), \text{code-3, } V(2r-1), U(2r-1), \dots, \\ &\text{code-3, } V(1), U(1), \text{code-3, } V(0), U(0). \end{aligned}$$

Go back to step C3.

- C6. [Save one product.] (At this point the multiplication algorithm has set w to one of the products $W(j) = U(j)V(j)$.) Put w onto stack W . (This number w contains $2(q_k + q_{k-1})$ bits.) Go back to step C3.

C7. [Find θ 's.] Set $r \leftarrow r_k$, $q \leftarrow q_k$, $p \leftarrow q_{k-1} + q_k$. (At this point stack W contains a sequence of numbers ending with $W(0)$, $W(1)$, \dots , $W(2r)$ from bottom to top, where each $W(j)$ is a $2p$ -bit number.)

Now for $j = 1, 2, 3, \dots, 2r$, perform the following loop: For $t = 2r$, $2r - 1, 2r - 2, \dots, j$, set $W(t) \leftarrow (W(t) - W(t-1))/j$. (Here j must increase and t must decrease. The quantity $(W(t) - W(t-1))/j$ will always be a nonnegative integer that fits in $2p$ bits; cf. (15).)

C8. [Find W 's.] For $j = 2r - 1, 2r - 2, \dots, 1$, perform the following loop: For $t = j, j+1, \dots, 2r - 1$, set $W(t) \leftarrow W(t) - jW(t+1)$. (Here j must decrease and t must increase. The result of this operation will again be a nonnegative $2p$ -bit integer; cf. (17).)

C9. [Set answer.] Set w to the $2(q_k + q_{k+1})$ -bit integer

$$(\dots (W(2r)2^q + W(2r-1))2^q + \dots + W(1))2^q + W(0).$$

Remove $W(2r), \dots, W(0)$ from stack W .

C10. [Return.] Set $k \leftarrow k + 1$. Remove the top of stack C . If it is code-3, go to step C6. If it is code-2, put w onto stack W and go to step C7. And if it is code-1, terminate the algorithm (w is the answer). ■

Let us now estimate the running time, $T(n)$, for Algorithm C, in terms of some things we shall call “cycles,” i.e., elementary machine operations. Step C1 takes $O(q_k)$ cycles, even if we represent the number q_k internally as a long string of q_k bits followed by some delimiter, since $q_k + q_{k-1} + \dots + q_0$ will be $O(q_k)$. Step C2 obviously takes $O(q_k)$ cycles.

Now let t_k denote the amount of computation required to get from step C3 to step C10 for a particular value of k (after k has been decreased at the beginning of step C3). Step C3 requires $O(q)$ cycles at most. Step C4 involves r multiplications of p -bit numbers by $(\lg 2r)$ -bit numbers, and r additions of p -bit numbers, all repeated $4r + 2$ times. Thus we need a total of $O(r^2 q \log r)$ cycles. Step C5 requires moving $4r + 2$ p -bit numbers, so it involves $O(rq)$ cycles. Step C6 requires $O(q)$ cycles, and it is done $2r + 1$ times per iteration. The recursion involved when the algorithm essentially invokes itself (by returning to step C3) requires t_{k-1} cycles, $2r + 1$ times. Step C7 requires $O(r^2)$ subtractions of p -bit numbers and divisions of $2p$ -bit by $(\lg 2r)$ -bit numbers, so it requires $O(r^2 q \log r)$ cycles. Similarly, step C8 requires $O(r^2 q \log r)$ cycles. Step C9 involves $O(rq)$ cycles, and C10 takes hardly any time at all.

Summing up, we have $T(n) = O(q_k) + O(q_k) + t_{k-1}$, where (if $q = q_k$ and $r = r_k$) the main contribution to the running time satisfies

$$\begin{aligned} t_k &= O(q) + O(r^2 q \log r) + O(rq) + (2r+1)O(q) + O(r^2 q \log r) \\ &\quad + O(r^2 q \log r) + O(rq) + O(q) + (2r+1)t_{k-1} \\ &= O(r^2 q \log r) + (2r+1)t_{k-1}. \end{aligned}$$

Thus there is a constant c such that

$$t_k \leq cr_k^2 q_k \lg r_k + (2r_k + 1)t_{k-1}.$$

To complete the estimation of t_k we can prove by brute force that

$$t_k \leq Cq_{k+1}2^{2.5\sqrt{\lg q_{k+1}}} \quad (18)$$

for some constant C . Let us choose $C > 20c$, and let us also take C large enough so that (18) is valid for $k \leq k_0$, where k_0 will be specified below. Then when $k > k_0$, let $Q_k = \lg q_k$, $R_k = \lg r_k$; we have by induction

$$t_k \leq cq_k r_k^2 \lg r_k + (2r_k + 1)Cq_k 2^{2.5\sqrt{Q_k}} = Cq_{k+1}2^{2.5\sqrt{\lg q_{k+1}}}(\eta_1 + \eta_2),$$

where

$$\eta_1 = \frac{c}{C}R_k 2^{R_k - 2.5\sqrt{Q_{k+1}}} < \frac{1}{20}R_k 2^{-R_k} < 0.05,$$

$$\eta_2 = \left(2 + \frac{1}{r_k}\right)2^{2.5(\sqrt{Q_k} - \sqrt{Q_{k+1}})} \rightarrow 2^{-1/4} < 0.85,$$

since

$$\sqrt{Q_{k+1}} - \sqrt{Q_k} = \sqrt{Q_k + \lfloor \sqrt{Q_k} \rfloor} - \sqrt{Q_k} \rightarrow \frac{1}{2}$$

as $k \rightarrow \infty$. It follows that we can find k_0 such that $\eta_2 < 0.95$ for all $k > k_0$, and this completes the proof of (18) by induction.

Finally, therefore, we may compute $T(n)$. Since $n > q_{k-1} + q_{k-2}$, we have $q_{k-1} < n$; hence

$$r_{k-1} = 2^{\lfloor \sqrt{\lg q_{k-1}} \rfloor} < 2^{\sqrt{\lg n}}, \quad \text{and} \quad q_k = r_{k-1}q_{k-1} < n2^{\sqrt{\lg n}}.$$

Thus

$$t_{k-1} \leq Cq_k 2^{2.5\sqrt{\lg q_k}} < Cn2^{\sqrt{\lg n} + 2.5(\sqrt{\lg n} + 1)},$$

and, since $T(n) = O(q_k) + t_{k-1}$, we have finally derived the following theorem:

Theorem C. *There is a constant c_0 such that the execution time of Algorithm C is less than $c_0 n 2^{3.5\sqrt{\lg n}}$ cycles. ■*

Since $n 2^{3.5\sqrt{\lg n}} = n^{1+3.5/\sqrt{\lg n}}$, this result is noticeably stronger than Theorem A. By adding a few complications to the algorithm, pushing the ideas to their apparent limits (see exercise 5), we can improve the estimated execution time to

$$T(n) = O(n 2^{\sqrt{2 \lg n}} \log n). \quad (19)$$

B. A modular method. There is another way to multiply large numbers very rapidly, based on the ideas of modular arithmetic as presented in Section 4.3.2. It is very hard to believe at first that this method can be of advantage, since a multiplication algorithm based on modular arithmetic must include the choice of moduli and the conversion of numbers into and out of modular representation, besides the actual multiplication operation itself. In spite of these formidable difficulties, A. Schönhage discovered that all of these operations can be carried out quite rapidly.

In order to understand the essential mechanism of Schönhage's method, we shall look at a special case. Consider the sequence defined by the rules

$$q_0 = 1, \quad q_{k+1} = 3q_k - 1, \quad (20)$$

so that $q_k = 3^k - 3^{k-1} - \dots - 1 = \frac{1}{2}(3^k + 1)$. We will study a procedure that multiplies $(18q_k + 8)$ -bit numbers, in terms of a method for multiplying $(18q_{k-1} + 8)$ -bit numbers. Thus, if we know how to multiply numbers having $(18q_0 + 8) = 26$ bits, the procedure to be described will show us how to multiply numbers of $(18q_1 + 8) = 44$ bits, then 98 bits, then 260 bits, etc., eventually increasing the number of bits by almost a factor of 3 at each step.

Let $p_k = 18q_k + 8$. When multiplying p_k -bit numbers, the idea is to use the six moduli

$$\begin{aligned} m_1 &= 2^{6q_k-1} - 1, & m_2 &= 2^{6q_k+1} - 1, & m_3 &= 2^{6q_k+2} - 1, \\ m_4 &= 2^{6q_k+3} - 1, & m_5 &= 2^{6q_k+5} - 1, & m_6 &= 2^{6q_k+7} - 1. \end{aligned} \quad (21)$$

These moduli are relatively prime, by Eq. 4.3.2-18, since the exponents

$$6q_k - 1, \quad 6q_k + 1, \quad 6q_k + 2, \quad 6q_k + 3, \quad 6q_k + 5, \quad 6q_k + 7 \quad (22)$$

are always relatively prime (see exercise 6). The six moduli in (21) are capable of representing numbers up to $m = m_1 m_2 m_3 m_4 m_5 m_6 > 2^{36q_k+16} = 2^{2p_k}$, so there is no chance of overflow in the multiplication of p_k -bit numbers u and v . Thus we may use the following method, when $k > 0$:

- a) Compute $u_1 = u \bmod m_1, \dots, u_6 = u \bmod m_6$; and $v_1 = v \bmod m_1, \dots, v_6 = v \bmod m_6$.
- b) Multiply u_1 by v_1 , u_2 by v_2, \dots, u_6 by v_6 . These are numbers of at most $6q_k + 7 = 18q_{k-1} + 1 < p_{k-1}$ bits, so the multiplications can be performed by using the assumed p_{k-1} -bit multiplication procedure.
- c) Compute $w_1 = u_1 v_1 \bmod m_1, w_2 = u_2 v_2 \bmod m_2, \dots, w_6 = u_6 v_6 \bmod m_6$.
- d) Compute w such that $0 \leq w < m$, $w \bmod m_1 = w_1, \dots, w \bmod m_6 = w_6$.

Let t_k be the amount of time needed for this process. It is not hard to see that operation (a) takes $O(p_k)$ cycles, since the determination of $u \bmod (2^\epsilon - 1)$ is quite simple (like "casting out nines"), as shown in Section 4.3.2. Similarly, operation (c) takes $O(p_k)$ cycles. Operation (b) requires essentially $6t_{k-1}$ cycles.

This leaves us with operation (d), which seems to be quite a difficult computation; but Schönhage has found an ingenious way to perform step (d) in $O(p_k \log p_k)$ cycles, and this is the crux of the method. As a consequence, we have

$$t_k = 6t_{k-1} + O(p_k \log p_k).$$

Since $p_k = 3^{k+2} + 17$, we can show that the time for n -bit multiplication is

$$T(n) = O(N^{\log_3 6}) = O(N^{1.63}). \quad (23)$$

(See exercise 7.)

Although the modular method is more complicated than the $O(n^{\lg 3})$ procedure discussed at the beginning of this section, Eq. (23) shows that it does, in fact, lead to an execution time substantially better than $O(n^2)$ for the multiplication of n -bit numbers. Thus we can improve on the classical method by using either of two completely different approaches.

Let us now analyze operation (d) above. Assume that we are given a set of positive integers $e_1 < e_2 < \dots < e_r$, relatively prime in pairs; let

$$m_1 = 2^{e_1} - 1, \quad m_2 = 2^{e_2} - 1, \quad \dots, \quad m_r = 2^{e_r} - 1. \quad (24)$$

We are also given numbers w_1, \dots, w_r such that $0 \leq w_j \leq m_j$. Our job is to determine the binary representation of the number w that satisfies the conditions

$$\begin{aligned} 0 \leq w &< m_1 m_2 \dots m_r, \\ w \equiv w_1 \pmod{m_1}, \quad \dots, \quad w \equiv w_r \pmod{m_r}. \end{aligned} \quad (25)$$

The method is based on (23) and (24) of Section 4.3.2. First we compute

$$w'_j = (\dots((w_j - w'_1)c_{1j} - w'_2)c_{2j} - \dots - w'_{j-1})c_{(j-1)j} \pmod{m_j}, \quad (26)$$

for $j = 2, \dots, r$, where $w'_1 = w_1 \pmod{m_1}$; then we compute

$$w = (\dots(w'_r m_{r-1} + w'_{r-1})m_{r-2} + \dots + w'_2)m_1 + w'_1. \quad (27)$$

Here c_{ij} is a number such that $c_{ij}m_i \equiv 1 \pmod{m_j}$; these numbers c_{ij} are not given, they must be determined from the e_j 's.

The calculation of (26) for all j involves $\binom{r}{2}$ additions modulo m_j , each of which takes $O(e_r)$ cycles, plus $\binom{r}{2}$ multiplications by c_{ij} , modulo m_j . The calculation of w by formula (27) involves r additions and r multiplications by m_j ; it is easy to multiply by m_j , since this is just adding, shifting, and subtracting, so it is clear that the evaluation of Eq. (27) takes $O(r^2 e_r)$ cycles. We will soon see that each of the multiplications by c_{ij} , modulo m_j , requires only $O(e_r \log e_r)$ cycles, and therefore it is possible to complete the entire job of conversion in $O(r^2 e_r \log e_r)$ cycles.

The above observations leave us with the following problem to solve: Given positive integers $e < f$ and a nonnegative integer $u < 2^f$, compute the value of $(cu) \bmod (2^f - 1)$, where c is the number such that $(2^e - 1)c \equiv 1 \pmod{2^f - 1}$; and the computation must be done in $O(f \log f)$ cycles. The result of exercise 4.3.2-6 gives a formula for c that suggests a suitable procedure. First we find the least positive integer b such that

$$be \equiv 1 \pmod{f}. \quad (28)$$

This can be done using Euclid's algorithm in $O((\log f)^3)$ cycles, since Euclid's algorithm applied to e and f requires $O(\log f)$ iterations, and each iteration requires $O((\log f)^2)$ cycles; alternatively, we could be very sloppy here without violating the total time constraint, by simply trying $b = 1, 2, \dots$, until (28) is satisfied, since such a process would take $O(f \log f)$ cycles in all. Once b has been found, exercise 4.3.2-6 tells us that

$$c = c[b] = \left(\sum_{0 \leq j < b} 2^{je} \right) \bmod (2^f - 1). \quad (29)$$

A brute-force multiplication of $(cu) \bmod (2^f - 1)$ would not be good enough to solve the problem, since we do not know how to multiply general f -bit numbers in $O(f \log f)$ cycles. But the special form of c provides a clue: The binary representation of c is composed of bits in a regular pattern, and Eq. (29) shows that the number $c[2b]$ can be obtained in a simple way from $c[b]$. This suggests that we can rapidly multiply a number u by $c[b]$ if we build $c[b]u$ up in $\lg b$ steps in a suitably clever manner, such as the following: Let the binary notation for b be

$$b = (b_s \dots b_2 b_1 b_0)_2;$$

we may calculate the sequences a_k, d_k, u_k, v_k defined by the rules

$$\begin{aligned} a_0 &= e, & a_k &= 2a_{k-1} \bmod f; \\ d_0 &= b_0 e, & d_k &= (d_{k-1} + b_k a_k) \bmod f; \\ u_0 &= u, & u_k &= (u_{k-1} + 2^{a_{k-1}} u_{k-1}) \bmod (2^f - 1); \\ v_0 &= b_0 u, & v_k &= (v_{k-1} + b_k 2^{d_{k-1}} u_k) \bmod (2^f - 1). \end{aligned} \quad (30)$$

It is easy to prove by induction on k that

$$\begin{aligned} a_k &= (2^k e) \bmod f; & u_k &= (c[2^k]u) \bmod (2^f - 1); \\ d_k &= ((b_k \dots b_1 b_0)_2 e) \bmod f; & v_k &= (c[(b_k \dots b_1 b_0)_2]u) \bmod (2^f - 1). \end{aligned} \quad (31)$$

Hence the desired result, $(c[b]u) \bmod (2^f - 1)$, is v_s . The calculation of a_k, d_k, u_k, v_k from $a_{k-1}, d_{k-1}, u_{k-1}, v_{k-1}$ takes $O(\log f) + O(\log f) + O(f) + O(f) = O(f)$ cycles, and therefore the entire calculation can be done in $sO(f) = O(f \log f)$ cycles as desired.

The reader will find it instructive to study the ingenious method represented by (30) and (31) very carefully. Similar techniques are discussed in Section 4.6.3.

Schönhage's paper [*Computing* 1 (1966), 182–196] shows that these ideas can be extended to the multiplication of n -bit numbers using $r \approx 2^{\sqrt{2}\lg n}$ moduli, obtaining a method analogous to Algorithm C. We shall not dwell on the details here, since Algorithm C is always superior; in fact, an even better method is next on our agenda.

C. Use of discrete Fourier transforms. The critical problem in high-precision multiplication is the determination of "convolution products" such as

$$u_r v_0 + u_{r-1} v_1 + \cdots + u_0 v_r,$$

and there is an intimate relation between convolutions and an important mathematical concept called "Fourier transformation." If $\omega = \exp(2\pi i/K)$ is a K th root of unity, the one-dimensional Fourier transform of the sequence of complex numbers $(u_0, u_1, \dots, u_{K-1})$ is defined to be the sequence $(\hat{u}_0, \hat{u}_1, \dots, \hat{u}_{K-1})$, where

$$\hat{u}_s = \sum_{0 \leq t < K} \omega^{st} u_t, \quad 0 \leq s < K. \quad (32)$$

Letting $(\hat{v}_0, \hat{v}_1, \dots, \hat{v}_{K-1})$ be defined in the same way, as the Fourier transform of $(v_0, v_1, \dots, v_{K-1})$, it is not difficult to see that $(\hat{u}_0 \hat{v}_0, \hat{u}_1 \hat{v}_1, \dots, \hat{u}_{K-1} \hat{v}_{K-1})$ is the transform of $(w_0, w_1, \dots, w_{K-1})$, where

$$\begin{aligned} w_r &= u_r v_0 + u_{r-1} v_1 + \cdots + u_0 v_r + u_{K-1} v_{r+1} + \cdots + u_{r+1} v_{K-1} \\ &= \sum_{i+j \equiv r \pmod{K}} u_i v_j. \end{aligned}$$

When $K \geq 2n - 1$ and $u_n = u_{n+1} = \cdots = u_{K-1} = v_n = v_{n+1} = \cdots = v_{K-1} = 0$, the w 's are just what we need for multiplication, since the terms $u_{K-1} v_{r+1} + \cdots + u_{r+1} v_{K-1}$ vanish when $0 \leq r \leq 2n - 2$. In other words, the *transform of a convolution product is the ordinary product of the transforms*. This idea is actually a special case of Toom's use of polynomials (cf. (10)), with x replaced by roots of unity.

If K is a power of 2, the discrete Fourier transform (32) can be obtained quite rapidly when the computations are arranged in a certain way, and so can the inverse transform (determining the w 's from the \hat{w} 's). This property of Fourier transforms was exploited by V. Strassen in 1968, who discovered how to multiply large numbers faster than was possible under all previously known schemes. He and A. Schönhage later refined the method and published improved procedures in *Computing* 7 (1971), 281–292. In order to understand their approach to the problem, let us first take a look at the mechanism of fast Fourier transforms.

Given a sequence of $K = 2^k$ complex numbers (u_0, \dots, u_{K-1}) , and given the complex number

$$\omega = \exp(2\pi i/K),$$

the sequence $(\hat{u}_0, \dots, \hat{u}_{K-1})$ defined in (32) can be calculated rapidly by carrying out the following scheme. (In these formulas the parameters s_j and t_j are either 0 or 1, so that each "pass" represents 2^k computations.)

Pass 0. Let $A^{[0]}(t_{k-1}, \dots, t_0) = u_t$, where $t = (t_{k-1} \dots t_0)_2$.

Pass 1. Set $A^{[1]}(s_{k-1}, t_{k-2}, \dots, t_0) \leftarrow$

$$A^{[0]}(0, t_{k-2}, \dots, t_0) + \omega^{(s_{k-1}0\dots0)_2} \cdot A^{[0]}(1, t_{k-2}, \dots, t_0).$$

Pass 2. Set $A^{[2]}(s_{k-1}, s_{k-2}, t_{k-3}, \dots, t_0) \leftarrow$

$$A^{[1]}(s_{k-1}, 0, t_{k-3}, \dots, t_0) + \omega^{(s_{k-2}s_{k-1}0\dots0)_2} \cdot A^{[1]}(s_{k-1}, 1, t_{k-3}, \dots, t_0).$$

...

Pass k . Set $A^{[k]}(s_{k-1}, \dots, s_1, s_0) \leftarrow$

$$A^{[k-1]}(s_{k-1}, \dots, s_1, 0) + \omega^{(s_0s_1\dots s_{k-1})_2} \cdot A^{[k-1]}(s_{k-1}, \dots, s_1, 1).$$

It is fairly easy to prove by induction that we have

$$\begin{aligned} A^{[j]}(s_{k-1}, \dots, s_{k-j}, t_{k-j-1}, \dots, t_0) \\ = \sum_{0 \leq t_{k-1}, \dots, t_{k-j} \leq 1} \omega^{(s_0s_1\dots s_{k-1})_2 \cdot (t_{k-1}\dots t_{k-j}0\dots0)_2} u_t, \end{aligned} \quad (33)$$

so that

$$A^{[k]}(s_{k-1}, \dots, s_1, s_0) = \hat{u}_s, \quad \text{where } s = (s_0s_1\dots s_{k-1})_2. \quad (34)$$

(Note the reversed order of the binary digits in s . For further discussion of transforms such as this, see Section 4.6.4.)

To get the inverse Fourier transform (u_0, \dots, u_{K-1}) from the values of $(\hat{u}_0, \dots, \hat{u}_{K-1})$, we may note that the “double transform” is

$$\begin{aligned} \hat{u}_r &= \sum_{0 \leq s < K} \omega^{rs} \hat{u}_s = \sum_{0 \leq s, t < K} \omega^{rs} \omega^{st} u_t \\ &= \sum_{0 \leq t < K} u_t \left(\sum_{0 \leq s < K} \omega^{s(t+r)} \right) = Ku_{(-r) \bmod K}, \end{aligned}$$

since the geometric series $\sum_{0 \leq s < K} \omega^{sj}$ sums to zero unless j is a multiple of K . Therefore the inverse transform can be computed in the same way as the transform itself, except that the final results must be divided by K and shuffled slightly.

Applying this to the problem of integer multiplication, suppose we wish to compute the product of two n -bit integers u and v . As in Algorithm C we shall work with groups of bits; let

$$2n \leq 2^k l < 4n, \quad K = 2^k, \quad L = 2^l, \quad (35)$$

and write

$$u = (U_{K-1} \dots U_1 U_0)_L, \quad v = (V_{K-1} \dots V_1 V_0)_L, \quad (36)$$

regarding u and v as K -place numbers in radix L so that each digit U_j or V_j is an l -bit integer. Actually the leading digits U_j and V_j are zero for all $j \geq K/2$,

because $2^{k-1}l \geq n$. We will select appropriate values for k and l later; at the moment our goal is to see what happens in general, so that we can choose k and l intelligently when all the facts are before us.

The next step of the multiplication procedure is to compute the Fourier transforms $(\hat{u}_0, \dots, \hat{u}_{K-1})$ and $(\hat{v}_0, \dots, \hat{v}_{K-1})$ of the sequences (u_0, \dots, u_{K-1}) and (v_0, \dots, v_{K-1}) , where we define

$$u_t = U_t/2^{k+l}, \quad v_t = V_t/2^{k+l}. \quad (37)$$

This scaling is done for convenience so that the absolute values $|u_t|$ and $|v_t|$ are less than 2^{-k} , ensuring that $|\hat{u}_s|$ and $|\hat{v}_s|$ will be less than 1 for all s .

An obvious problem arises here, since the complex number ω can't be represented exactly in binary notation. How are we going to compute a reliable Fourier transform? By a stroke of good luck, it turns out that everything will work properly if we do the calculations with only a modest amount of precision. For the moment let us bypass this question and assume that infinite-precision calculations are being performed; we shall analyze later how much accuracy is actually needed.

Once the \hat{u}_s and \hat{v}_s have been found, we let $\hat{w}_s = \hat{u}_s \hat{v}_s$ for $0 \leq s < K$ and determine the inverse Fourier transform (w_0, \dots, w_{K-1}) . As explained above, we now have

$$w_r = \sum_{i+j=r} u_i v_j = \sum_{i+j=r} U_i V_j / 2^{2k+2l},$$

so the integers $W_r = 2^{2k+2l} w_r$ are the coefficients in the desired product

$$u \cdot v = W_{K-2} L^{K-2} + \dots + W_1 L + W_0. \quad (38)$$

Since $0 \leq W_r < (r+1)L^2 < KL^2$, each W_r has at most $k+2l$ bits, so it will not be difficult to compute the binary representation when the W 's are known unless k is large compared to l .

Let us try to estimate how much time this method takes, if m -bit fixed point arithmetic is used in calculating the Fourier transforms. Exercise 10 shows that all of the quantities $A^{[j]}$ during all the passes of the transform calculations will be less than 1 in magnitude because of the scaling (37), hence it suffices to deal with m -bit fractions $(.a_{-1} \dots a_{-m})_2$ for the real and imaginary parts of all the intermediate quantities. Simplifications are possible because the inputs u_t and v_t are real-valued; only K real values instead of $2K$ need to be carried in each step (see exercise 4.6.4-14). We will ignore such refinements in order to keep complications to a minimum.

The first job is to compute ω and its powers. For simplicity we shall make a table of the values $\omega^0, \dots, \omega^{K-1}$. Let

$$\omega_r = \exp(2\pi i/2^r),$$

so that $\omega_1 = -1$, $\omega_2 = i$, $\omega_3 = (1+i)/\sqrt{2}$, \dots , $\omega_k = \omega$. If $\omega_r = x_r + iy_r$, we have

$$\omega_{r+1} = \sqrt{\frac{1+x_r}{2}} + i\sqrt{\frac{1-y_r}{2}}. \quad (39)$$

The calculation of $\omega_1, \omega_2, \dots, \omega_k$ takes negligible time compared with the other computations we need, so we can use any straightforward algorithm for square roots. Once the ω_r have been calculated we can compute all of the powers ω^j by starting with $\omega^0 = 1$ and using the following idea for $j > 0$: If $j = 2^{K-r} \cdot q$ where q is odd, and if $j_0 = 2^{K-r} \cdot (q - 1)$, we have

$$\omega^j = \omega^{j_0} \cdot \omega_r. \quad (40)$$

This method of calculation keeps errors from propagating, since each ω^j is a product of at most k of the ω_r 's. The total time to calculate all the ω^j is $O(KM)$, where M is the time to do an m -bit complex multiplication; this is less time than the subsequent steps will require, so we can ignore it.

Each of the three Fourier transformations comprises k passes, each of which involves K operations of the form $a \leftarrow b + \omega^j c$, so the total time to calculate the Fourier transforms is

$$O(kKM) = O(Mnk/l).$$

Finally, the work involved in computing the binary digits of $u \cdot v$ using (38) is $O(K(k+l)) = O(n+nk/l)$. Summing over all operations, we find that the total time to multiply n -bit numbers u and v will be $O(n) + O(Mnk/l)$.

Now let's see how large the intermediate precision m needs to be, so that we know how large M needs to be. For simplicity we shall be content with safe estimates of the accuracy, instead of finding the best possible bounds. It will be convenient to compute all the ω^j so that our approximations $(\omega^j)'$ will satisfy $|(\omega^j)'| \leq 1$; this condition is easy to guarantee if we truncate towards zero instead of rounding. The operations we need to perform with m -bit fixed point complex arithmetic are all obtained by replacing an exact computation of the form $a \leftarrow b + \omega^j c$ by the approximate computation

$$a' \leftarrow \text{truncate}(b' + (\omega^j)' c'), \quad (41)$$

where b' , $(\omega^j)'$, and c' are previously computed approximations; all of these complex numbers and their approximations are bounded by 1 in absolute value. If $|b' - b| \leq \delta_1$, $|(\omega^j)' - \omega^j| \leq \delta_2$, and $|c' - c| \leq \delta_3$, it is not difficult to see that we will have $|a' - a| < \delta + \delta_1 + \delta_2 + \delta_3$, where

$$\delta = |2^{-m} + 2^{-m} i| = 2^{1/2-m},$$

because we have $|(\omega^j)' c' - \omega^j c| = |((\omega^j)' - \omega^j)c' + \omega^j(c' - c)| \leq \delta_2 + \delta_3$, and δ is the maximum truncation error. The approximations $(\omega^j)'$ are obtained by starting with approximate values ω'_r to the numbers defined in (39), and we may assume that $|\omega'_r - \omega_r| \leq \delta$. Each multiplication (40) has the form of (41) with $b' = 0$, so an additional error of at most 2δ is made per multiplication, and we have $|(\omega^j)' - \omega^j| \leq (2k-1)\delta$ for all j .

If we have errors of at most ϵ before any pass of the fast Fourier transform, the operations of that pass therefore have the form (41) where $\delta_1 = \delta_3 = \epsilon$ and $\delta_2 = (2k - 1)\delta$, and the errors after the pass will be at most $2\epsilon + 2k\delta$. There is no error in “Pass 0,” so we find by induction on j that the maximum error after “Pass j ” is bounded by $(2^j - 1) \cdot 2k\delta$, and the computed values of \hat{u}_s will satisfy $|(\hat{u}_s)' - \hat{u}_s| < (2^k - 1) \cdot 2k\delta$. A similar formula will hold for $(\hat{v}_s)'$; and we will have

$$|(\hat{w}_s)' - \hat{w}_s| < 2(2^k - 1) \cdot 2k\delta + \delta.$$

During the inverse transformation there is an additional accumulation of errors, but the division by $K = 2^k$ ameliorates most of this; by the same argument we find that the computed values w_r' will satisfy

$$|w_r' - w_r| < 4k2^k\delta.$$

We need enough precision to make $2^{2k+2l}w_r'$ round to the correct integer W_r , hence we need

$$2^{2k+2l+2+\lg k+k+1/2-m} \leq \frac{1}{2},$$

i.e., $m \geq 3k + 2l + \lg k + 7/2$. This will hold if we simply require that

$$k \geq 7 \quad \text{and} \quad m \geq 4k + 2l. \quad (42)$$

Relations (35) and (42) can be used to determine parameters k, l, m so that multiplication takes $O(n) + O(Mnk/l)$ units of time, where M is the time to multiply m -bit fractions.

If we are using MIX, for example, suppose we want to multiply binary numbers having $n = 2^{13} = 8192$ bits each. We can choose $k = 11, l = 8, m = 60$, so that the necessary m -bit operations are nothing more than double precision arithmetic. The running time M needed to do fixed point m -bit complex multiplication will therefore be comparatively small. With triple-precision operations we can go up for example to $k = l = 15, n \leq 15 \cdot 2^{14}$, which takes us way beyond MIX’s memory capacity.

Further study of the choice of k, l , and m leads in fact to a rather surprising conclusion: *For all practical purposes we can assume that M is constant, and the Schönhage–Strassen multiplication technique will have a running time linearly proportional to n .* The reason is that we can choose $k = l$ and $m = 6k$; this choice of k is always less than $\lg n$, so we will never need to use more than sextuple precision unless n is larger than the word size of our computer. (In particular, n would have to be larger than the capacity of an index register, so we probably couldn’t fit the numbers u and v in main memory.)

The practical problem of fast multiplication is therefore solved, except for improvements in the constant factor. In fact, the all-integer convolution algorithm of exercise 4.6.4–59 is probably a better choice for practical high-precision multiplication, even though it has a slightly worse asymptotic behavior. Our interest in multiplying large numbers is partly theoretical, however, because it

is interesting to explore the ultimate limits of computational complexity. So let's forget practical considerations and suppose that n is extremely huge, perhaps much larger than the number of atoms in the universe. We can let m be approximately $6 \lg n$, and use the same algorithm recursively to do the m -bit multiplications. The running time will satisfy $T(n) = O(nT(\log n))$; hence

$$T(n) \leq Cn(C\lg n)(C\lg\lg n)(C\lg\lg\lg n)\dots,$$

where the product continues until reaching a factor with $\lg\dots\lg n \leq 1$.

Schönhage and Strassen showed how to improve this theoretical upper bound to $O(n \log n \log \log n)$ in their paper, by using integer numbers ω to carry out fast Fourier transforms on integers, modulo numbers of the form $2^e + 1$. This upper bound applies to Turing machines, i.e., to computers with bounded memory and a finite number of arbitrarily long tapes.

If we allow ourselves a more powerful computer, with random access to any number of words of bounded size, Schönhage has pointed out that the upper bound drops to $O(n \log n)$. For we can choose $k = l$ and $m = 6k$, and we have time to build a complete multiplication table of all possible products xy for $0 \leq x, y < 2^{\lceil m/12 \rceil}$. (The number of such products is 2^k or 2^{k+1} , and we can compute each table entry by addition from one of its predecessors in $O(k)$ steps, hence $O(k2^k) = O(n)$ steps will suffice for the calculation.) In this case M is the time needed to do 12-place arithmetic in radix $2^{\lceil m/12 \rceil}$, and it follows that $M = O(k) = O(\log n)$ because 1-place multiplication can be done by table lookup.

Schönhage discovered in 1979 that a *pointer machine* can carry out n -bit multiplication in $O(n)$ steps; see exercise 12. Such devices (which are also called “storage modification machines” and “linking automata”) seem to provide the best models of computation when $n \rightarrow \infty$, as discussed at the end of Section 2.6. So we can conclude that multiplication in $O(n)$ steps is possible for theoretical purposes as well as in practice.

D. Division. Now that we have efficient routines for multiplication, let's consider the inverse problem. It turns out that division can be performed just as fast as multiplication, except for a constant factor.

To divide an n -bit number u by an n -bit number v , we may first find an n -bit approximation to $1/v$, then multiply by u to get an approximation \hat{q} to u/v ; finally, we can make the slight correction necessary to \hat{q} to ensure that $0 \leq u - qv < v$ by using another multiplication. From this reasoning, we see that it suffices to have an efficient algorithm for approximating the reciprocal of an n -bit number. The following algorithm does this, using “Newton's method” as explained at the end of Section 4.3.1.

Algorithm R (High-precision reciprocal). Let v have the binary representation $v = (0.v_1v_2v_3\dots)_2$, where $v_1 = 1$. This algorithm computes an approximation z to $1/v$, such that

$$|z - 1/v| \leq 2^{-n}. \tag{43}$$

- R1.** [Initial approximation.] Set $z \leftarrow \frac{1}{4}[32/(4v_1 + 2v_2 + v_3)]$ and $k \leftarrow 0$.
- R2.** [Newtonian iteration.] (At this point we have a number z of the binary form $(xx.xx\dots x)_2$ with $2^k + 1$ places after the radix point, and $z \leq 2$.) Calculate $z^2 = (xxx.xx\dots x)_2$ exactly, using a high-speed multiplication routine. Then calculate $V_k z^2$ exactly, where $V_k = (0.v_1 v_2 \dots v_{2k+1} + 3)_2$. Then set $z \leftarrow 2z - V_k z^2 + r$, where $0 \leq r < 2^{-2^{k+1}-1}$ is added if necessary to "round up" z so that it is a multiple of $2^{-2^{k+1}-1}$. Finally, set $k \leftarrow k + 1$.
- R3.** [Test for end.] If $2^k < n$, go back to step R2; otherwise the algorithm terminates. ■

This algorithm is based on a suggestion by S. A. Cook. A similar technique has been used in computer hardware [see Anderson, Earle, Goldschmidt, and Powers, *IBM J. Res. Dev.* **11** (1967), 48–52]. Of course, it is necessary to check the accuracy of Algorithm R quite carefully, because it comes very close to being inaccurate. We will prove by induction that

$$z \leq 2 \quad \text{and} \quad |z - 1/v| \leq 2^{-2^k} \quad (44)$$

at the beginning and end of step R2.

For this purpose, let $\delta_k = 1/v - z_k$, where z_k is the value of z after k iterations of step R2. To start the induction on k , we have

$$\delta_0 = 1/v - 8/v' + (32/v' - [32/v'])/4 = \eta_1 + \eta_2,$$

where $v' = (v_1 v_2 v_3)_2$ and $\eta_1 = (v' - 8v)/vv'$, so that we have $-\frac{1}{2} < \eta_1 \leq 0$ and $0 \leq \eta_2 < \frac{1}{4}$. Hence $|\delta_0| < \frac{1}{2}$. Now suppose that (44) has been verified for k ; then

$$\begin{aligned} \delta_{k+1} &= 1/v - z_{k+1} = 1/v - z_k - z_k(1 - z_k V_k) - r \\ &= \delta_k - z_k(1 - z_k v) - z_k^2(v - V_k) - r \\ &= \delta_k - (1/v - \delta_k)v\delta_k - z_k^2(v - V_k) - r \\ &= v\delta_k^2 - z_k^2(v - V_k) - r. \end{aligned}$$

Now

$$0 \leq v\delta_k^2 < \delta_k^2 \leq (2^{-2^k})^2 = 2^{-2^{k+1}},$$

and

$$0 \leq z^2(v - V_k) + r < 4(2^{-2^{k+1}-3}) + 2^{-2^{k+1}-1} = 2^{-2^{k+1}},$$

so $|\delta_{k+1}| \leq 2^{-2^{k+1}}$. We must still verify the first inequality of (44); to show that $z_{k+1} \leq 2$, there are three cases: (a) $V_k = \frac{1}{2}$; then $z_{k+1} = 2$. (b) $V_k \neq \frac{1}{2} = V_{k-1}$; then $z_k = 2$, so $2z_k - z_k^2 V_k \leq 2 - 2^{-2^{k+1}-1}$. (c) $V_{k-1} \neq \frac{1}{2}$; then $z_{k+1} = 1/v - \delta_{k+1} < 2 - 2^{-2^{k+1}} \leq 2$, since $k > 0$.

The running time of Algorithm R is bounded by

$$2T(4n) + 2T(2n) + 2T(n) + 2T(\frac{1}{2}n) + \dots + O(n)$$

steps, where $T(n)$ is an upper bound on the time needed to do a multiplication of n -bit numbers. If $T(n)$ has the form $nf(n)$ for some monotonically nondecreasing function $f(n)$, we have

$$T(4n) + T(2n) + T(n) + \dots < T(8n),$$

so division can be done with a speed comparable to that of multiplication except for a constant factor.

R. P. Brent has shown that functions such as $\log x$, $\exp x$, and $\arctan x$ can be evaluated to n significant bits in $O(M(n) \log n)$ steps, if it takes $M(n)$ units of time to multiply n -bit numbers [JACM 23 (1976), 242–251].

E. An even faster multiplication method. It is natural to wonder if multiplication of n -bit numbers can be accomplished in just n steps. We have come from order n^2 down to order n , so perhaps we can squeeze the time down to the absolute minimum. In fact, it is actually possible to output the answer as fast as we input the digits, if we leave the domain of conventional computer programming and allow ourselves to build a computer that has an unlimited number of components all acting at once.

A *linear iterative array* of automata is a set of devices M_1, M_2, M_3, \dots that can each be in a finite set of “states” at each step of a computation. The machines M_2, M_3, \dots all have *identical* circuitry, and their state at time $t+1$ is a function of their own state at time t as well as the states of their left and right neighbors at time t . The first machine M_1 is slightly different: its state at time $t+1$ is a function of its own state and that of M_2 , at time t , and also of the *input* at time t . The *output* of a linear iterative array is a function defined on the states of M_1 .

Let $u = (u_{n-1} \dots u_1 u_0)_2$, $v = (v_{n-1} \dots v_1 v_0)_2$, and $q = (q_{n-1} \dots q_1 q_0)_2$ be binary numbers, and let $uv + q = w = (w_{2n-1} \dots w_1 w_0)_2$. It is a remarkable fact that a linear iterative array can be constructed, independent of n , that will output w_0, w_1, w_2, \dots at times 1, 2, 3, …, if it is given the inputs (u_0, v_0, q_0) , (u_1, v_1, q_1) , (u_2, v_2, q_2) , … at times 0, 1, 2, ….

We can state this phenomenon in the language of computer hardware, by saying that it is possible to design a single “integrated circuit module” with the following property: If we wire together sufficiently many of these devices in a straight line, with each module communicating only with its left and right neighbors, the resulting circuitry will produce the $2n$ -bit product of n -bit numbers in exactly $2n$ clock pulses.

Here is the basic idea behind this construction: At time 0, machine M_1 senses (u_0, v_0, q_0) and it therefore is able to output $(u_0 v_0 + q_0) \bmod 2$ at time 1. Then it sees (u_1, v_1, q_1) and it can output $(u_0 v_1 + u_1 v_0 + q_1 + k_1) \bmod 2$, where k_1 is the “carry” left over from the previous step, at time 2. Next it sees (u_2, v_2, q_2) and outputs $(u_0 v_2 + u_1 v_1 + u_2 v_0 + q_2 + k_2) \bmod 2$; furthermore, its state holds the values of u_2 and v_2 so that machine M_2 will be able to sense these values at time 3, and M_2 will be able to compute $u_2 v_2$ for the benefit of M_1 at time 4. Machine M_1 essentially arranges to start M_2 multiplying the sequence (u_2, v_2) ,

Table 1

MULTIPLICATION IN A LINEAR ITERATIVE ARRAY

Time	Input		Module M_1					Module M_2					Module M_3							
			u_j	v_j	c	x_0	x_1	x	z_2	c	x_0	x_1	x	z_2	c	x_0	x_1	x	z_2	
						y_0	y_1	y	z_1		y_0	y_1	y	z_1	z_0	y_0	y_1	y	z_1	z_0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
						0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
						1	0	0	1	0	0	0	0	0	0	0	0	0	0	
2	1	0	2	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	
						1	1	0	0	0	0	0	0	0	0	0	0	0	0	
3	0	1	3	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
						1	1	1	1	1	0	0	0	0	1	0	0	0	0	
4	1	0	3	1	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	
						1	1	0	0	0	1	0	0	0	1	0	0	0	0	
5	0	0	3	1	1	1	0	0	0	2	1	0	0	0	0	0	0	0	0	
						1	1	1	1	1	1	0	0	0	1	0	0	0	0	
6	0	0	3	1	1	0	0	1	0	3	1	0	1	1	0	0	0	0	0	
						1	1	0	0	0	1	0	1	1	0	0	0	0	0	
7	0	0	3	1	1	0	0	0	0	3	1	0	0	0	0	1	1	0	0	
						1	1	0	0	0	1	0	0	0	0	1	1	0	0	
8	0	0	3	1	1	0	0	0	0	3	1	0	0	0	0	2	1	0	0	
						1	1	0	0	0	1	0	0	0	0	2	1	0	0	
9	0	0	3	1	1	0	0	0	0	3	1	0	0	0	0	3	1	0	0	
						1	1	0	0	0	1	0	0	0	1	3	1	0	0	
10	0	0	3	1	1	0	0	0	0	3	1	0	0	0	0	3	1	0	0	
						1	1	0	0	0	1	0	0	0	0	3	1	0	0	
11	0	0	3	1	1	0	0	0	0	3	1	0	0	0	0	3	1	0	0	
						1	1	0	0	0	1	0	0	0	0	3	1	0	0	

$(u_3, v_3), \dots$, and M_2 will ultimately give M_3 the job of multiplying (u_4, v_4) , (u_5, v_5) , etc. Fortunately, things just work out so that no time is lost. The reader will find it interesting to deduce further details from the formal description that follows.

Each automaton has 2^{11} states

$$(c, x_0, y_0, x_1, y_1, x, y, z_2, z_1, z_0),$$

where $0 \leq c < 4$ and each of the x 's, y 's, and z 's is either 0 or 1. Initially, all devices are in state $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$. Suppose that a machine M_j , for $j > 1$, is in state $(c, x_0, y_0, x_1, y_1, x, y, z_2, z_1, z_0)$ at time t , and its left neighbor M_{j-1} is in state $(c^l, x_0^l, y_0^l, x_1^l, y_1^l, x^l, y^l, z_2^l, z_1^l, z_0^l)$ while its right neighbor M_{j+1} is in state $(c^r, x_0^r, y_0^r, x_1^r, y_1^r, x^r, y^r, z_2^r, z_1^r, z_0^r)$ at that time. Then machine M_j will go into state $(c', x'_0, y'_0, x'_1, y'_1, x', y', z'_2, z'_1, z'_0)$ at time $t + 1$, where

$$\begin{aligned} c' &= \min(c + 1, 3) && \text{if } c^l = 3, & 0 && \text{otherwise;} \\ (x'_0, y'_0) &= (x^l, y^l) && \text{if } c = 0, & (x_0, y_0) && \text{otherwise;} \\ (x'_1, y'_1) &= (x^l, y^l) && \text{if } c = 1, & (x_1, y_1) && \text{otherwise;} \\ (x', y') &= (x^l, y^l) && \text{if } c \geq 2, & (x, y) && \text{otherwise;} \end{aligned} \quad (45)$$

and $(z'_2 z'_1 z'_0)_2$ is the binary notation for

$$z_0^r + z_1 + z_2^l + \begin{cases} x^l y^l, & \text{if } c = 0; \\ x_0 y^l + x^l y_0, & \text{if } c = 1; \\ x_0 y^l + x_1 y_1 + x^l y_0, & \text{if } c = 2; \\ x_0 y^l + x_1 y + x y_1 + x^l y_0, & \text{if } c = 3. \end{cases} \quad (46)$$

The leftmost machine M_1 behaves in almost the same way as the others; it acts exactly as if there were a machine to its left in state $(3, 0, 0, 0, 0, u, v, q, 0, 0)$ when it is receiving the inputs (u, v, q) . The output of the array is the z_0 component of M_1 .

Table 1 shows an example of this array acting on the inputs

$$u = v = (\dots 00010111)_2, \quad q = (\dots 00001011)_2.$$

The output sequence appears in the lower right portion of the states of M_1 :

$$0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, \dots,$$

representing the number $(\dots 01000011100)_2$ from right to left.

This construction is based on a similar one first published by A. J. Atrubin, *IEEE Trans. EC-14* (1965), 394–399.

S. Winograd [JACM 14 (1967), 793–802] has investigated the minimum multiplication time achievable in a logical circuit when n is given and when the inputs are available all at once in coded form. See also C. S. Wallace, *IEEE Trans. EC-13* (1964), 14–17; A. C. Yao, to appear.

EXERCISES

1. [22] The idea expressed in (2) can be generalized to the decimal system, if the radix 2 is replaced by 10. Using this generalization, calculate 2718 times 4742 (reducing this product of four-digit numbers to three products of two-digit numbers, and reducing each of the latter to products of one-digit numbers).

2. [M22] Prove that, in step C1 of Algorithm C, the value of R either stays the same or increases by one when we set $R \leftarrow \lfloor \sqrt{Q} \rfloor$. (Therefore, as observed in that step, we need not calculate a square root.)

3. [M23] Prove that the sequences q_k, r_k defined in Algorithm C satisfy the inequality $2^{q_{k+1}}(2r_k)^{r_k} \leq 2^{q_{k-1}+q_k}$, when $k > 0$.

► 4. [28] (K. Baker.) Show that it is advantageous to evaluate the polynomial $W(x)$ at the points $x = -r, \dots, 0, \dots, r$ instead of at the nonnegative points $x = 0, 1, \dots, 2r$ as in Algorithm C. The polynomial $U(x)$ can be written

$$U(x) = U_e(x^2) + xU_o(x^2),$$

and similarly $V(x)$ and $W(x)$ can be expanded in this way; show how to exploit this idea, obtaining faster calculations in steps C7 and C8.

► 5. [35] Show that if in step C1 of Algorithm C we set $R \leftarrow \lceil \sqrt{2Q} \rceil + 1$ instead of $R \leftarrow \lfloor \sqrt{Q} \rfloor$, with suitable initial values of q_0, q_1, r_0 , and r_1 , then (19) can be improved to $t_k \leq q_{k+1} 2^{\sqrt{2 \lg q_{k+1}}} (\lg q_{k+1})$.

6. [M23] Prove that the six numbers in (22) are relatively prime in pairs.

7. [M23] Prove (23).

► 8. [25] Prove that it takes only $O(K \log K)$ arithmetic operations to evaluate the discrete Fourier transform (32), even when K is not a power of 2. [Hint: Rewrite (32) in the form

$$\hat{u}_s = \omega^{-s^2/2} \sum_{0 \leq t < K} \omega^{(s+t)^2/2} \omega^{-t^2/2} u_t$$

and express this sum as a convolution product.]

9. [M15] Suppose the Fourier transformation method of the text is applied with all occurrences of ω replaced by ω^q , where q is some fixed integer. Find a simple relation between the numbers $(\tilde{u}_0, \tilde{u}_1, \dots, \tilde{u}_{K-1})$ obtained by this general procedure and the numbers $(\hat{u}_0, \hat{u}_1, \dots, \hat{u}_{K-1})$ obtained when $q = 1$.

10. [M26] The scaling in (37) makes it clear that all the complex numbers $A^{[j]}$ computed by pass j of the transformation subroutine will be less than 2^{j-k} in absolute value, during the calculations of \hat{u}_s and \tilde{v}_s in the Schönhage–Strassen multiplication algorithm. Show that all of the $A^{[j]}$ will be less than 1 in absolute value during the third Fourier transformation (the calculation of w_r).

► 11. [M26] If n is fixed, how many of the automata in the linear iterative array (45), (46) are needed to compute the product of n -bit numbers? (Note that the automaton M_j is influenced only by the component z'_0 of the machine on its right, so we may remove all automata whose z_0 component is always zero whenever the inputs are n -bit numbers.)

► 12. [30] (A. Schönhage.) The purpose of this exercise is to prove that a simple form of pointer machine can multiply n -bit numbers in $O(n)$ steps. The machine has no built-in facilities for arithmetic; all it does is work with nodes and pointers. Each node has the same finite number of link fields, and there are finitely many link registers. The only operations this machine can do are:

- i) read one bit of input and jump if that bit is 0;
- ii) output 0 or 1;
- iii) load a register with the contents of another register or with the contents of a link field in a node pointed to by a register;
- iv) store the contents of a register into a link field in a node pointed to by a register;
- v) jump if two registers are equal;
- vi) create a new node and make a register point to it;
- vii) halt.

Implement the Fourier-transform multiplication method efficiently on such a machine. [Hints: First show that if N is any positive integer, it is possible to create N nodes representing the integers $\{0, 1, \dots, N - 1\}$, where the node representing p has pointers to the nodes representing $p + 1$, $\lfloor p/2 \rfloor$, and $2p$. These nodes can be created in $O(N)$ steps. Show that arithmetic with radix N can now be simulated without difficulty: for example, it takes $O(\log N)$ steps to find the node for $(p+q) \bmod N$ and to determine if $p+q \geq N$, given pointers to p and q ; and multiplication can be simulated in $O(\log N)^2$ steps. Now consider the algorithm in the text, with $k = l$ and $m = 6k$ and $N = 2^{\lceil m/13 \rceil}$, so that all quantities in the fixed point arithmetic calculations are 13-place integers with radix N . Finally, show that each pass of the fast Fourier transformations can be done in $O(K + (N \log N)^2) = O(K)$ steps, using the following idea: Each of the K necessary assignments can be “compiled” into a bounded list of instructions for a simulated MIX-like computer whose word size is N , and instructions for K such machines acting in parallel can be simulated in $O(K + (N \log N)^2)$ steps if they are first sorted so that all identical instructions are performed together. (Two instructions are identical if they have the same operation code, the same register contents, and the same memory operand contents.) Note that $N^2 = O(n^{12/13})$, so $(N \log N)^2 = O(K)$.]

13. [M25] (A. Schönhage.) What is a good upper bound on the time needed to multiply an m -bit number by an n -bit number, when both m and n are very large but n is much larger than m , based on the results proved in this section for $m = n$?

14. [M42] Write a program for Algorithm C, incorporating the improvements of exercise 4. Compare it with a program for Algorithm 4.3.1M and with a program based on (2), to see how large n must be before Algorithm C is an improvement.

15. [M49] (S. A. Cook.) A multiplication algorithm is said to be *on line* if the $(k+1)$ st input bits of the operands, from right to left, are not read until the k th output bit has been produced. What are the fastest possible on-line multiplication algorithms achievable on various species of automata?

(The best upper bound known is $O(n(\log n)^2 \log \log n)$, due to M. J. Fischer and L. J. Stockmeyer [*J. Comp. and Syst. Sci.* 9 (1974), 317–331]; their construction works on multitape Turing machines, hence also on pointer machines. The best lower bound known is of order $n \log n / \log \log n$, due to M. S. Paterson, M. J. Fischer, and A. R. Meyer [*SIAM/AMS Proceedings* 7 (1974), 97–111]; this applies to multitape Turing machines but not to pointer machines.)

4.4. RADIX CONVERSION

IF OUR ANCESTORS had invented arithmetic by counting with their two fists or their eight fingers, instead of their ten “digits,” we would never have to worry about writing binary-decimal conversion routines. (And we would perhaps never have learned as much about number systems.) In this section, we shall discuss the conversion of numbers from positional notation with one radix into positional notation with another radix; this process is, of course, most important on binary computers when converting decimal input data into binary form, and converting binary answers into decimal form.

A. The four basic methods. Binary-decimal conversion is one of the most machine-dependent operations of all, since computer designers keep inventing different ways to provide for it in the hardware. Therefore we shall discuss only the general principles involved, from which a programmer can select the procedure that is best suited to his machine.

We shall assume that only nonnegative numbers enter into the conversion, since the manipulation of signs is easily accounted for.

Let us assume that we are converting from radix b to radix B . (The methods can also be generalized to mixed-radix notations, as shown in exercises 1 and 2.) Most radix-conversion routines are based on multiplication and division, using one of the following four schemes:

1) Conversion of integers (radix point at the right).

- Method (1a) *Division by B* (using radix- b arithmetic). Given an integer number u , we can obtain its radix- B representation $(U_M \dots U_1 U_0)_B$ as follows:

$$\begin{aligned} U_0 &= u \bmod B \\ U_1 &= \lfloor u/B \rfloor \bmod B \\ U_2 &= \lfloor \lfloor u/B \rfloor /B \rfloor \bmod B \\ &\dots \end{aligned}$$

etc., stopping when $\lfloor \dots \lfloor \lfloor u/B \rfloor /B \rfloor \dots /B \rfloor = 0$.

- Method (1b) *Multiplication by b* (using radix- B arithmetic). If u has the radix- b representation $(u_m \dots u_1 u_0)_b$, we can use radix- B arithmetic to evaluate the polynomial $u_m b^m + \dots + u_1 b + u_0 = u$ in the form

$$((\dots(u_m b + u_{m-1}) b + \dots) b + u_1) b + u_0.$$

- 2) Conversion of fractions (radix point at the left). Note that it is often impossible to express a terminating radix- b fraction $(0.u_{-1}u_{-2}\dots u_{-m})_b$ exactly as a terminating radix- B fraction $(0.U_{-1}U_{-2}\dots U_{-M})_B$. For example, the fraction $\frac{1}{10}$ has the infinite binary representation $(0.0001100110011\dots)_2$. Therefore methods of rounding the result to M places are sometimes necessary.

- Method (2a) *Multiplication by B* (using radix- b arithmetic). Given a fractional number u , we can obtain the digits of its radix- B representation $(.U_{-1}U_{-2}\dots)_B$ as follows:

$$\begin{aligned} U_{-1} &= \lfloor uB \rfloor \\ U_{-2} &= \lfloor \{uB\}B \rfloor \\ U_{-3} &= \lfloor \{\{uB\}B\}B \rfloor \\ &\dots \end{aligned}$$

where $\{x\}$ denotes $x \bmod 1 = x - \lfloor x \rfloor$. If it is desired to round the result to M places, the computation can be stopped after U_{-M} has been calculated, and U_{-M} should be increased by unity if $\{\dots\{uB\}B\}\dots B$ is greater than $\frac{1}{2}$. (Note, however, that this may cause carries to propagate, and these carries must be incorporated into the answer using radix- B arithmetic. It would be simpler to add the constant $\frac{1}{2}B^{-M}$ to the original number u before the calculation begins, but this may lead to a terribly incorrect answer when $\frac{1}{2}B^{-M}$ cannot be represented exactly as a radix- b number inside the computer. Note further that it is possible for the answer to round up to $(1.00\dots 0)_B$, if $b^m \geq 2B^M$.)

Exercise 3 shows how to extend this method so that M is variable, just large enough to represent the original number to a specified accuracy; in this case the problem of carries does not occur.

- Method (2b) *Division by b* (using radix- B arithmetic). If u has the radix- b representation $(0.u_{-1}u_{-2}\dots u_{-m})_b$, we can use radix- B arithmetic to evaluate $u_{-1}b^{-1} + u_{-2}b^{-2} + \dots + u_{-m}b^{-m}$ in the form

$$((\dots(u_{-m}/b + u_{1-m})/b + \dots + u_{-2})/b + u_{-1})/b.$$

Care should be taken to control errors that might occur due to truncation or rounding in the division by b ; these are often negligible, but not always.

To summarize, Methods (1a), (1b), (2a), and (2b) give us two choices for a conversion process, depending on whether our number is an integer or a fraction. And it is certainly possible to convert between integers and fractions by multiplying or dividing by an appropriate power of b or B ; therefore there are at least four methods to choose from when trying to do a conversion.

B. Single-precision conversion. To illustrate these four methods, let us assume that MIX is a binary computer, and suppose that we want to convert a binary integer u to a decimal integer. Thus $b = 2$ and $B = 10$. Method (1a) could be programmed as follows:

ENT1 0	Set $j \leftarrow 0$.
LDX U	
ENTA 0	Set rAX $\leftarrow u$.
1H DIV =10=	$(rA, rX) \leftarrow (\lfloor rAX/10 \rfloor, rAX \bmod 10)$. (1)
STX ANSWER,1	$U_j \leftarrow rX$.
INC1 1	$j \leftarrow j + 1$.
SRAZ 5	$rAX \leftarrow rA$.
JXP 1B	Repeat until result is zero. ■

This requires $18M + 4$ cycles to obtain M digits.

The above method uses division by 10; Method (2a) uses multiplication by 10, so it might be a little faster. In order to use Method (2a), we must deal with fractions, and this leads to an interesting situation. Let w be the word size of the computer, and assume that $u < 10^n < w$. With a single division we can find q and r , where

$$wu = 10^n q + r, \quad 0 \leq r < 10^n. \quad (2)$$

Now if we apply Method (2a) to the fraction $(q+1)/w$, we will obtain the digits of u from left to right, in n steps, since

$$\left\lfloor 10^n \frac{q+1}{w} \right\rfloor = \left\lfloor u + \frac{10^n - r}{w} \right\rfloor = u. \quad (3)$$

(This idea is due to P. A. Samet, *Software—Practice and Experience 1* (1971), 93–96.)

Here is the corresponding MIX program:

JOV OFLO	Ensure overflow is off.
LDA U	
LDX =10 ⁿ =	rAX \leftarrow $wu + 10^n$.
DIV =10 ⁿ =	rA \leftarrow $q + 1$, rX $\leftarrow r$.
JOV ERROR	Jump if $u \geq 10^n$.
ENT1 n-1	Set $j \leftarrow n - 1$.
2H MUL =10=	Now imagine radix point at left, rA = x .
STA ANSWER,1	Set $U_j \leftarrow \lfloor 10x \rfloor$.
SLAX 5	$x \leftarrow \{10x\}$.
DEC1 1	$j \leftarrow j - 1$.
J1NN 2B	Repeat for $n > j \geq 0$. ■

This slightly longer routine requires $16n + 19$ cycles, so it is a little faster than program (1) if $n = M \geq 8$; when leading zeros are present, (1) will be faster.

Program (4) as it stands cannot be used to convert integers $u \geq 10^m$ when $10^m < w < 10^{m+1}$, since we would need to take $n = m + 1$. In this case we can obtain the leading digit of u by computing $\lfloor u/10^m \rfloor$; then $u \bmod 10^m$ can be converted as above with $n = m$.

The fact that the answer digits are obtained from left to right may be an advantage in some applications (e.g., when typing out the answer one digit at a time). Thus we see that a fractional method can be used for conversion of integers, although the use of inexact division makes a little bit of numerical analysis necessary.

A modification of Method (1a) can be used to avoid division by 10, by replacing it with two multiplications. It is worth mentioning this modification here, because radix conversion is often done by small “satellite” computers that have no division capability. If we let x be an approximation to $\frac{1}{10}$, so that $\frac{1}{10} < x < \frac{1}{10} + 1/w$, it is easy to prove (see exercise 7) that $\lfloor ux \rfloor = \lfloor u/10 \rfloor$ or $\lfloor u/10 \rfloor + 1$, so long as $0 \leq u < w$. Therefore, if we compute $u - 10\lfloor ux \rfloor$, we

will be able to determine the value of $\lfloor u/10 \rfloor$:

$$\lfloor u/10 \rfloor = \begin{cases} \lfloor ux \rfloor, & \text{if } u - 10\lfloor ux \rfloor \geq 0; \\ \lfloor ux \rfloor - 1, & \text{if } u - 10\lfloor ux \rfloor < 0. \end{cases} \quad (5)$$

This procedure simultaneously determines $u \bmod 10$. A MIX program for conversion using this idea appears in exercise 8; it requires about 33 cycles per digit.

The procedure represented by (5) can be used effectively even if the computer has no built-in multiplication instruction, since multiplication by 10 consists of two shifts and one addition ($10 = 2^3 + 2$). Even the task of multiplication by $\frac{1}{10}$ can be done by judiciously combining a few shifting and adding operations, as explained in exercise 9.

Another way to convert from binary to decimal is to use Method (1b), but to do this we need to simulate doubling in a decimal number system. This idea is generally most suitable for incorporation into computer hardware; however, it is possible to program the doubling process for decimal numbers, using binary addition, binary shifting, and binary extraction (“logical AND” on each bit in the register), as shown in the following table.

Table 1

DOUBLING A BINARY-CODED DECIMAL NUMBER

Operation	General form	Example
1. Given number	$u_1 u_2 u_3 u_4 \ u_5 u_6 u_7 u_8 \ u_9 u_{10} u_{11} u_{12}$	$0011 \ 0110 \ 1001 = 3 \ 6 \ 9$
2. Add 3 to each digit	$v_1 v_2 v_3 v_4 \ v_5 v_6 v_7 v_8 \ v_9 v_{10} v_{11} v_{12}$	$0110 \ 1001 \ 1100$
3. Shift left one	$v_1 \ v_2 v_3 v_4 v_5 \ v_6 v_7 v_8 v_9 \ v_{10} v_{11} v_{12} \ 0$	$0 \ 1101 \ 0011 \ 1000$
4. Extract low bit	$v_1 \ 0 \ 0 \ 0 \ v_5 \ 0 \ 0 \ 0 \ v_9 \ 0 \ 0 \ 0 \ 0$	$0 \ 0001 \ 0001 \ 0000$
5. Shift right two	$0 \ v_1 \ 0 \ 0 \ 0 \ v_5 \ 0 \ 0 \ 0 \ v_9 \ 0 \ 0$	$0000 \ 0100 \ 0100$
6. Shift right one and add	$0 \ v_1 v_1 \ 0 \ 0 \ v_5 v_5 \ 0 \ 0 \ v_9 \ v_9 \ 0$	$0000 \ 0110 \ 0110$
7. Add result of step 3	$* \ * \ * \ * \ * \ * \ * \ * \ * \ 0$	$0 \ 1101 \ 1001 \ 1110$
8. Subtract 6 from each	$y_1 \ y_2 y_3 y_4 y_5 \ y_6 y_7 y_8 y_9 \ y_{10} y_{11} y_{12} \ 0$	$0 \ 0111 \ 0011 \ 1000 = 7 \ 3 \ 8$

This method changes each individual digit d into $((d+3) \times 2 + 0) - 6 = 2d$ when $0 \leq d \leq 4$, and into $((d+3) \times 2 + 6) - 6 = (2d - 10) + 2^4$ when $5 \leq d \leq q$; and that is just what is needed to double decimal numbers encoded with 4 bits per digit.

Another related idea is to keep a table of the powers of two in decimal form, and to add the appropriate powers together by simulating decimal addition. A survey of bit-manipulation techniques appears in Section 7.1.

Finally, even Method (2b) can be used for the conversion of binary integers to decimal integers. We can find q as in (2), and then we can simulate the decimal division of $q + 1$ by w , using a “halving” process (exercise 10) that is similar to the doubling process just described, retaining only the first n digits to the right of the radix point in the answer. In this situation, Method (2b) does not seem to offer advantages over the other three methods already discussed, but we have confirmed the remark made earlier that at least four distinct methods are available for converting integers from one radix to another.

Now let us consider decimal-to-binary conversion (so that $b = 10$, $B = 2$). Method (1a) simulates a decimal division by 2; this is feasible (see exercise 10), but it is primarily suitable for incorporation in hardware instead of programs.

Method (1b) is the most practical method for decimal-to-binary conversion in the great majority of cases. Here it is in MIX code, assuming that there are at least two digits in the number $(u_m \dots u_1 u_0)_{10}$ being converted:

ENT1 M-1	Set $j \leftarrow m - 1$.
LDA INPUT+M	Set $U \leftarrow u_m$.
1H MUL =10=	
SLAX 5	
ADD INPUT,1	$U \leftarrow 10U + u_j$.
DEC1 1	
J1NN 1B	Repeat for $m > j \geq 0$. ■

(6)

Note again that adding and shifting may be substituted for the multiplication by 10.

A trickier but perhaps faster method, which uses about $\lg m$ multiplications, extractions, and additions instead of m multiplications and additions, is described in exercise 19.

For the conversion of decimal fractions $(0.u_{-1}u_{-2} \dots u_{-m})_{10}$ to binary form, we can use Method (2b); or, more commonly, we can convert the integer $(u_{-1}u_{-2} \dots u_{-m})_{10}$ by Method (1b) and then divide by 10^m .

C. Hand calculation. It is occasionally necessary for computer programmers to convert numbers by hand, and since this is a subject not yet taught in elementary schools, it may be worthwhile to examine it briefly here. There are very simple pencil-and-paper methods for converting between decimal and octal notations, and these methods are easily learned, so they ought to be more widely known.

Converting octal integers to decimal. The simplest conversion is from octal to decimal; this technique was apparently first published by Walter Soden, *Math. Comp.* 7 (1953), 273–274. To do the conversion, write down the given octal number; then at the k th step, double the k leading digits using decimal arithmetic, and subtract this from the $k + 1$ leading digits using decimal arithmetic. The process terminates in $n - 1$ steps if the given number has n digits. It is a good idea to insert a radix point to show which digits are being doubled, as shown in the following example, in order to prevent embarrassing mistakes.

Example 1. Convert $(5325121)_8$ to decimal.

$$\begin{array}{r}
 5.3\ 2\ 5\ 1\ 2\ 1 \\
 -1\ 0 \\
 \hline
 4\ 3.2\ 5\ 1\ 2\ 1 \\
 -8\ 6 \\
 \hline
 3\ 4\ 6.5\ 1\ 2\ 1 \\
 -6\ 9\ 2 \\
 \hline
 2\ 7\ 7\ 3.1\ 2\ 1 \\
 -5\ 5\ 4\ 6 \\
 \hline
 2\ 2\ 1\ 8\ 5.2\ 1 \\
 -4\ 4\ 3\ 7\ 0 \\
 \hline
 1\ 7\ 7\ 4\ 8\ 2.1 \\
 -3\ 5\ 4\ 9\ 6\ 4 \\
 \hline
 1\ 4\ 1\ 9\ 8\ 5\ 7
 \end{array}$$

Answer: $(14198757)_{10}$.

A reasonably good check on the computations may be had by “casting out nines”: The sum of the digits of the decimal number must be congruent modulo 9 to the alternating sum and difference of the digits of the octal number, with the rightmost digit of the latter given a plus sign. In the above example, we have $1 + 4 + 1 + 9 + 8 + 5 + 7 = 35$, and $1 - 2 + 1 - 5 + 2 - 3 + 5 = -1$; the difference is 36 (a multiple of 9). If this test fails, it can be applied to the $k + 1$ leading digits after the k th step, and the error can be located using a “binary search” procedure; i.e., we start by checking the middle result, then use the same procedure on the first or second half of the calculation, depending on whether the middle result is incorrect or correct.

The “casting-out-nines” process is only about 89 percent reliable, because there is one chance in nine that two *random* integers will differ by a multiple of nine. An even better check is to convert the answer back to octal by using an inverse method, which we shall now consider.

Converting decimal integers to octal. A similar procedure can be used for the opposite conversion: Write down the given decimal number; then at the k th step, double the k leading digits using *octal* arithmetic, and *add* these to the $k + 1$ leading digits using *octal* arithmetic. The process terminates in $n - 1$ steps if the given number has n digits. (See Example 2 on the following page.)

The two procedures just given are essentially Method (1b) of the general radix-conversion procedures. Doubling and subtracting in decimal notation is like multiplying by $10 - 2 = 8$; doubling and adding in octal notation is like multiplying by $8 + 2 = 10$. There is a similar method for hexadecimal/decimal conversions, but it is a little more difficult since it involves multiplication by 6 instead of by 2.

To keep these two methods straight in our minds, it is not hard to remember that we must subtract to go from octal to decimal, since the decimal representation of a number is smaller; similarly we must add to go from decimal to

Example 2. Convert $(1419857)_{10}$ to octal.

$$\begin{array}{r}
 1.4\ 1\ 9\ 8\ 5\ 7 \\
 +\ 2 \\
 \hline
 1\ 6.\ 1\ 9\ 8\ 5\ 7 \\
 +\ 3\ 4 \\
 \hline
 2\ 1\ 5.9\ 8\ 5\ 7 \\
 +\ 4\ 3\ 2 \\
 \hline
 2\ 6\ 1\ 3.8\ 5\ 7 \\
 +\ 5\ 4\ 2\ 6 \\
 \hline
 3\ 3\ 5\ 6\ 6.5\ 7 \\
 +\ 6\ 7\ 3\ 5\ 4 \\
 \hline
 4\ 2\ 5\ 2\ 4\ 1.7 \\
 +\ 1\ 0\ 5\ 2\ 5\ 0\ 2 \\
 \hline
 5\ 3\ 2\ 5\ 1\ 2\ 1
 \end{array}$$

(Note that the nonoctal digits 8 and 9 enter into this octal computation.) The answer can be checked as discussed above. This method was published by Charles P. Rozier, *IEEE Trans. CE-11* (1962), 708-709.

Answer: $(5325121)_8$.

octal. The computations are performed using the radix of the answer, not the radix of the given number, otherwise we couldn't get the desired answer.

Converting fractions. No equally fast method of converting fractions manually is known; the best way seems to be Method (2a), with doubling and adding or subtracting to simplify the multiplications by 10 or by 8. In this case, we reverse the addition-subtraction criterion, adding when we convert to decimal and subtracting when we convert to octal; we also use the radix of the given input number, not the radix of the answer, in this computation (see Examples 3 and 4). The process is about twice as hard as the above method for integers.

Example 3. Convert $(.14159)_{10}$ to octal.

$$\begin{array}{r}
 .1\ 4\ 1\ 5\ 9 \\
 2\ 8\ 3\ 1\ 8 - \\
 \hline
 1.1\ 3\ 2\ 7\ 2 \\
 2\ 6\ 5\ 4\ 4 - \\
 \hline
 1.0\ 6\ 1\ 7\ 6 \\
 1\ 2\ 3\ 5\ 2 - \\
 \hline
 0.4\ 9\ 4\ 0\ 8 \\
 9\ 8\ 8\ 1\ 6 - \\
 \hline
 3.9\ 5\ 2\ 4\ 6 \\
 1\ 9\ 0\ 5\ 2\ 8 - \\
 \hline
 7.6\ 2\ 1\ 1\ 2 \\
 1\ 2\ 4\ 2\ 2\ 4 - \\
 \hline
 4.9\ 6\ 8\ 9\ 6
 \end{array}$$

Answer: $(.110374\dots)_8$.

Example 4. Convert $(.110374)_8$ to decimal.

$$\begin{array}{r}
 .1\ 1\ 0\ 3\ 7\ 4 \\
 2\ 2\ 0\ 7\ 7\ 0 + \\
 \hline
 1.3\ 2\ 4\ 7\ 3\ 0 \\
 6\ 5\ 1\ 6\ 6\ 0 + \\
 \hline
 4.1\ 2\ 1\ 1\ 6\ 0 \\
 2\ 4\ 2\ 3\ 4\ 0 + \\
 \hline
 1.4\ 5\ 4\ 1\ 4\ 0 \\
 1\ 1\ 3\ 0\ 3\ 0\ 0 + \\
 \hline
 5.6\ 7\ 1\ 7\ 0\ 0 \\
 1\ 5\ 6\ 3\ 6\ 0\ 0 + \\
 \hline
 8.5\ 0\ 2\ 6\ 0\ 0 \\
 1\ 2\ 0\ 5\ 4\ 0\ 0 + \\
 \hline
 6.2\ 3\ 3\ 4\ 0\ 0
 \end{array}$$

Answer: $(.141586\dots)_{10}$.

D. Floating point conversion. When floating point values are to be converted, it is necessary to deal with both the exponent and the fraction parts simultaneously, since conversion of the exponent will affect the fraction part. Given the number $f \cdot 2^e$ to be converted to decimal, we may express 2^e in the form $F \cdot 10^E$ (usually by means of auxiliary tables), and then convert Ff to decimal. Alternatively, we can multiply e by $\log_{10} 2$ and round this to the nearest integer E ; then divide $f \cdot 2^e$ by 10^E and convert the result. Conversely, given the number $F \cdot 10^E$ to be converted to binary, we may convert F and then multiply it by the floating point number 10^E (again by using auxiliary tables). Obvious techniques can be used to reduce the maximum size of the auxiliary tables by using several multiplications and/or divisions, although this can cause rounding errors to propagate.

E. Multiple-precision conversion. When converting extremely long numbers, it is most convenient to start by converting blocks of digits, which can be handled by single-precision techniques, and then to combine these blocks by using simple multiple-precision techniques. For example, suppose that 10^n is the highest power of 10 less than the computer word size. Then

- a) To convert a multiple-precision integer from binary to decimal, divide it repeatedly by 10^n (thus converting from binary to radix 10^n by Method (1a)). Single-precision operations will give the n decimal digits for each place of the radix- 10^n representation.
- b) To convert a multiple-precision fraction from binary to decimal, proceed similarly, multiplying by 10^n (i.e., using Method (2a) with $B = 10^n$).
- c) To convert a multiple-precision integer from decimal to binary, convert blocks of n digits first; then use Method (1b) to convert from radix 10^n to binary.
- d) To convert a multiple-precision fraction from decimal to binary, convert first to radix 10^n as in (c), then use Method (2b).

F. History and Bibliography. Radix-conversion techniques implicitly originated in ancient problems dealing with weights, measures, and currencies, where mixed-radix systems were generally involved; auxiliary tables were usually prepared to help make the conversions. During the seventeenth century, when sexagesimal fractions were being supplanted by decimal fractions, it was necessary to convert between the two systems in order to use existing books of astronomical tables; a systematic method to transform fractions from radix 60 to radix 10 and vice versa was given in the 1667 edition of William Oughtred's *Clavis Mathematicæ*, Chapter 6, Section 18. (This material was not present in the original 1631 edition of Oughtred's book.) Conversion rules had already been given by al-Kashî of Persia in his *Key to Arithmetic* (c. 1414), where Methods (1a), (1b), and (2a) are clearly explained [*Istoriko-Mat. Issled.* 7 (1954), 126–135], but his work was unknown in Europe. The 18th century American mathematician Hugh Jones used the words "octavation" and "decimation" to describe octal/decimal conversions, but his methods were not as clever as his terminology. A. M. Legendre [*Théorie des nombres* (Paris: 1798), 229] noted that positive integers may be conveniently converted to binary form if they are repeatedly divided by 64.

In 1946, H. H. Goldstine and J. von Neumann gave prominent consideration to radix conversion in their classic memoir, "Planning and coding problems for an electronic computing instrument," because it was necessary to justify the use of binary arithmetic; see John von Neumann, *Collected Works 5* (New York: Macmillan, 1963), 127–142. Another early discussion of radix conversion on binary computers was published by F. Koons and S. Lubkin, *Math. Comp.* 3 (1949), 427–431, who suggested a rather unusual method. The first discussion of floating point conversion was given somewhat later by F. L. Bauer and K. Samelson [*Zeit. für angewandte Math. und Physik* 4 (1953), 312–316].

The following articles may be useful for further reference: A note by G. T. Lake [CACM 5 (1962), 468–469] mentions some hardware techniques for conversion and gives clear examples. A. H. Stroud and D. Secrest [*Comp. J.* 6 (1963), 62–66] have discussed conversion of multiple-precision floating point numbers. The conversion of *unnormalized* floating point numbers, preserving the amount of "significance" implied by the representation, has been discussed by H. Kanner [*JACM* 12 (1965), 242–246] and by N. Metropolis and R. L. Ashenhurst [*Math. Comp.* 19 (1965), 435–441]. See also K. Sikdar, *Sankhyā (B)* 30 (1968), 315–334, and the references cited in his paper.

EXERCISES

- 1. [25] Generalize Method (1b) so that it works with arbitrary mixed-radix notations, converting

$$a_m b_{m-1} \dots b_1 b_0 + \dots + a_1 b_0 + a_0 \quad \text{into} \quad A_M B_{M-1} \dots B_1 B_0 + \dots + A_1 B_0 + A_0,$$

where $0 \leq a_j < b_j$ and $0 \leq A_J < B_J$ for $0 \leq j < m$ and $0 \leq J < M$.

Give an example of your generalization by manually converting the quantity "3 days, 9 hours, 12 minutes, and 37 seconds" into long tons, hundredweights, stones, pounds, and ounces. (Let one second equal one ounce. The British system of weights has 1 stone = 14 pounds, 1 hundredweight = 8 stone, 1 long ton = 20 hundredweight.) In other words, let $b_0 = 60$, $b_1 = 60$, $b_2 = 24$, $m = 3$, $B_0 = 16$, $B_1 = 14$, $B_2 = 8$, $B_3 = 20$, $M = 4$; the problem is to find A_4, \dots, A_0 in the proper ranges such that $3b_2 b_1 b_0 + 2b_1 b_0 + 12b_0 + 37 = A_4 B_3 B_2 B_1 B_0 + A_3 B_2 B_1 B_0 + A_2 B_1 B_0 + A_1 B_0 + A_0$, using a systematic method that generalizes Method (1b). (All arithmetic is to be done in a mixed-radix system.)

2. [25] Generalize Method (1a) so that it works with mixed-radix notations, as in exercise 1, and give an example of your generalization by manually solving the same conversion problem stated in exercise 1.

- 3. [25] (D. Taranto.) When fractions are being converted, there is no obvious way to decide how many digits to give in the answer. Design a simple generalization of Method (2a) that, given two positive radix- b fractions u and ϵ between 0 and 1, converts u to a rounded radix- B equivalent U that has just enough places M to the right of the radix point to ensure that $|U - u| < \epsilon$. (In particular if u is a multiple of b^{-m} and $\epsilon = b^{-m}/2$, the value of U will have just enough digits so that u can be recomputed exactly, given U and m . Note that M might be zero; for example, if $\epsilon \leq \frac{1}{2}$ and $u > 1 - \epsilon$, the proper answer is $U = 1$.)

4. [M21] (a) Prove that every real number with a terminating binary representation also has a terminating decimal representation. (b) Find a simple condition on the positive integers b and B that is satisfied if and only if every real number that has a terminating radix- b representation also has a terminating radix- B representation.

5. [M20] Show that program (4) would still work if the instruction "LDX =10ⁿ=" were replaced by "LDX =c=" for certain other constants c .

6. [30] Discuss using Methods (1a), (1b), (2a), and (2b) when b or B is -2 .

7. [M18] Given that $0 < \alpha \leq x \leq \alpha + 1/w$ and $0 \leq u \leq w$, prove that $\lfloor ux \rfloor$ is equal to either $\lfloor \alpha u \rfloor$ or $\lfloor \alpha u \rfloor + 1$. Furthermore $\lfloor ux \rfloor = \lfloor \alpha u \rfloor$ exactly, if $u < \alpha w$ and α^{-1} is an integer.

8. [24] Write a MIX program analogous to (1) that uses (5) and includes no division instructions.

9. [M27] Let u be an integer, $0 \leq u < 2^{34}$. Assume that the following sequence of operations (equivalent to addition and binary "shift-right" instructions) is performed:

$$\begin{aligned} v &\leftarrow \lfloor \frac{1}{2}u \rfloor, & v &\leftarrow v + \lfloor \frac{1}{2}v \rfloor, & v &\leftarrow v + \lfloor 2^{-4}v \rfloor, \\ v &\leftarrow v + \lfloor 2^{-8}v \rfloor, & v &\leftarrow v + \lfloor 2^{-16}v \rfloor, & v &\leftarrow \lfloor \frac{1}{8}v \rfloor. \end{aligned}$$

Prove that $v = \lfloor u/10 \rfloor$ or $\lfloor u/10 \rfloor - 1$.

10. [22] The text shows how a binary-coded decimal number can be doubled by using various shifting, extracting, and addition operations on a binary computer. Give an analogous method that computes half of a binary-coded decimal number (throwing away the remainder if the number is odd).

11. [16] Convert $(57721)_8$ to decimal.

► 12. [22] Invent a rapid pencil-and-paper method for converting integers from ternary notation to decimal, and illustrate your method by converting $(1212011210210)_3$ into decimal. How would you go from decimal to ternary?

► 13. [25] Assume that locations $U + 1, U + 2, \dots, U + m$ contain a multiple-precision fraction $(.u_{-1}u_{-2}\dots u_{-m})_b$, where b is the word size of MIX. Write a MIX routine that converts this fraction to decimal notation, truncating it to 180 decimal digits. The answer should be printed on two lines, with the digits grouped into 20 blocks of nine each separated by blanks. (Use the CHAR instruction.)

► 14. [M27] (A. Schönhage.) The text's method of converting multiple-precision integers requires an execution time of order n^2 to convert an n -place integer, when n is large. Show that it is possible to convert n -digit decimal integers into binary notation in $O(M(n)\log n)$ steps, where $M(n)$ is an upper bound on the number of steps needed to multiply n -bit binary numbers that satisfies the "smoothness condition" $M(2n) \geq 2M(n)$.

15. [M47] Can the upper bound on the time to convert large integers, given in exercise 14, be substantially lowered? (Cf. exercise 4.3.3-12.)

16. [41] Construct a fast linear iterative array for radix conversion from decimal to binary (cf. Section 4.3.3).

17. [M40] Design "ideal" floating point conversion subroutines, taking p -digit decimal numbers into P -digit binary numbers and vice versa, in both cases producing a true rounded result in the sense of Section 4.2.2.

18. [HMS34] (David W. Matula.) Let $\text{round}_b(u, p)$ be the function of b , u , and p that represents the best p -digit base b floating point approximation to u , in the sense of Section 4.2.2. Under the assumption that $\log_B b$ is irrational and that the range of exponents is unlimited, prove that

$$u = \text{round}_b(\text{round}_B(u, P), p)$$

holds for all p -digit base b floating point numbers u if and only if $B^{P-1} \geq b^p$. (In other words, an “ideal” input conversion of u into an independent base B , followed by an “ideal” output conversion of this result, will always yield u again if and only if the intermediate precision P is suitably large, as specified by the formula above.)

19. [M29] Let the decimal number $u = (u_7 \dots u_1 u_0)_{10}$ be represented as the binary-coded decimal number $U = (u_7 \dots u_1 u_0)_{16}$. Find appropriate constants c_i and masks m_i so that the operation $U \leftarrow U - c_i(U \wedge m_i)$, repeated for $i = 1, 2, 3$, will convert U to the binary representation of u , where “ \wedge ” denotes extraction (i.e., “logical AND” on individual bits).

4.5. RATIONAL ARITHMETIC

IT IS OFTEN important to know that the answer to some numerical problem is exactly $\frac{1}{3}$, not a floating point number that gets printed as "0.333333574". If arithmetic is done on fractions instead of on approximations to fractions, many computations can be done entirely *without any accumulated rounding errors*. This results in a comfortable feeling of security that is often lacking when floating point calculations have been made, and it means that the accuracy of the calculation cannot be improved upon.

4.5.1. Fractions

When fractional arithmetic is desired, the numbers can be represented as pairs of integers, (u/u') , where u and u' are relatively prime to each other and $u' > 0$. The number zero is represented as $(0/1)$. In this form, $(u/u') = (v/v')$ if and only if $u = v$ and $u' = v'$.

Multiplication of fractions is, of course, easy; to form $(u/u') \times (v/v') = (w/w')$, we can simply compute uv and $u'v'$. The two products uv and $u'v'$ might not be relatively prime, but if $d = \gcd(uv, u'v')$, the desired answer is $w = uv/d$, $w' = u'v'/d$. (See exercise 2.) Efficient algorithms to compute the greatest common divisor are discussed in Section 4.5.2.

Another way to perform the multiplication is to find $d_1 = \gcd(u, v')$ and $d_2 = \gcd(u', v)$; then the answer is $w = (u/d_1)(v/d_2)$, $w' = (u'/d_2)(v'/d_1)$. (See exercise 3.) This method requires two gcd calculations, but it is not really slower than the former method; the gcd process involves a number of iterations that is essentially proportional to the logarithm of its inputs, so the total number of iterations needed to evaluate both d_1 and d_2 is essentially the same as the number of iterations during the single calculation of d . Furthermore, each iteration in the evaluation of d_1 and d_2 is potentially faster, because comparatively small numbers are being examined. If u , u' , v , and v' are single-precision quantities, this method has the advantage that no double-precision numbers appear in the calculation unless it is impossible to represent both of the answers w and w' in single-precision form.

Division may be done in a similar manner; see exercise 4.

Addition and subtraction are slightly more complicated. The obvious procedure is to set $(u/u') \pm (v/v') = ((uv' \pm u'v)/u'v')$ and then to reduce this fraction to lowest terms by calculating $d = \gcd(uv' \pm u'v, u'v')$ as in the first multiplication method. But again it is possible to avoid working with such large numbers, if we start by calculating $d_1 = \gcd(u', v')$. If $d_1 = 1$, then the desired numerator and denominator are $w = uv' \pm u'v$ and $w' = u'v'$. (According to Theorem 4.5.2D, d_1 will be 1 about 61 percent of the time, if the denominators u' and v' are randomly distributed, so it is wise to single this case out separately.) If $d_1 > 1$, then let $t = u(v'/d_1) \pm v(u'/d_1)$ and calculate $d_2 = \gcd(t, d_1)$; finally the answer is $w = t/d_2$, $w' = (u'/d_1)(v'/d_2)$. (Exercise 6 proves that these values of w and w' are relatively prime to each other.) If single-precision

numbers are being used, this method requires only single-precision operations, except that t may be a double-precision number or slightly larger (see exercise 7); since $\gcd(t, d_1) = \gcd(t \bmod d_1, d_1)$, the calculation of d_2 does not require double precision.

For example, to compute $(7/66) + (17/12)$, we form $d_1 = \gcd(66, 12) = 6$; then $t = 7 \cdot 2 + 17 \cdot 11 = 201$, and $d_2 = \gcd(201, 6) = 3$, so the answer is

$$\frac{201}{3} / \frac{66}{6} \frac{12}{3} = 67/44.$$

To help check out subroutines for rational arithmetic, inversion of matrices with known inverses (e.g., Cauchy matrices, exercise 1.2.3–41) is suggested.

Experience with fractional calculations shows that in many cases the numbers grow to be quite large. So if u and u' are intended to be single-precision numbers for each fraction (u/u') , it is important to include tests for overflow in each of the addition, subtraction, multiplication, and division subroutines. For numerical problems in which perfect accuracy is important, a set of subroutines for fractional arithmetic with arbitrary precision allowed in numerator and denominator is very useful.

The methods of this section extend also to other number fields besides the rational numbers; for example, we could do arithmetic on quantities of the form $(u + u'\sqrt{5})/u''$, where u, u', u'' are integers, $\gcd(u, u', u'') = 1$, and $u'' > 0$; or on quantities of the form $(u + u'\sqrt[3]{2} + u''\sqrt[3]{4})/u'''$, etc.

Instead of insisting on exact calculations with fractions, it is interesting to consider also “fixed slash” and “floating slash” numbers, which are analogous to floating point numbers but based on rational fractions instead of radix-oriented fractions. In a binary fixed-slash scheme, the numerator and denominator of a representable fraction each consist of at most p bits, for some given p . In a floating-slash scheme, the sum of numerator bits plus denominator bits must be a total of at most q , for some given q , and another field of the representation is used to indicate how many of these q bits belong to the numerator. To do arithmetic on such numbers, we define $x \oplus y = \text{round}(x + y)$, $x \ominus y = \text{round}(x - y)$, etc., where $\text{round}(x) = x$ if x is representable, otherwise it is one of the two representable numbers that surround x .

It may seem at first that the best definition of $\text{round}(x)$ would be to choose the representable number that is closest to x , by analogy with the way we round in floating point arithmetic. But experience has shown that it is best to bias our rounding towards “simple” numbers, since numbers with small numerator and denominator occur much more often than complicated fractions do. We want more numbers to round to $\frac{1}{2}$ than to $\frac{127}{255}$. The rounding rule that turns out to be most successful in practice is called “median rounding”: If (u/u') and (v/v') are adjacent representable numbers, so that whenever $u/u' \leq x \leq v/v'$ we must have $\text{round}(x)$ equal to (u/u') or (v/v') , the median rounding rule says that

$$\text{round}(x) = \frac{u}{u'} \text{ for } x < \frac{u+v}{u'+v'}, \quad \text{round}(x) = \frac{v}{v'} \text{ for } x > \frac{u+v}{u'+v'}. \quad (1)$$

If $x = (u + v)/(u' + v')$ exactly, we let $\text{round}(x)$ be the neighboring fraction with the smallest denominator (or, if $u' = v'$, with the smallest numerator).

For example, suppose we are doing fixed slash arithmetic with $p = 8$, so that the representable numbers (u/u') have $-128 < u < 128$ and $0 \leq v < 256$ and $\gcd(u, u') = 1$. This isn't much precision, but it is enough to give us a feel for slash arithmetic. The numbers adjacent to $0 = (0/1)$ are $(-1/255)$ and $(1/255)$; according to the mediant rounding rule, we will therefore have $\text{round}(x) = 0$ if and only if $|x| \leq 1/256$. Suppose we have a calculation that would take the overall form $\frac{22}{7} = \frac{314}{159} + \frac{1300}{1113}$ if we were working in exact rational arithmetic, but the intermediate quantities have had to be rounded to representable numbers. In this case $\frac{314}{159}$ would round to $(79/40)$ and $\frac{1300}{1113}$ would round to $(7/6)$. The sum $\frac{79}{40} + \frac{7}{6} = \frac{377}{120}$ rounds to $(22/7)$, so we have obtained the correct answer even though three roundings were required. This example was not specially contrived; when the answer to a problem is a simple fraction, slash arithmetic tends to make the intermediate rounding errors cancel out.

Exact representation of fractions within a computer was first discussed in the literature by P. Henrici, *JACM* 3 (1956), 6–9. Fixed and floating slash arithmetic was proposed by David W. Matula, in *Applications of Number Theory to Numerical Analysis*, ed. by S. K. Zaremba (New York: Academic Press, 1972), 486–489. Further developments of the idea are discussed by Matula and Kornerup in *Proc. IEEE Symp. Computer Arith.* 4 (1978), 29–47; *Lecture Notes in Comp. Sci.* 72 (1979), 383–397; *Computing, Suppl.* 2 (1980), 85–111.

EXERCISES

1. [15] Suggest a reasonable computational method for comparing two fractions, to test whether or not $(u/u') < (v/v')$.
2. [M15] Prove that if $d = \gcd(u, v)$ then u/d and v/d are relatively prime.
3. [M20] Prove that if u and u' are relatively prime, and if v and v' are relatively prime, then $\gcd(uv, u'v') = \gcd(u, v')\gcd(u', v)$.
4. [11] Design a division algorithm for fractions, analogous to the second multiplication method of the text. (Note that the sign of v must be considered.)
5. [10] Compute $(17/120) + (-27/70)$ by the method recommended in the text.
- 6. [M23] Show that if u, u' are relatively prime and if v, v' are relatively prime, then $\gcd(uv' + vu', u'v') = d_1d_2$, where $d_1 = \gcd(u', v')$ and $d_2 = \gcd(d_1, u(v'/d_1) + v(u'/d_1))$. (Hence if $d_1 = 1$, then $uv' + vu'$ is relatively prime to $u'v'$.)
7. [M22] How large can the absolute value of the quantity t become, in the addition-subtraction method recommended in the text, if the numerators and denominators of the inputs are less than N in absolute value?
- 8. [22] Discuss using $(1/0)$ and $(-1/0)$ as representations for ∞ and $-\infty$, and/or as representations of overflow.
9. [M23] If $1 \leq u', v' < 2^n$, show that $\lfloor 2^{2n}u/u' \rfloor = \lfloor 2^{2n}v/v' \rfloor$ implies $u/u' = v/v'$.
10. [41] Extend the subroutines suggested in exercise 4.3.1–34 so that they deal with “arbitrary” rational numbers.

11. [M23] Consider fractions of the form $(u + u'\sqrt{5})/u''$, where u, u', u'' are integers, $\gcd(u, u', u'') = 1$, and $u'' > 0$. Explain how to divide two such fractions and to obtain a quotient having the same form.

12. [20] (Matula and Kornerup.) Discuss the representation of floating slash numbers in a 32-bit word.

13. [M23] Explain how to compute the exact number of pairs of integers (u, u') such that $1 \leq u \leq M$ and $N_1 < u' \leq N_2$ and $\gcd(u, u') = 1$. (This can be used to determine how many numbers are representable in slash arithmetic. According to Theorem 4.5.2D, the number will be approximately $(6/\pi^2)M(N_2 - N_1)$.)

14. [42] Modify one of the compilers at your installation so that it will replace all floating point calculations by floating slash calculations. Experiment with the use of slash arithmetic by running existing programs that were written by programmers who actually had floating point arithmetic in mind. (When special subroutines like square root or logarithm are called, your system should automatically convert slash numbers to floating point form before the subroutine is invoked, then back to slash form again afterwards. There should be a new option to print slash numbers in a fractional format; however, if you make no changes to a user's source program, you probably will have to print slash numbers in decimal notation, in order to keep from messing up any column alignments.) Are the results better or worse, when floating slash numbers are substituted?

4.5.2. The Greatest Common Divisor

If u and v are integers, not both zero, we say that their *greatest common divisor*, $\gcd(u, v)$, is the largest integer that evenly divides both u and v . This definition makes sense, because if $u \neq 0$ then no integer greater than $|u|$ can evenly divide u , but the integer 1 does divide both u and v ; hence there must be a largest integer that divides them both. When u and v are both zero, every integer evenly divides zero, so the above definition does not apply; it is convenient to set

$$\gcd(0, 0) = 0. \quad (1)$$

The definitions just given obviously imply that

$$\gcd(u, v) = \gcd(v, u), \quad (2)$$

$$\gcd(u, v) = \gcd(-u, v), \quad (3)$$

$$\gcd(u, 0) = |u|. \quad (4)$$

In the previous section, we reduced the problem of expressing a rational number in "lowest terms" to the problem of finding the greatest common divisor of its numerator and denominator. Other applications of the greatest common divisor have been mentioned for example in Sections 3.2.1.2, 3.3.3, 4.3.2, 4.3.3. So the concept of $\gcd(u, v)$ is important and worthy of serious study.

The *least common multiple* of two integers u and v , written $\text{lcm}(u, v)$, is a related idea that is also important. It is defined to be the smallest positive integer

that is a multiple of (i.e., evenly divisible by) both u and v ; and $\text{lcm}(0, 0) = 0$. The classical method for teaching children how to add fractions $u/u' + v/v'$ is to train them to find the “least common denominator,” which is $\text{lcm}(u', v')$.

According to the “fundamental theorem of arithmetic” (proved in exercise 1.2.4–21), each positive integer u can be expressed in the form

$$u = 2^{u_2} 3^{u_3} 5^{u_5} 7^{u_7} 11^{u_{11}} \dots = \prod_{p \text{ prime}} p^{u_p}, \quad (5)$$

where the exponents u_2, u_3, \dots are uniquely determined nonnegative integers, and where all but a finite number of the exponents are zero. From this canonical factorization of a positive integer, it is easy to discover one way to compute the greatest common divisor of u and v : By (2), (3), and (4), we may assume that u and v are positive integers, and if both of them have been canonically factored into primes we have

$$\gcd(u, v) = \prod_{p \text{ prime}} p^{\min(u_p, v_p)}, \quad (6)$$

$$\text{lcm}(u, v) = \prod_{p \text{ prime}} p^{\max(u_p, v_p)}. \quad (7)$$

Thus, for example, the greatest common divisor of $u = 7000 = 2^3 \cdot 5^3 \cdot 7$ and $v = 4400 = 2^4 \cdot 5^2 \cdot 11$ is $2^{\min(3, 4)} 5^{\min(3, 2)} 7^{\min(1, 0)} 11^{\min(0, 1)} = 2^3 \cdot 5^2 = 200$. The least common multiple of the same two numbers is $2^4 \cdot 5^3 \cdot 7 \cdot 11 = 154000$.

From formulas (6) and (7) we can easily prove a number of basic identities concerning the gcd and the lcm:

$$\gcd(u, v)w = \gcd(uw, vw), \quad \text{if } w \geq 0; \quad (8)$$

$$\text{lcm}(u, v)w = \text{lcm}(uw, vw), \quad \text{if } w \geq 0; \quad (9)$$

$$u \cdot v = \gcd(u, v) \cdot \text{lcm}(u, v), \quad \text{if } u, v \geq 0; \quad (10)$$

$$\gcd(\text{lcm}(u, v), \text{lcm}(u, w)) = \text{lcm}(u, \gcd(v, w)); \quad (11)$$

$$\text{lcm}(\gcd(u, v), \gcd(u, w)) = \gcd(u, \text{lcm}(v, w)). \quad (12)$$

The latter two formulas are “distributive laws” analogous to the familiar identity $uv + uw = u(v + w)$. Equation (10) reduces the calculation of $\gcd(u, v)$ to the calculation of $\text{lcm}(u, v)$, and conversely.

Euclid's algorithm. Although Eq. (6) is useful for theoretical purposes, it is generally no help for calculating a greatest common divisor in practice, because it requires that we first determine the factorization of u and v . There is no known method for finding the prime factors of an integer very rapidly (see Section 4.5.4). But fortunately there is an efficient way to calculate the greatest common divisor of two integers without factoring them, and, in fact, such a method was discovered over 2250 years ago; this is “Euclid's algorithm,” which we have already examined in Sections 1.1 and 1.2.1.

Euclid's algorithm is found in Book 7, Propositions 1 and 2 of his *Elements* (c. 300 B.C.), but it probably wasn't his own invention. Scholars believe that the method was known up to 200 years earlier, at least in its subtractive form, and it was almost certainly known to Eudoxus (c. 375 B.C.); cf. K. von Fritz, *Ann. Math.* (2) 46 (1945), 242–264. We might call it the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day. (The chief rival for this honor is perhaps the ancient Egyptian method for multiplication, which was based on doubling and adding, and which forms the basis for efficient calculation of n th powers as explained in Section 4.6.3. But the Egyptian manuscripts merely give examples that are not completely systematic, and these examples were certainly not stated systematically; the Egyptian method is therefore not quite deserving of the name "algorithm." Several ancient Babylonian methods, for doing such things as solving special sets of quadratic equations in two variables, are also known. Genuine algorithms are involved in this case, not just special solutions to the equations for certain input parameters; for even though the Babylonians invariably presented each method in conjunction with an example worked with particular input data, they regularly explained the general procedure in the accompanying text. [See D. E. Knuth, *CACM* 15 (1972), 671–677; 19 (1976), 108.] Many of these Babylonian algorithms predate Euclid by 1500 years, and they are the earliest known instances of written procedures for mathematics. But they do not have the stature of Euclid's algorithm, since they do not involve iteration and since they have been superseded by modern algebraic methods.)

In view of the importance of Euclid's algorithm, for historical as well as practical reasons, let us now consider how Euclid himself treated it. Paraphrasing his words into modern terminology, this is essentially what he wrote:

Proposition. *Given two positive integers, find their greatest common divisor.*

Let A, C be the two given positive integers; it is required to find their greatest common divisor. If C divides A , then C is a common divisor of C and A , since it also divides itself. And it clearly is in fact the greatest, since no greater number than C will divide C .

But if C does not divide A , then continually subtract the lesser of the numbers A, C from the greater, until some number is left that divides the previous one. This will eventually happen, for if unity is left, it will divide the previous number.

Now let E be the positive remainder of A divided by C ; let F be the positive remainder of C divided by E ; and let F be a divisor of E . Since F divides E and E divides $C - F$, F also divides $C - F$; but it also divides itself, so it divides C . And C divides $A - E$; therefore F also divides $A - E$. But it also divides E ; therefore it divides A . Hence it is a common divisor of A and C .

I now claim that it is also the greatest. For if F is not the greatest common divisor of A and C , some larger number will divide them both. Let such a number be G .

Now since G divides C while C divides $A - E$, G divides $A - E$. G also divides the whole of A , so it divides the remainder E . But E divides $C - F$; therefore G also divides $C - F$. And G also divides the whole of C , so it divides the remainder F ; that is, a greater number divides a smaller one. This is impossible.

Therefore no number greater than F will divide A and C , so F is their greatest common divisor.

Corollary. This argument makes it evident that any number dividing two numbers divides their greatest common divisor. *Q.E.D.*

Note. Euclid's statements have been simplified here in one nontrivial respect: Greek mathematicians did not regard unity as a "divisor" of another positive integer. Two positive integers were either both equal to unity, or they were relatively prime, or they had a greatest common divisor. In fact, unity was not even considered to be a "number," and zero was of course nonexistent. These rather awkward conventions made it necessary for Euclid to duplicate much of his discussion, and he gave two separate propositions that are each essentially like the one appearing here.

In his discussion, Euclid first suggests subtracting the smaller of the two current numbers from the larger, repeatedly, until we get two numbers where one is a multiple of the other. But in the proof he really relies on taking the remainder of one number divided by another; and since he has no simple concept of zero, he cannot speak of the remainder when one number divides the other. It is reasonable to say that he imagines each *division* (not the individual subtractions) as a single step of the algorithm, and hence an "authentic" rendition of his algorithm can be phrased as follows:

Algorithm E (Original Euclidean algorithm). Given two integers A and C greater than unity, this algorithm finds their greatest common divisor.

E1. [A divisible by C ?] If C divides A , the algorithm terminates with C as the answer.

E2. [Replace A by remainder.] If $A \bmod C$ is equal to unity, the given numbers were relatively prime, so the algorithm terminates. Otherwise replace the pair of values (A, C) by $(C, A \bmod C)$ and return to step E1. ■

The "proof" Euclid gave, which is quoted above, is especially interesting because it is not really a proof at all! He verifies the result of the algorithm only if step E1 is performed once or thrice. Surely he must have realized that step E1 could take place more than three times, although he made no mention of such a possibility. Not having the notion of a proof by mathematical induction, he could only give a proof for a finite number of cases. (In fact, he often proved only the case $n = 3$ of a theorem that he wanted to establish for general n .) Although Euclid is justly famous for the great advances he made in the art of logical deduction, techniques for giving valid proofs by induction were not discovered until many centuries later, and the crucial ideas for proving the validity of *algorithms* are only now becoming really clear. (See Section 1.2.1 for a complete proof of Euclid's algorithm, together with a short discussion of general proof procedures for algorithms.)

It is worth noting that this algorithm for finding the greatest common divisor was chosen by Euclid to be the very first step in his development of the theory of numbers. The same order of presentation is still in use today in modern

textbooks. Euclid also gave a method (Proposition 34) to find the least common multiple of two integers u and v , namely to divide u by $\gcd(u, v)$ and to multiply the result by v ; this is equivalent to Eq. (10).

If we avoid Euclid's bias against the numbers 0 and 1, we can reformulate Algorithm E in the following way.

Algorithm A (Modern Euclidean algorithm). Given nonnegative integers u and v , this algorithm finds their greatest common divisor. (Note: The greatest common divisor of arbitrary integers u and v may be obtained by applying this algorithm to $|u|$ and $|v|$, because of Eqs. (2) and (3).)

A1. [$v = 0?$] If $v = 0$, the algorithm terminates with u as the answer.

A2. [Take $u \bmod v$.] Set $r \leftarrow u \bmod v$, $u \leftarrow v$, $v \leftarrow r$, and return to A1. (The operations of this step decrease the value of v , but they leave $\gcd(u, v)$ unchanged.) ■

For example, we may calculate $\gcd(40902, 24140)$ as follows:

$$\begin{aligned}\gcd(40902, 24140) &= \gcd(24140, 16762) = \gcd(16762, 7378) \\ &= \gcd(7378, 2006) = \gcd(2006, 1360) = \gcd(1360, 646) \\ &= \gcd(646, 68) = \gcd(68, 34) = \gcd(34, 0) = 34.\end{aligned}$$

A proof that Algorithm A is valid follows readily from Eq. (4) and the fact that

$$\gcd(u, v) = \gcd(v, u - qv), \quad (13)$$

if q is any integer. Equation (13) holds because any common divisor of u and v is a divisor of both v and $u - qv$, and, conversely, any common divisor of v and $u - qv$ must divide both u and v .

The following MIX program illustrates the fact that Algorithm A can easily be implemented on a computer:

Program A (Euclid's algorithm). Assume that u and v are single-precision, nonnegative integers, stored respectively in locations U and V; this program puts $\gcd(u, v)$ into rA.

```

LDX  U    1      rX ← u.
JMP  2F   1
1H  STX  V    T      v ← rX.
      SRAZ 5    T      rAX ← rA.
      DIV   V    T      rX ← rAX mod v.
2H  LDA  V    1 + T  rA ← v.
      JXNZ 1B  1 + T  Done if rX = 0. ■

```

The running time for this program is $19T + 6$ cycles, where T is the number of divisions performed. The discussion in Section 4.5.3 shows that we may take $T = 0.842766 \ln N + 0.06$ as an approximate average value, when u and v are independently and uniformly distributed in the range $1 \leq u, v \leq N$.

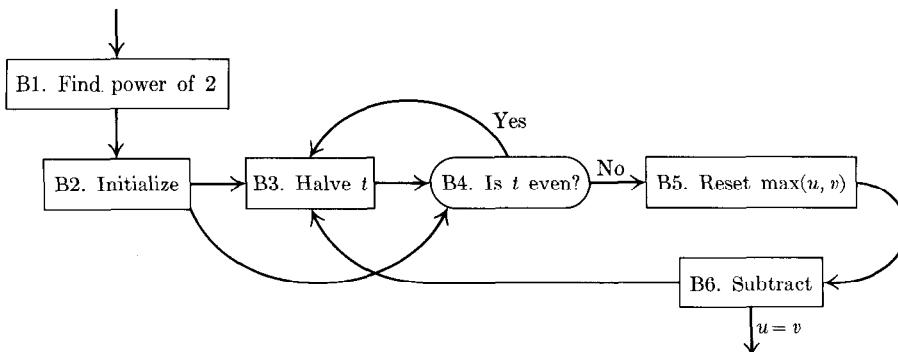


Fig. 9. Binary algorithm for the greatest common divisor.

A binary method. Since Euclid's patriarchal algorithm has been used for so many centuries, it is a rather surprising fact that it may not always be the best method for finding the greatest common divisor after all. A quite different gcd algorithm, which is primarily suited to binary arithmetic, was discovered by J. Stein in 1961 [see *J. Comp. Phys.* **1** (1967), 397–405]. This new algorithm requires no division instruction; it relies solely on the operations of (i) subtraction, (ii) testing whether a number is even or odd, and (iii) shifting the binary representation of an even number to the right (halving).

The binary gcd algorithm is based on four simple facts about positive integers u and v :

- If u and v are both even, then $\gcd(u, v) = 2 \gcd(u/2, v/2)$. [See Eq. (8).]
- If u is even and v is odd, then $\gcd(u, v) = \gcd(u/2, v)$. [See Eq. (6).]
- As in Euclid's algorithm, $\gcd(u, v) = \gcd(u - v, v)$. [See Eqs. (13), (2).]
- If u and v are both odd, then $u - v$ is even, and $|u - v| < \max(u, v)$.

These facts immediately suggest the following algorithm:

Algorithm B (Binary gcd algorithm). Given positive integers u and v , this algorithm finds their greatest common divisor.

- B1. [Find power of 2.] Set $k \leftarrow 0$, and then repeatedly set $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$, zero or more times until u and v are not both even.
- B2. [Initialize.] (Now the original values of u and v have been divided by 2^k , and at least one of their present values is odd.) If u is odd, set $t \leftarrow -v$ and go to B4. Otherwise set $t \leftarrow u$.
- B3. [Halve t .] (At this point, t is even, and nonzero.) Set $t \leftarrow t/2$.
- B4. [Is t even?] If t is even, go back to B3.
- B5. [Reset $\max(u, v)$.] If $t > 0$, set $u \leftarrow t$; otherwise set $v \leftarrow -t$. (The larger of u and v has been replaced by $|t|$, except perhaps during the first time this step is performed.)
- B6. [Subtract.] Set $t \leftarrow u - v$. If $t \neq 0$, go back to B3. Otherwise the algorithm terminates with $u \cdot 2^k$ as the output. ■

As an example of Algorithm B, let us consider $u = 40902$, $v = 24140$, the same numbers we have used to try out Euclid's algorithm. Step B1 sets $k \leftarrow 1$, $u \leftarrow 20451$, $v \leftarrow 12070$. Then t is set to -12070 , and replaced by -6035 ; then v is replaced by 6035 , and the computation proceeds as follows:

u	v	t
20451	6035	+14416, +7208, +3604, +1802, +901;
901	6035	-5134, -2567;
901	2567	-1666, -833;
901	833	+68, +34, +17;
17	833	-816, -408, -204, -102, -51;
17	51	-34, -17;
17	17	0.

The answer is $17 \cdot 2^1 = 34$. A few more iterations were necessary here than we needed with Algorithm A, but each iteration was somewhat simpler since no division steps were used.

A MIX program for Algorithm B requires just a little more code than for Algorithm A. In order to make such a program fairly typical of a binary computer's representation of Algorithm B, let us assume that MIX is extended to include the following operators:

- SLB (shift left AX binary). C = 6; F = 6.

The contents of registers A and X are "shifted left" M binary places; that is, $|rAX| \leftarrow |2^M rAX| \bmod B^{10}$, where B is the byte size. (As with all MIX shift commands, the signs of rA and rX are not affected.)

- SRB (shift right AX binary). C = 6; F = 7.

The contents of registers A and X are "shifted right" M binary places; that is, $|rAX| \leftarrow \lfloor |rAX| / 2^M \rfloor$.

- JAE, JAO (jump A even, jump A odd). C = 40; F = 6, 7, respectively.

A JMP occurs if rA is even or odd, respectively.

- JXE, JXO (jump X even, jump X odd). C = 47; F = 6, 7, respectively.

Analogous to JAE, JAO.

Program B (Binary gcd algorithm). Assume that u and v are single-precision positive integers, stored respectively in locations U and V; this program uses Algorithm B to put $\gcd(u, v)$ into rA. Register assignments: $t \equiv rA$, $k \equiv rI1$.

```

01 ABS EQU 1:5
02 B1 ENT1 0           1      B1. Find power of 2.
03 LDX U              1      rX  $\leftarrow u$ .
04 LDAN V              1      rA  $\leftarrow -v$ .
05 JMP 1F               1
06 2H SRB 1            A      Halve rA, rX.
07 INC1 1               A       $k \leftarrow k + 1$ .
08 STX U               A       $u \leftarrow u/2$ .
09 STA V(ABS)          A       $v \leftarrow v/2$ .

```

10	1H	JXO	B4	$1 + A$	To B4 with $t \leftarrow -v$ if u is odd.
11	B2	JAE	2B	$B + A$	<u>B2. Initialize.</u>
12		LDA	U	B	$t \leftarrow u$.
13	B3	SRB	1	D	<u>B3. Halve t.</u>
14	B4	JAE	B3	$1 - B + D$	<u>B4. Is t even?</u>
15	B5	JAN	1F	C	<u>B5. Reset max(u, v).</u>
16		STA	U	E	If $t > 0$, set $u \leftarrow t$.
17		SUB	V	E	$t \leftarrow u - v$.
18		JMP	2F	E	
19	1H	STA	V(ABS)	$C - E$	If $t < 0$, set $v \leftarrow -t$.
20	B6	ADD	U	$C - E$	<u>B6. Subtract.</u>
21	2H	JANZ	B3	C	To B3 if $t \neq 0$.
22		LDA	U	1	$rA \leftarrow u$.
23		ENTX	0	1	$rX \leftarrow 0$.
24		SLB	0,1	1	$rA \leftarrow 2^k \cdot rA$. ■

The running time of this program is

$$9A + 2B + 6C + 3D + E + 13$$

units, where $A = k$, $B = 1$ if $t \leftarrow u$ in step B2 (otherwise $B = 0$), C is the number of subtraction steps, D is the number of halvings in step B3, and E is the number of times $t > 0$ in step B5. Calculations discussed later in this section imply that we may take $A = \frac{1}{3}$, $B = \frac{1}{3}$, $C = 0.71n - 0.5$, $D = 1.41n - 2.7$, $E = 0.35n - 0.4$ as average values for these quantities, assuming random inputs u and v in the range $1 \leq u, v < 2^n$. The total running time is therefore about $8.8n + 5$ cycles, compared to about $11.1n + 7$ for Program A under the same assumptions. The worst possible running time for u and v in this range occurs when $A = 0$, $B = 0$, $C = n$, $D = 2n - 2$, $E = 1$; this amounts to $12n + 8$ cycles. (The corresponding value for Program A is $26.8n + 19$.)

Thus the greater speed of the iterations in Program B, due to the simplicity of the operations, compensates for the greater number of iterations required. We have found that the binary algorithm is about 20 percent faster than Euclid's algorithm on the MIX computer. Of course, the situation may be different on other computers, and in any event both programs are quite efficient; but it appears that not even a procedure as venerable as Euclid's algorithm can withstand progress.

V. C. Harris [Fibonacci Quarterly 8 (1970), 102–103] has suggested an interesting cross between Euclid's algorithm and the binary algorithm. If u and v are odd, with $u \geq v > 0$, we can always write $u = qv \pm r$ where $0 \leq r < v$ and r is even; if $r \neq 0$ we set $r \leftarrow r/2$ until r is odd, then set $u \leftarrow v$, $v \leftarrow r$ and repeat the process. In subsequent iterations, $q \geq 3$.

Extensions. We can extend the methods used to calculate $\gcd(u, v)$ in order to solve some slightly more difficult problems. For example, assume that we want to compute the greatest common divisor of n integers u_1, u_2, \dots, u_n .

One way to calculate $\gcd(u_1, u_2, \dots, u_n)$, assuming that the u 's are all non-negative, is to extend Euclid's algorithm in the following way: If all u_j are zero,

the greatest common divisor is taken to be zero; otherwise if only one u_j is nonzero, it is the greatest common divisor; otherwise replace u_k by $u_k \bmod u_j$ for all $k \neq j$, where u_j is the minimum of the nonzero u 's.

The algorithm sketched in the preceding paragraph is a natural generalization of Euclid's method, and it can be justified in a similar manner. But there is a simpler method available, based on the easily verified identity

$$\gcd(u_1, u_2, \dots, u_n) = \gcd(u_1, \gcd(u_2, \dots, u_n)). \quad (14)$$

To calculate $\gcd(u_1, u_2, \dots, u_n)$, we may therefore proceed as follows:

C1. Set $d \leftarrow u_n$, $j \leftarrow n - 1$.

C2. If $d \neq 1$ and $j > 0$, set $d \leftarrow \gcd(u_j, d)$ and $j \leftarrow j - 1$ and repeat this step. Otherwise $d = \gcd(u_1, \dots, u_n)$.

This method reduces the calculation of $\gcd(u_1, \dots, u_n)$ to repeated calculations of the greatest common divisor of two numbers at a time. It makes use of the fact that $\gcd(u_1, \dots, u_j, 1) = 1$; and this will be helpful, since we will already have $\gcd(u_{n-1}, u_n) = 1$ over 60 percent of the time if u_{n-1} and u_n are chosen at random. In most cases, the value of d will decrease rapidly during the first few stages of the calculation, and this will make the remainder of the computation quite fast. Here Euclid's algorithm has an advantage over Algorithm B, in that its running time is primarily governed by the value of $\min(u, v)$, while the running time for Algorithm B is primarily governed by $\max(u, v)$; it would be reasonable to perform one iteration of Euclid's algorithm, replacing u by $u \bmod v$ if u is much larger than v , and then to continue with Algorithm B.

The assertion that $\gcd(u_{n-1}, u_n)$ will be equal to unity more than 60 percent of the time for random inputs is a consequence of the following well-known result of number theory:

Theorem D (G. Lejeune Dirichlet, *Abhandlungen Königlich Preuß. Akad. Wiss.* (1849), 69–83). *If u and v are integers chosen at random, the probability that $\gcd(u, v) = 1$ is $6/\pi^2 \approx .60793$.*

A precise formulation of this theorem, which carefully defines what is meant by being “chosen at random,” appears in exercise 10 with a rigorous proof. Let us content ourselves here with a heuristic argument that shows why the theorem is plausible.

If we assume, without proof, the existence of a well-defined probability p that $\gcd(u, v)$ equals unity, then we can determine the probability that $\gcd(u, v) = d$ for any positive integer d , because $\gcd(u, v) = d$ if and only if u is a multiple of d and v is a multiple of d and $\gcd(u/d, v/d) = 1$. Thus the probability that $\gcd(u, v) = d$ is equal to $1/d$ times $1/d$ times p , namely p/d^2 . Now let us sum these probabilities over all possible values of d ; we should get

$$1 = \sum_{d \geq 1} p/d^2 = p\left(1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \dots\right).$$

Since the sum $1 + \frac{1}{4} + \frac{1}{9} + \dots = H_\infty^{(2)}$ is equal to $\pi^2/6$ (cf. Section 1.2.7), we need $p = 6/\pi^2$ in order to make this equation come out right. ■

Euclid's algorithm can be extended in another important way: We can calculate integers u' and v' such that

$$uu' + vv' = \gcd(u, v) \quad (15)$$

at the same time $\gcd(u, v)$ is being calculated. This extension of Euclid's algorithm can be described conveniently in vector notation:

Algorithm X (*Extended Euclid's algorithm*). Given nonnegative integers u and v , this algorithm determines a vector (u_1, u_2, u_3) such that $uu_1 + vu_2 = u_3 = \gcd(u, v)$. The computation makes use of auxiliary vectors $(v_1, v_2, v_3), (t_1, t_2, t_3)$; all vectors are manipulated in such a way that the relations

$$ut_1 + vt_2 = t_3, \quad uu_1 + vu_2 = u_3, \quad uv_1 + vv_2 = v_3 \quad (16)$$

hold throughout the calculation.

X1. [Initialize.] Set $(u_1, u_2, u_3) \leftarrow (1, 0, u)$, $(v_1, v_2, v_3) \leftarrow (0, 1, v)$.

X2. [Is $v_3 = 0$?] If $v_3 = 0$, the algorithm terminates.

X3. [Divide, subtract.] Set $q \leftarrow \lfloor u_3/v_3 \rfloor$, and then set

$$(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3)q, \\ (u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3), \quad (v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3).$$

Return to step X2. ■

For example, let $u = 40902$, $v = 24140$. At step X2 we have

q	u_1	u_2	u_3	v_1	v_2	v_3
—	1	0	40902	0	1	24140
1	0	1	24140	1	-1	16762
1	1	-1	16762	-1	2	7378
2	-1	2	7378	3	-5	2006
3	3	-5	2006	-10	17	1360
1	-10	17	1360	13	-22	646
2	13	-22	646	-36	61	68
9	-36	61	68	337	-571	34
2	337	-571	34	-710	1203	0

The solution is therefore $337 \cdot 40902 - 571 \cdot 24140 = 34 = \gcd(40902, 24140)$.

The validity of Algorithm X follows from (16) and the fact that the algorithm is identical to Algorithm A with respect to its manipulation of u_3 and v_3 ; a detailed proof of Algorithm X is discussed in Section 1.2.1. Gordon H. Bradley has observed that we can avoid a good deal of the calculation in Algorithm X by suppressing u_2 , v_2 , and t_2 ; then u_2 can be determined afterwards using the relation $uu_1 + vu_2 = u_3$.

Exercise 14 shows that the values of $|u_1|, |u_2|, |v_1|, |v_2|$ remain bounded by the size of the inputs u and v . Algorithm B, which computes the greatest common divisor using properties of binary notation, can be extended in a similar way; see exercise 35. For some instructive extensions to Algorithm X, see exercises 18 and 19 in Section 4.6.1.

The ideas underlying Euclid's algorithm can also be applied to find a *general solution in integers* of any set of linear equations with integer coefficients. For example, suppose that we want to find all integers w, x, y, z that satisfy the two equations

$$10w + 3x + 3y + 8z = 1, \quad (17)$$

$$6w - 7x - 5z = 2. \quad (18)$$

We can introduce a new variable

$$\lfloor 10/3 \rfloor w + \lfloor 3/3 \rfloor x + \lfloor 3/3 \rfloor y + \lfloor 8/3 \rfloor z = 3w + x + y + 2z = t_1,$$

and use it to eliminate y ; Eq. (17) becomes

$$(10 \bmod 3)w + (3 \bmod 3)x + 3t_1 + (8 \bmod 3)z = w + 3t_1 + 2z = 1, \quad (19)$$

and Eq. (18) remains unchanged. The new equation (19) may be used to eliminate w , and (18) becomes

$$6(1 - 3t_1 - 2z) - 7x - 5z = 2;$$

that is,

$$7x + 18t_1 + 17z = 4. \quad (20)$$

Now as before we introduce a new variable

$$x + 2t_1 + 2z = t_2$$

and eliminate x from (20):

$$7t_2 + 4t_1 + 3z = 4. \quad (21)$$

Another new variable can be introduced in the same fashion, in order to eliminate the variable z , which has the smallest coefficient:

$$2t_2 + t_1 + z = t_3.$$

Eliminating z from (21) yields

$$t_2 + t_1 + 3t_3 = 4, \quad (22)$$

and this equation, finally, can be used to eliminate t_2 . We are left with two independent variables, t_1 and t_3 ; substituting back for the original variables, we obtain the general solution

$$\begin{aligned} w &= 17 - 5t_1 - 14t_3, \\ x &= 20 - 5t_1 - 17t_3, \\ y &= -55 + 19t_1 + 45t_3, \\ z &= -8 + t_1 + 7t_3. \end{aligned} \quad (23)$$

In other words, all integer solutions (w, x, y, z) to the original equations (17), (18) are obtained from (23) by letting t_1 and t_3 independently run through all integers.

The general method that has just been illustrated is based on the following procedure: Find a nonzero coefficient c of smallest absolute value in the system of equations. Suppose that this coefficient appears in an equation having the form

$$cx_0 + c_1x_1 + \cdots + c_kx_k = d; \quad (24)$$

assume for simplicity that $c > 0$. If $c = 1$, use this equation to eliminate the variable x_0 from the other equations remaining in the system; then repeat the procedure on the remaining equations. (If no more equations remain, the computation stops, and a general solution in terms of the variables not yet eliminated has essentially been obtained.) If $c > 1$, then if $c_1 \bmod c = \cdots = c_k \bmod c = 0$ check that $d \bmod c = 0$, otherwise there is no integer solution; then divide both sides of (24) by c and eliminate x_0 as in the case $c = 1$. Finally, if $c > 1$ and not all of $c_1 \bmod c, \dots, c_k \bmod c$ are zero, then introduce a new variable

$$\lfloor c/c \rfloor x_0 + \lfloor c_1/c \rfloor x_1 + \cdots + \lfloor c_k/c \rfloor x_k = t; \quad (25)$$

eliminate the variable x_0 from the other equations, in favor of t , and replace the original equation (24) by

$$ct + (c_1 \bmod c)x_1 + \cdots + (c_k \bmod c)x_k = d. \quad (26)$$

(Cf. (19) and (21) in the above example.)

This process must terminate, since each step reduces either the number of equations or the size of the smallest nonzero coefficient in the system. A study of the above procedure will reveal its intimate connection with Euclid's algorithm. The method is a comparatively simple means of solving linear equations when the variables are required to take on integer values only. It isn't the best available method for this problem, however; substantial refinements are possible, but beyond the scope of this book.

High-precision calculation. If u and v are very large integers, requiring a multiple-precision representation, the binary method (Algorithm B) is a simple and fairly efficient means of calculating their greatest common divisor, since it involves only subtractions and shifting.

By contrast, Euclid's algorithm seems much less attractive, since step A2 requires a multiple-precision division of u by v . But this difficulty is not really as bad as it seems, since we will prove in Section 4.5.3 that the quotient $\lfloor u/v \rfloor$ is almost always very small; for example, assuming random inputs, the quotient $\lfloor u/v \rfloor$ will be less than 1000 approximately 99.856 percent of the time. Therefore it is almost always possible to find $\lfloor u/v \rfloor$ and $(u \bmod v)$ using single-precision calculations, together with the comparatively simple operation of calculating $u - qv$ where q is a single-precision number. Furthermore, if it does turn out that u is much larger than v (e.g., the initial input data might have this form),

we don't really mind having a large quotient q , since Euclid's algorithm makes a great deal of progress when it replaces u by $u \bmod v$ in such a case.

A significant improvement in the speed of Euclid's algorithm when high-precision numbers are involved can be achieved by using a method due to D. H. Lehmer [AMM 45 (1938), 227–233]. Working only with the leading digits of large numbers, it is possible to do most of the calculations with single-precision arithmetic, and to make a substantial reduction in the number of multiple-precision operations involved. We save a lot of time by doing a "virtual" calculation instead of the actual one.

For example, let us consider the pair of eight-digit numbers $u = 27182818$, $v = 10000000$, assuming that we are using a machine with only four-digit words. Let $u' = 2718$, $v' = 1001$, $u'' = 2719$, $v'' = 1000$; then u'/v' and u''/v'' are approximations to u/v , with

$$u'/v' < u/v < u''/v''. \quad (27)$$

The ratio u/v determines the sequence of quotients obtained in Euclid's algorithm. If we carry out Euclid's algorithm simultaneously on the single-precision values (u', v') and (u'', v'') until we get a different quotient, it is not difficult to see that the same sequence of quotients would have appeared to this point if we had worked with the multiple-precision numbers (u, v) . Thus, consider what happens when Euclid's algorithm is applied to (u', v') and to (u'', v'') :

u'	v'	q'	u''	v''	q''
2718	1001	2	2719	1000	2
1001	716	1	1000	719	1
716	285	2	719	281	2
285	146	1	281	157	1
146	139	1	157	124	1
139	7	19	124	33	3

The first five quotients are the same in both cases, so they must be the true ones. But on the sixth step we find that $q' \neq q''$, so the single-precision calculations are suspended. We have gained the knowledge that the calculation would have proceeded as follows if we had been working with the original multiple-precision numbers:

u	v	q
u_0	v_0	2
v_0	$u_0 - 2v_0$	1
$u_0 - 2v_0$	$-u_0 + 3v_0$	2
$-u_0 + 3v_0$	$3u_0 - 8v_0$	1
$3u_0 - 8v_0$	$-4u_0 + 11v_0$	1
$-4u_0 + 11v_0$	$7u_0 - 19v_0$?

(28)

(The next quotient lies somewhere between 3 and 19.) No matter how many digits are in u and v , the first five steps of Euclid's algorithm would be the same as (28),

so long as (27) holds. We can therefore avoid the multiple-precision operations of the first five steps, and replace them all by a multiple-precision calculation of $-4u_0 + 11v_0$ and $7u_0 - 19v_0$. In this case we obtain $u = 1268728$, $v = 279726$; the calculation can now proceed in a similar manner with $u' = 1268$, $v' = 280$, $u'' = 1269$, $v'' = 279$, etc. If we had a larger accumulator, more steps could be done by single-precision calculations; our example showed that only five cycles of Euclid's algorithm were combined into one multiple step, but with (say) a word size of 10 digits we could do about twelve cycles at a time. (Results proved in Section 4.5.3 imply that the number of multiple-precision cycles that can be replaced at each iteration is essentially proportional to the number of digits used in the single-precision calculations.)

Lehmer's method can be formulated as follows:

Algorithm L (*Euclid's algorithm for large numbers*). Let u, v be nonnegative integers, with $u \geq v$, represented in multiple precision. This algorithm computes the greatest common divisor of u and v , making use of auxiliary single-precision p -digit variables $\hat{u}, \hat{v}, A, B, C, D, T, q$, and auxiliary multiple-precision variables t and w .

L1. [Initialize.] If v is small enough to be represented as a single-precision value, calculate $\gcd(u, v)$ by Algorithm A and terminate the computation. Otherwise, let \hat{u} be the p leading digits of u , and let \hat{v} be the corresponding digits of v ; in other words, if radix- b notation is being used, $\hat{u} \leftarrow \lfloor u/b^k \rfloor$ and $\hat{v} \leftarrow \lfloor v/b^k \rfloor$, where k is as small as possible consistent with the condition $\hat{u} < b^p$.

Set $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$. (These variables represent the coefficients in (28), where

$$u = Au_0 + Bu_0, \quad v = Cu_0 + Du_0, \quad (29)$$

in the equivalent actions of Algorithm A on multiple-precision numbers. We also have

$$u' = \hat{u} + B, \quad v' = \hat{v} + D, \quad u'' = \hat{u} + A, \quad v'' = \hat{v} + C \quad (30)$$

in terms of the notation in the example worked above.)

L2. [Test quotient.] Set $q \leftarrow \lfloor (\hat{u} + A)/(\hat{v} + C) \rfloor$. If $q \neq \lfloor (\hat{u} + B)/(\hat{v} + D) \rfloor$, go to step L4. (This step tests if $q' \neq q''$, in the notation of the above example. Note that single-precision overflow can occur in special circumstances during the computation in this step, but only when $\hat{u} = b^p - 1$ and $A = 1$ or when $\hat{v} = b^p - 1$ and $D = 1$; the conditions

$$\begin{aligned} 0 \leq \hat{u} + A &\leq b^p, & 0 \leq \hat{v} + C &< b^p, \\ 0 \leq \hat{u} + B &< b^p, & 0 \leq \hat{v} + D &\leq b^p \end{aligned} \quad (31)$$

will always hold, because of (30). It is possible to have $\hat{v} + C = 0$ or $\hat{v} + D = 0$, but not both simultaneously; therefore division by zero in this step is taken to mean "Go directly to L4.")

- L3.** [Emulate Euclid.] Set $T \leftarrow A - qC$, $A \leftarrow C$, $C \leftarrow T$, $T \leftarrow B - qD$, $B \leftarrow D$, $D \leftarrow T$, $T \leftarrow \hat{u} - q\hat{v}$, $\hat{u} \leftarrow \hat{v}$, $\hat{v} \leftarrow T$, and go back to step L2. (These single-precision calculations are the equivalent of multiple-precision operations, as in (28), under the conventions of (29).)
- L4.** [Multiprecision step.] If $B = 0$, set $t \leftarrow u \bmod v$, $u \leftarrow v$, $v \leftarrow t$, using multiple-precision division. (This happens only if the single-precision operations cannot simulate any of the multiple-precision ones. It implies that Euclid's algorithm requires a very large quotient, and this is an extremely rare occurrence.) Otherwise, set $t \leftarrow Au$, $t \leftarrow t+Bv$, $w \leftarrow Cu$, $w \leftarrow w+Dv$, $u \leftarrow t$, $v \leftarrow w$ (using straightforward multiple-precision operations). Go back to step L1. ■

The values of A , B , C , D remain as single-precision numbers throughout this calculation, because of (31).

Algorithm L requires a somewhat more complicated program than Algorithm B, but with large numbers it will be faster on many computers. The binary technique of Algorithm B can, however, be speeded up in a similar way (see exercise 34), to the point where it continues to win. Algorithm L has the advantage that it can readily be extended, as in Algorithm X (see exercise 17); furthermore, it determines the sequence of quotients obtained in Euclid's algorithm, and this yields the regular continued fraction expansion of a real number (see exercise 4.5.3-18).

Analysis of the binary algorithm. Let us conclude this section by studying the running time of Algorithm B, in order to justify the formulas stated earlier.

An exact determination of the behavior of Algorithm B appears to be exceedingly difficult to derive, but we can begin to study it by means of an approximate model of its behavior. Suppose that u and v are odd numbers, with $u > v$ and

$$\lfloor \lg u \rfloor = m, \quad \lfloor \lg v \rfloor = n. \quad (32)$$

(Thus, u is an $(m+1)$ -bit number, and v is an $(n+1)$ -bit number.) Algorithm B forms $u - v$ and shifts this quantity right until obtaining an odd number u' that replaces u . Under random conditions, we would expect to have $u' = (u - v)/2$ about one-half of the time, $u' = (u - v)/4$ about one-fourth of the time, $u' = (u - v)/8$ about one-eighth of the time, and so on. We have

$$\lfloor \lg u' \rfloor = m - k - r, \quad (33)$$

where k is the number of places that $u - v$ is shifted right, and where r is $\lfloor \lg u \rfloor - \lfloor \lg(u - v) \rfloor$, the number of bits lost at the left during the subtraction of v from u . Note that $r \leq 1$ when $m \geq n + 2$, and $r \geq 1$ when $m = n$. For simplicity, we will assume that $r = 0$ when $m \neq n$ and that $r = 1$ when $m = n$, although this lower bound tends to make u' seem larger than it usually is.

The approximate model we shall use to study Algorithm B is based solely on the values $m = \lfloor \lg u \rfloor$ and $n = \lfloor \lg v \rfloor$ throughout the course of the algorithm,

not on the actual values of u and v . Let us call this approximation a *lattice-point model*, since we will say that we are “at the point (m, n) ” when $\lfloor \lg u \rfloor = m$ and $\lfloor \lg v \rfloor = n$. From point (m, n) the algorithm takes us to (m', n) if $u > v$, or to (m, n') if $u < v$, or terminates if $u = v$. For example, the calculation starting with $u = 20451$ and $v = 6035$ begins at the point $(14, 12)$, then goes to $(9, 12)$, $(9, 11)$, $(9, 9)$, $(4, 9)$, $(4, 5)$, $(4, 4)$, and terminates. In line with the comments of the preceding paragraph, we will make the following assumptions about the probability that we reach a given point just after point (m, n) :

Case 1, $m > n$.

Next point	Probability
$(m - 1, n)$	$\frac{1}{2}$
$(m - 2, n)$	$\frac{1}{4}$
...	...
$(1, n)$	$(\frac{1}{2})^{m-1}$
$(0, n)$	$(\frac{1}{2})^{m-1}$

Case 2, $m < n$.

Next point	Probability
$(m, n - 1)$	$\frac{1}{2}$
$(m, n - 2)$	$\frac{1}{4}$
...	...
$(m, 1)$	$(\frac{1}{2})^{n-1}$
$(m, 0)$	$(\frac{1}{2})^{n-1}$

Case 3, $m = n > 0$.

Next point	Probability
$(m - 2, n), (m, n - 2)$	$\frac{1}{4}, \frac{1}{4}$
$(m - 3, n), (m, n - 3)$	$\frac{1}{8}, \frac{1}{8}$
...	...
$(0, n), (m, 0)$	$(\frac{1}{2})^m, (\frac{1}{2})^m$
terminate	$(\frac{1}{2})^{m-1}$

For example, from points $(5, 3)$ the lattice-point model would go to points $(4, 3)$, $(3, 3)$, $(2, 3)$, $(1, 3)$, $(0, 3)$ with the respective probabilities $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{16}$; from $(4, 4)$ it would go to $(2, 4)$, $(1, 4)$, $(0, 4)$, $(4, 2)$, $(4, 1)$, $(4, 0)$, or would terminate, with the respective probabilities $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{8}$. When m and n are both 0, the formulas above do not apply; the algorithm always terminates in such a case, since $m = n = 0$ implies that $u = v = 1$.

The detailed calculations of exercise 18 show that this lattice-point model is somewhat pessimistic. In fact, when $m > 3$ the actual probability that Algorithm B goes from (m, m) to one of the two points $(m - 2, m)$ or $(m, m - 2)$ is equal to $\frac{1}{8}$, although we have assumed the value $\frac{1}{2}$; the algorithm actually goes from (m, m) to $(m - 3, m)$ or $(m, m - 3)$ with probability $\frac{7}{32}$, not $\frac{1}{4}$; and it actually goes from $(m + 1, m)$ to (m, m) with probability $\frac{1}{8}$, not $\frac{1}{2}$. The probabilities in the model are nearly correct when $|m - n|$ is large, but when $|m - n|$ is small the model predicts slower convergence than is actually obtained. In spite of the fact that our model is not a completely faithful representation of Algorithm B, it has one great virtue: It can be completely analyzed! Furthermore, empirical experiments with Algorithm B show that the behavior predicted by the lattice-point model is analogous to the true behavior.

An analysis of the lattice-point model can be carried out by solving the following rather complicated set of double recurrence relations:

$$\begin{aligned} A_{mm} &= a + \frac{1}{2}A_{m(m-2)} + \cdots + \frac{1}{2^{m-1}}A_{m0} + \frac{b}{2^{m-1}}, \quad \text{if } m \geq 1; \\ A_{mn} &= c + \frac{1}{2}A_{(m-1)n} + \cdots + \frac{1}{2^{m-1}}A_{1n} + \frac{1}{2^{m-1}}A_{0n}, \quad \text{if } m > n \geq 0; \\ A_{mn} &= A_{nm}, \quad \text{if } n > m \geq 0. \end{aligned} \quad (34)$$

The problem is to solve for A_{mn} in terms of m , n , and the parameters a , b , c , and A_{00} . This is an interesting set of recurrence equations, which have an interesting solution.

First we observe that if $0 \leq n < m$,

$$\begin{aligned} A_{(m+1)n} &= c + \sum_{1 \leq k \leq m} 2^{-k} A_{(m+1-k)n} + 2^{-m} A_{0n} \\ &= c + \frac{1}{2}A_{mn} + \sum_{1 \leq k < m} 2^{-k-1} A_{(m-k)n} + 2^{-m} A_{0n} \\ &= c + \frac{1}{2}A_{mn} + \frac{1}{2}(A_{mn} - c) \\ &= \frac{1}{2}c + A_{mn}. \end{aligned}$$

Hence $A_{(m+k)n} = \frac{1}{2}ck + A_{mn}$, by induction on k . In particular, since $A_{10} = c + A_{00}$, we have

$$A_{m0} = \frac{1}{2}c(m+1) + A_{00}, \quad m > 0. \quad (35)$$

Now let $A_m = A_{mm}$. If $m > 0$, we have

$$\begin{aligned} A_{(m+1)m} &= c + \sum_{1 \leq k \leq m+1} 2^{-k} A_{(m+1-k)m} + 2^{-m-1} A_{0m} \\ &= c + \frac{1}{2}A_{mm} + \sum_{1 \leq k \leq m} (2^{-k-1}(A_{(m-k)(m+1)} - c/2)) + 2^{-m-1} A_{0m} \\ &= c + \frac{1}{2}A_m + \frac{1}{2}(A_{(m+1)(m+1)} - a - 2^{-m}b) - \frac{1}{4}c(1 - 2^{-m}) \\ &\quad + 2^{-m-1}(\frac{1}{2}c(m+1) + A_{00}) \\ &= \frac{1}{2}(A_m + A_{m+1}) + \frac{3}{4}c - \frac{1}{2}a + 2^{-m-1}(c - b + A_{00}) + m2^{-m-2}c. \end{aligned} \quad (36)$$

Similar maneuvering, as shown in exercise 19, establishes the relation

$$A_{n+1} = \frac{3}{4}A_n + \frac{1}{4}A_{n-1} + \alpha + 2^{-n-1}\beta + (n+2)2^{-n-1}\gamma, \quad n \geq 2, \quad (37)$$

where $\alpha = \frac{1}{4}a + \frac{7}{8}c$, $\beta = A_{00} - b - \frac{3}{2}c$, and $\gamma = \frac{1}{2}c$.

Thus the double recurrence (34) can be transformed into the single recurrence relation in (37). Use of the generating function $G(z) = A_0 + A_1 z + A_2 z^2 + \dots$ now transforms (37) into the equation

$$(1 - \frac{3}{4}z - \frac{1}{4}z^2)G(z) = a_0 + a_1 z + a_2 z^2 + \frac{\alpha}{1-z} + \frac{\beta}{1-z/2} + \frac{\gamma}{(1-z/2)^2}, \quad (38)$$

where a_0 , a_1 , and a_2 are constants that can be determined by the values of A_0 , A_1 , and A_2 . Since $1 - \frac{3}{4}z + \frac{1}{4}z^2 = (1 + \frac{1}{4}z)(1 - z)$, we can express $G(z)$ by the method of partial fractions in the form

$$G(z) = b_0 + b_1 z + \frac{b_2}{(1-z)^2} + \frac{b_3}{1-z} + \frac{b_4}{(1-z/2)^2} + \frac{b_5}{1-z/2} + \frac{b_6}{1+z/4}.$$

Tedious manipulations produce the values of these constants b_0, \dots, b_6 , and thus the coefficients of $G(z)$ are determined. We finally obtain the solution

$$\begin{aligned} A_{nn} &= n(\frac{1}{5}a + \frac{7}{10}c) + (\frac{16}{25}a + \frac{2}{5}b - \frac{23}{50}c + \frac{3}{5}A_{00}) \\ &\quad + 2^{-n}(-\frac{1}{3}cn + \frac{2}{3}b - \frac{1}{6}c - \frac{2}{3}A_{00}) \\ &\quad + (-\frac{1}{4})^n(-\frac{16}{25}a - \frac{16}{15}b + \frac{16}{225}c + \frac{16}{15}A_{00}) + \frac{1}{2}c\delta_{n0}; \\ A_{mn} &= \frac{1}{2}mc + n(\frac{1}{5}a + \frac{1}{5}c) + (\frac{6}{25}a + \frac{2}{5}b + \frac{7}{50}c + \frac{3}{5}A_{00}) + 2^{-n}(\frac{1}{3}c) \\ &\quad + (-\frac{1}{4})^n(-\frac{6}{25}a - \frac{2}{5}b + \frac{2}{5}c + \frac{2}{5}A_{00}), \quad m > n. \end{aligned} \quad (39)$$

With these elaborate calculations done, we can readily determine the behavior of the lattice-point model. Assume that the inputs u and v to the algorithm are odd, and let $m = \lfloor \lg u \rfloor$, $n = \lfloor \lg v \rfloor$. The average number of subtraction cycles, namely the quantity C in the analysis of Program B, is obtained by setting $a = 1$, $b = 0$, $c = 1$, $A_{00} = 1$ in the recurrence (34). By (39) we see that (for $m \geq n$) the lattice model predicts

$$C = \frac{1}{2}m + \frac{2}{5}n + \frac{49}{50} - \frac{1}{5}\delta_{mn} \quad (40)$$

subtraction cycles, plus terms that rapidly go to zero as n approaches infinity.

The average number of times that $\gcd(u, v) = 1$ is obtained by setting $a = b = c = 0$, $A_{00} = 1$; this gives the probability that u and v are relatively prime, approximately $\frac{3}{5}$. Actually, since u and v are assumed to be odd, they should be relatively prime with probability $8/\pi^2$ (see exercise 13), so this reflects the degree of inaccuracy of the lattice-point model.

The average number of times that a path from (m, n) goes through one of the “diagonal” points (m', m') for some $m' \geq 1$ is obtained by setting $a = 1$, $b = c = A_{00} = 0$ in (34); so we find that this quantity is approximately

$$\frac{1}{5}n + \frac{6}{25} + \frac{2}{5}\delta_{mn}, \quad \text{when } m \geq n.$$

Now we can determine the average number of shifts, i.e., the number of times that step B3 is performed. (This is the quantity D in Program B.) In any

Table 1
NUMBER OF SUBTRACTIONS IN ALGORITHM B

	<i>n</i>							<i>n</i>						
	0	1	2	3	4	5		0	1	2	3	4	5	
0	1.00	2.00	2.50	3.00	3.50	4.00		1.00	2.00	2.50	3.00	3.50	4.00	0
1	2.00	1.00	2.50	3.00	3.50	4.00		2.00	1.00	3.00	2.75	3.63	3.94	1
2	2.50	2.50	2.25	3.38	3.88	4.38		2.50	3.00	2.00	3.50	3.88	4.25	2
3	3.00	3.00	3.38	3.25	4.22	4.72		3.00	2.75	3.50	2.88	4.13	4.34	3
4	3.50	3.50	3.88	4.22	4.25	5.10		3.50	3.63	3.88	4.13	3.94	4.80	4
5	4.00	4.00	4.38	4.72	5.10	5.19		4.00	3.94	4.25	4.34	4.80	4.60	5
<i>m</i>	Predicted by model							Actual average values						<i>m</i>

execution of Algorithm B, with u and v both odd, the corresponding path in the lattice model must satisfy the relation

$$\text{number of shifts} + \text{number of diagonal points} + 2\lfloor \lg \gcd(u, v) \rfloor = m + n,$$

since we are assuming that r in (33) is always either 0 or 1. The average value of $\lfloor \lg \gcd(u, v) \rfloor$ predicted by the lattice-point model is approximately $\frac{4}{5}$ (see exercise 20). Therefore we have, for the total number of shifts,

$$\begin{aligned} D &= m + n - \left(\frac{1}{5}n + \frac{6}{25} + \frac{2}{5}\delta_{mn}\right) - \frac{4}{5} \\ &= m + \frac{4}{5}n - \frac{46}{25} - \frac{2}{5}\delta_{mn}, \end{aligned} \quad (41)$$

when $m \geq n$, plus terms that decrease rapidly to zero for large n .

To summarize the most important facts we have derived from the lattice-point model, we have shown that if u and v are odd and if $\lfloor \lg u \rfloor = m$, $\lfloor \lg v \rfloor = n$, then the quantities C and D that are the critical factors in the running time of Program B will have average values given by

$$C = \frac{1}{2}m + \frac{2}{5}n + O(1), \quad D = m + \frac{4}{5}n + O(1), \quad m \geq n. \quad (42)$$

But the model that we have used to derive (42) is only a pessimistic approximation to the true behavior; Table 1 compares the true average values of C , computed by actually running Algorithm B with all possible inputs, to the values predicted by the lattice-point model, for small m and n . The lattice model is completely accurate when m or n is zero, but it tends to be less accurate when $|m - n|$ is small and $\min(m, n)$ is large. When $m = n = 9$, the lattice-point model gives $C = 8.78$, compared to the true value $C = 7.58$.

Empirical tests of Algorithm B with several million random inputs and with various values of m, n in the range $29 \leq m, n \leq 37$ indicate that the actual average behavior of the algorithm is given by

$$\begin{aligned} C &\approx \frac{1}{2}m + 0.203n + 1.9 - 0.4(0.6)^{m-n}, \\ D &\approx m + 0.41n - 0.5 - 0.7(0.6)^{m-n}, \end{aligned} \quad m \geq n. \quad (43)$$

These experiments showed a rather small standard deviation from the observed average values. The coefficients $\frac{1}{2}$ and 1 of m in (42) and (43) can be verified rigorously without using the lattice-point approximation (see exercise 21); so the error in the lattice-point model is apparently in the coefficient of n , which is too high.

The above calculations have been made under the assumption that u and v are odd and in the ranges $2^m \leq u < 2^{m+1}$ and $2^n \leq v < 2^{n+1}$. If we assume instead that u and v are to be any integers, independently and uniformly distributed over the ranges

$$1 \leq u < 2^N, \quad 1 \leq v < 2^N,$$

then we can calculate the average values of C and D from the data already given; in fact, if C_{mn} denotes the average value of C under our earlier assumptions, exercise 22 shows that we have

$$\begin{aligned} (2^N - 1)^2 C = N^2 C_{00} + 2N \sum_{1 \leq n \leq N} (N - n) 2^{n-1} C_{n0} \\ + 2 \sum_{1 \leq n < m \leq N} (N - m)(N - n) 2^{m+n-2} C_{mn} \\ + \sum_{1 \leq n \leq N} (N - n)^2 2^{2n-2} C_{nn}. \end{aligned} \tag{44}$$

The same formula holds for D in terms of D_{mn} . If the indicated sums are carried out using the approximations in (43), we obtain

$$C \approx 0.70N + O(1), \quad D \approx 1.41N + O(1).$$

(See exercise 23.) This agrees perfectly with the results of further empirical tests, made on several million random inputs for $N \leq 30$; the latter tests show that we may take

$$C = 0.70N - 0.5, \quad D = 1.41N - 2.7 \tag{45}$$

as good estimates of the values, given this distribution of the inputs u and v .

Richard Brent has discovered a continuous model that accounts for the leading terms in (45). Let us assume that u and v are large, and that the number of shifts per iteration has the value d with probability exactly 2^{-d} . If we let $X = u/v$, the effect of steps B3–B5 is to replace X by $(X - 1)/2^d$ if $X > 1$, or by $2^d/(X^{-1} - 1)$ if $X < 1$. The random variable X has a limiting distribution that makes it possible to analyze the average value of the ratio by which $\max(u, v)$ decreases at each iteration; see exercise 25. Numerical calculations show that this maximum decreases by $\beta = 0.705971246102$ bits per iteration; the agreement with experiment is so good that Brent's constant β must be the true value of the number "0.70" in (45), and we should replace 0.203 by 0.206 in (43). [See *Algorithms and Complexity*, ed. by J. F. Traub (New York: Academic Press, 1976), 321–355.]

This completes our analysis of the average values of C and D . The other three quantities appearing in the running time of Algorithm B are rather easily analyzed; see exercises 6, 7, and 8.

Thus we know approximately how Algorithm B behaves on the average. Let us now consider a “worst case” analysis: What values of u and v are in some sense the hardest to handle? If we assume as before that

$$\lfloor \lg u \rfloor = m \quad \text{and} \quad \lfloor \lg v \rfloor = n,$$

let us try to find (u, v) that make the algorithm run most slowly. In view of the fact that the subtractions take somewhat longer than the shifts, when the auxiliary bookkeeping is considered, this question may be rephrased by asking for the inputs u and v that require most subtractions. The answer is somewhat surprising; the maximum value of C is exactly

$$\max(m, n) + 1, \tag{46}$$

although the lattice-point model would predict that substantially higher values of C are possible (see exercise 26). The derivation of the worst case (46) is quite interesting, so it has been left as an amusing problem for the reader to work out by himself (see exercises 27 and 28).

EXERCISES

1. [M21] How can (8), (9), (10), (11), and (12) be derived easily from (6) and (7)?

2. [M22] Given that u divides $v_1 v_2 \dots v_n$, prove that u divides

$$\gcd(u, v_1) \gcd(u, v_2) \dots \gcd(u, v_n).$$

3. [M23] Show that the number of ordered pairs of positive integers (u, v) such that $\text{lcm}(u, v) = n$ is the number of divisors of n^2 .

4. [M21] Given positive integers u and v , show that there are divisors u' of u and v' of v such that $\gcd(u', v') = 1$ and $u'v' = \text{lcm}(u, v)$.

► 5. [M26] Invent an algorithm (analogous to Algorithm B) for calculating the greatest common divisor of two integers based on their *balanced ternary* representation. Demonstrate your algorithm by applying it to the calculation of $\gcd(40902, 24140)$.

6. [M22] Given that u and v are random positive integers, find the mean and the standard deviation of the quantity A that enters into the timing of Program B. (This is the number of right shifts applied to both u and v during the preparatory phase.)

7. [M20] Analyze the quantity B that enters into the timing of Program B.

► 8. [M25] Show that in Program B, the average value of E is approximately equal to $\frac{1}{2}C_{\text{ave}}$, where C_{ave} is the average value of C .

9. [18] Using Algorithm B and hand calculation, find $\gcd(31408, 2718)$. Also find integers m and n such that $31408m + 2718n = \gcd(31408, 2718)$, using Algorithm X.

► 10. [HM24] Let q_n be the number of ordered pairs of integers (u, v) lying in the range $1 \leq u, v \leq n$ such that $\gcd(u, v) = 1$. The object of this exercise is to prove that we have $\lim_{n \rightarrow \infty} q_n/n^2 = 6/\pi^2$, thereby establishing Theorem D.

a) Use the principle of inclusion and exclusion (Section 1.3.3) to show that

$$q_n = n^2 - \sum_{p_1} \lfloor n/p_1 \rfloor^2 + \sum_{p_1 < p_2} \lfloor n/p_1 p_2 \rfloor^2 - \dots,$$

where the sums are taken over all prime numbers p_i .

- b) The Möbius function $\mu(n)$ is defined by the rules $\mu(1) = 1$, $\mu(p_1 p_2 \dots p_r) = (-1)^r$ if p_1, p_2, \dots, p_r are distinct primes, and $\mu(n) = 0$ if n is divisible by the square of a prime. Show that $q_n = \sum_{k \geq 1} \mu(k) \lfloor n/k \rfloor^2$.
- c) As a consequence of (b), prove that $\lim_{n \rightarrow \infty} q_n/n^2 = \sum_{k \geq 1} \mu(k)/k^2$.
- d) Prove that $(\sum_{k \geq 1} \mu(k)/k^2)(\sum_{m \geq 1} 1/m^2) = 1$. Hint: When the series are absolutely convergent we have

$$\left(\sum_{k \geq 1} a_k/k^z \right) \left(\sum_{m \geq 1} b_m/m^z \right) = \sum_{n \geq 1} \left(\sum_{d|n} a_d b_{n/d} \right) / n^z.$$

11. [M22] What is the probability that $\gcd(u, v) \leq 3$? (See Theorem D.) What is the average value of $\gcd(u, v)$?

12. [M24] (E. Cesàro.) If u and v are random positive integers, what is the average number of (positive) divisors they have in common? [Hint: See the identity in exercise 10(d), with $a_k = b_m = 1$.]

13. [HM23] Given that u and v are random odd positive integers, show that they are relatively prime with probability $8/\pi^2$.

14. [M26] What are the values of v_1 and v_2 when Algorithm X terminates?

► 15. [M22] Design an algorithm to divide u by v modulo m , given positive integers u , v , and m , with v relatively prime to m . In other words, your algorithm should find w , in the range $0 \leq w < m$, such that $u \equiv vw$ (modulo m).

16. [21] Use the text's method to find a general solution in integers to the following sets of equations:

a) $3x + 7y + 11z = 1$	b) $3x + 7y + 11z = 1$
$5x + 7y - 5z = 3$	$5x + 7y - 5z = -3$

► 17. [M24] Show how Algorithm L can be extended (as Algorithm A was extended to Algorithm X) to obtain solutions of (15) when u and v are large.

18. [M37] Let u and v be odd integers, independently and uniformly distributed in the ranges $2^m \leq u < 2^{m+1}$, $2^n \leq v < 2^{n+1}$. What is the exact probability that a single “subtract and shift” cycle in Algorithm B, namely an operation that starts at step B6 and then stops after step B5 is finished, reduces u and v to the ranges $2^{m'} \leq u < 2^{m'+1}$, $2^{n'} \leq v < 2^{n'+1}$, as a function of m , n , m' , and n' ? (This exercise gives more accurate values for the transition probabilities than the text's model does.)

19. [M24] Complete the text's derivation of (38) by establishing (37).

20. [M26] Let $\lambda = \lfloor \lg \gcd(u, v) \rfloor$. Show that the lattice-point model gives $\lambda = 1$ with probability $\frac{1}{5}$, $\lambda = 2$ with probability $\frac{1}{10}$, $\lambda = 3$ with probability $\frac{1}{20}$, etc., plus correction terms that go rapidly to zero as u and v approach infinity; hence the average value of λ is approximately $\frac{4}{3}$. [Hint: Consider the relation between the probability of a path from (m, n) to $(k+1, k+1)$ and a corresponding path from $(m-k, n-k)$ to $(1, 1)$.]

21. [HM26] Let C_{mn} and D_{mn} be the average number of subtraction and shift cycles, respectively, in Algorithm B, when u and v are odd, $\lfloor \lg u \rfloor = m$, $\lfloor \lg v \rfloor = n$. Show that for fixed n , $C_{mn} = \frac{1}{2}m + O(1)$ and $D_{mn} = m + O(1)$ as $m \rightarrow \infty$.

22. [23] Prove Eq. (44).

23. [M28] Show that if $C_{mn} = \alpha m + \beta n + \gamma$ for some constants α , β , and γ , then

$$\sum_{1 \leq n \leq m \leq N} (N-m)(N-n)2^{m+n-2}C_{mn} = 2^{2N}(\frac{11}{27}(\alpha+\beta)N + O(1)),$$

$$\sum_{1 \leq n \leq N} (N-n)^2 2^{2n-2}C_{nn} = 2^{2N}(\frac{5}{27}(\alpha+\beta)N + O(1)).$$

► **24.** [M30] If $v = 1$ but u is large, during Algorithm B, it may take fairly long for the algorithm to determine that $\gcd(u, v) = 1$. Perhaps it would be worthwhile to add a test at the beginning of step B5: "If $t = 1$, the algorithm terminates with 2^k as the answer." Explore the question of whether or not this would be an improvement when the algorithm deals with random inputs, by determining the average number of times that step B6 is executed with $u = 1$ or $v = 1$, using the lattice-point model.

► **25.** [M26] (R. P. Brent.) Let u_n and v_n be the values of u and v after n iterations of steps B3–B5; let $X_n = u_n/v_n$, and assume that $F_n(x)$ is the probability that $X_n \leq x$, for $0 \leq x < \infty$. (a) Express $F_{n+1}(x)$ in terms of $F_n(x)$, under the assumption that step B4 always branches to B3 with independent probability $\frac{1}{2}$. (b) Let $G_n(x) = F_n(x) + 1 - F_n(x^{-1})$ be the probability that $Y_n \leq x$, for $0 \leq x \leq 1$, where $Y_n = \min(u_n, v_n)/\max(u_n, v_n)$. Express G_{n+1} in terms of G_n . (c) Express the distribution $H_n(x) = \text{probability that } \max(u_{n+1}, v_{n+1})/\max(u_n, v_n) < x$ in terms of G_n .

26. [M23] What is the length of the longest path from (m, n) to $(0, 0)$ in the lattice-point model?

► **27.** [M28] Given $m \geq n \geq 1$, find values of u , v with $\lfloor \lg u \rfloor = m$, $\lfloor \lg v \rfloor = n$ such that Algorithm B requires $m+1$ subtraction steps.

28. [M37] Prove that the subtraction step B6 of Algorithm B is never executed more than $1 + \lfloor \lg \max(u, v) \rfloor$ times.

29. [M30] Evaluate the determinant

$$\begin{vmatrix} \gcd(1, 1) & \gcd(1, 2) & \dots & \gcd(1, n) \\ \gcd(2, 1) & \gcd(2, 2) & \dots & \gcd(2, n) \\ \vdots & \vdots & & \vdots \\ \gcd(n, 1) & \gcd(n, 2) & \dots & \gcd(n, n) \end{vmatrix}.$$

30. [M25] Show that Euclid's algorithm (Algorithm A) applied to two n -bit binary numbers requires $O(n^2)$ units of time, as $n \rightarrow \infty$. (The same upper bound obviously holds for Algorithm B.)

31. [M22] Use Euclid's algorithm to find a simple formula for $\gcd(2^m - 1, 2^n - 1)$ when m and n are nonnegative integers.
32. [M43] Can the upper bound $O(n^2)$ in exercise 30 be decreased, if another algorithm for calculating the greatest common divisor is used?
33. [M46] Analyze V. C. Harris's "binary Euclidean algorithm."
- 34. [M32] (R. W. Gosper.) Demonstrate how to modify Algorithm B for large numbers, using ideas analogous to those in Algorithm L.
- 35. [M28] (V. R. Pratt.) Extend Algorithm B to an Algorithm Y that is analogous to Algorithm X.
36. [HM49] Find a rigorous proof that Brent's model describes the asymptotic behavior of Algorithm B.

*4.5.3. Analysis of Euclid's Algorithm

The execution time of Euclid's algorithm depends on T , the number of times the division step A2 is performed. (See Algorithm 4.5.2A and Program 4.5.2A.) The quantity T is also an important factor in the running time of other algorithms, such as the evaluation of functions satisfying a reciprocity formula (see Section 3.3.3). We shall see in this section that the mathematical analysis of this quantity T is interesting and instructive.

Relation to continued fractions. Euclid's algorithm is intimately connected with *continued fractions*, which are expressions of the form

$$\cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{\dots + \cfrac{b_n}{a_{n-1} + \cfrac{b_n}{a_n}}}}} = b_1 / (a_1 + b_2 / (a_2 + b_3 / (\dots / (a_{n-1} + b_n / a_n) \dots))). \quad (1)$$

Continued fractions have a beautiful theory that is the subject of several books. [See, for example, O. Perron, *Die Lehre von den Kettenbrüchen*, 3rd ed. (Stuttgart: Teubner, 1954), 2 vols.; A. Khinchin, *Continued Fractions*, tr. by Peter Wynn (Groningen: P. Noordhoff, 1963); H. S. Wall, *Analytic Theory of Continued Fractions* (New York: Van Nostrand, 1948); and see also J. Tropfke, *Geschichte der Elementar-Mathematik* 6 (Berlin: Gruyter, 1924), 74–84, for the early history of the subject.] It is necessary to limit ourselves to a comparatively brief treatment of the theory here, studying only those aspects that give us more insight into the behavior of Euclid's algorithm.

The continued fractions of primary interest to us are those in which all of the b 's in (1) are equal to unity. For convenience in notation, let us define

$$[x_1, x_2, \dots, x_n] = 1 / (x_1 + 1 / (x_2 + 1 / (\dots + (x_{n-1} + 1 / x_n) \dots))). \quad (2)$$

Thus, for example,

$$|x_1| = \frac{1}{x_1}, \quad |x_1, x_2| = \frac{1}{x_1 + 1/x_2} = \frac{x_2}{x_1 x_2 + 1}. \quad (3)$$

If $n = 0$, the symbol $|x_1, \dots, x_n|$ is taken to mean 0. Let us also define the polynomials $Q_n(x_1, x_2, \dots, x_n)$ of n variables, for $n \geq 0$, by the rule

$$Q_n(x_1, x_2, \dots, x_n) = \begin{cases} 1, & \text{if } n = 0; \\ x_1, & \text{if } n = 1; \\ x_1 Q_{n-1}(x_2, \dots, x_n) + Q_{n-2}(x_3, \dots, x_n), & \text{if } n > 1. \end{cases} \quad (4)$$

Thus $Q_2(x_1, x_2) = x_1 x_2 + 1$, $Q_3(x_1, x_2, x_3) = x_1 x_2 x_3 + x_1 + x_3$, etc. In general, as noted by L. Euler in the eighteenth century, $Q_n(x_1, x_2, \dots, x_n)$ is the sum of all terms obtainable by starting with $x_1 x_2 \dots x_n$ and deleting zero or more non-overlapping pairs of consecutive variables $x_j x_{j+1}$; there are F_{n+1} such terms. The polynomials defined in (4) are called "continuants."

The basic property of the Q -polynomials is that

$$|x_1, x_2, \dots, x_n| = Q_{n-1}(x_2, \dots, x_n)/Q_n(x_1, x_2, \dots, x_n), \quad n \geq 1. \quad (5)$$

This can be proved by induction, since it implies that

$$x_0 + |x_1, \dots, x_n| = Q_{n+1}(x_0, x_1, \dots, x_n)/Q_n(x_1, \dots, x_n);$$

hence $|x_0, x_1, \dots, x_n|$ is the reciprocal of the latter quantity.

The Q -polynomials are symmetrical in the sense that

$$Q_n(x_1, x_2, \dots, x_n) = Q_n(x_n, \dots, x_2, x_1). \quad (6)$$

This follows from Euler's observation above, and as a consequence we have

$$Q_n(x_1, \dots, x_n) = x_n Q_{n-1}(x_1, \dots, x_{n-1}) + Q_{n-2}(x_1, \dots, x_{n-2}) \quad (7)$$

for $n > 1$. The Q -polynomials also satisfy the important identity

$$\begin{aligned} Q_n(x_1, \dots, x_n) Q_n(x_2, \dots, x_{n+1}) - Q_{n+1}(x_1, \dots, x_{n+1}) Q_{n-1}(x_2, \dots, x_n) \\ = (-1)^n, \quad n \geq 1. \end{aligned} \quad (8)$$

(See exercise 4.) The latter equation in connection with (5) implies that

$$\begin{aligned} |x_1, \dots, x_n| &= \frac{1}{q_0 q_1} - \frac{1}{q_1 q_2} + \frac{1}{q_2 q_3} - \dots + \frac{(-1)^{n-1}}{q_{n-1} q_n}, \\ &\text{where } q_k = Q_k(x_1, \dots, x_k). \end{aligned} \quad (9)$$

Thus the Q -polynomials are intimately related to continued fractions.

Every real number X in the range $0 \leq X < 1$ has a *regular continued fraction* defined as follows: Let $X_0 = X$, and for all $n \geq 0$ such that $X_n \neq 0$ let

$$A_{n+1} = \lfloor 1/X_n \rfloor, \quad X_{n+1} = 1/X_n - A_{n+1}. \quad (10)$$

If $X_n = 0$, the quantities A_{n+1} and X_{n+1} are not defined, and the regular continued fraction for X is $|A_1, \dots, A_n|$. If $X_n \neq 0$, this definition guarantees that $0 \leq X_{n+1} < 1$, so each of the A 's is a positive integer. The definition (10) clearly implies that

$$X = X_0 = \frac{1}{A_1 + X_1} = \frac{1}{A_1 + 1/(A_2 + X_2)} = \dots;$$

hence

$$X = |A_1, \dots, A_{n-1}, A_n + X_n| \quad (11)$$

for all $n \geq 1$, whenever X_n is defined. In particular, we have $X = |A_1, \dots, A_n|$ when $X_n = 0$. If $X_n \neq 0$, the number X always lies between $|A_1, \dots, A_n|$ and $|A_1, \dots, A_n + 1|$, since by (7) the quantity $q_n = Q_n(A_1, \dots, A_n + X_n)$ increases monotonically from $Q_n(A_1, \dots, A_n)$ up to $Q_n(A_1, \dots, A_n + 1)$ as X_n increases from 0 to 1, and by (9) the continued fraction increases or decreases when q_n increases, according as n is even or odd. In fact,

$$\begin{aligned} |X - |A_1, \dots, A_n|| &= ||A_1, \dots, A_n + X_n| - |A_1, \dots, A_n|| \\ &= ||A_1, \dots, A_n, 1/X_n| - |A_1, \dots, A_n|| \\ &= \left| \frac{Q_n(A_2, \dots, A_n, 1/X_n)}{Q_{n+1}(A_1, \dots, A_n, 1/X_n)} - \frac{Q_{n-1}(A_2, \dots, A_n)}{Q_n(A_1, \dots, A_n)} \right| \\ &= 1/(Q_n(A_1, \dots, A_n)Q_{n+1}(A_1, \dots, A_n, 1/X_n)) \\ &\leq 1/(Q_n(A_1, \dots, A_n)Q_{n+1}(A_1, \dots, A_n, A_{n+1})) \end{aligned} \quad (12)$$

by (5), (7), (8), and (10). Therefore $|A_1, \dots, A_n|$ is an extremely close approximation to X , unless n is small. If X is irrational, it is impossible to have $X_n = 0$ for any n , so the regular continued fraction expansion in this case is an *infinite continued fraction* $|A_1, A_2, A_3, \dots|$. The value of an infinite continued fraction is defined to be

$$\lim_{n \rightarrow \infty} |A_1, A_2, \dots, A_n|,$$

and from the inequality (12) it is clear that this limit equals X .

The regular continued fraction expansion of real numbers has several properties analogous to the representation of numbers in the decimal system. If we use the formulas above to compute the regular continued fraction expansions of

some familiar real numbers, we find, for example, that

$$\begin{aligned}\frac{8}{29} &= [3, 1, 1, 1, 2]; \\ \sqrt{\frac{8}{29}} &= [1, 1, 9, 2, 2, 3, 2, 2, 9, 1, 2, 1, 9, 2, 2, 3, 2, 2, 9, 1, 2, 1, 9, 2, 2, 3, 2, 2, 9, 1, \dots]; \\ \sqrt[3]{2} &= 1 + [3, 1, 5, 1, 1, 4, 1, 1, 8, 1, 14, 1, 10, 2, 1, 4, 12, 2, 3, 2, 1, 3, 4, 1, 1, 2, 14, 3, \dots]; \\ \pi &= 3 + [7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3, 13, \dots]; \\ e &= 2 + [1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1, 16, 1, 1, 18, 1, \dots]; \\ \gamma &= [1, 1, 2, 1, 2, 1, 4, 3, 13, 5, 1, 1, 8, 1, 2, 4, 1, 1, 40, 1, 11, 3, 7, 1, 7, 1, 1, 5, \dots]; \\ \phi &= 1 + [1, 1, 1, 1, 1, 1, 1, 1, 1, \dots].\end{aligned}\tag{13}$$

The numbers A_1, A_2, \dots are called the *partial quotients* of X . Note the regular pattern that appears in the partial quotients for $\sqrt{8/29}, \phi$, and e ; the reasons for this behavior are discussed in exercises 12 and 16. There is no apparent pattern in the partial quotients for $\sqrt[3]{2}$, π , or γ .

It is interesting to note that the ancient Greeks' first definition of real numbers, once they had discovered the existence of irrationals, was essentially stated in terms of infinite continued fractions. (Later they adopted the suggestion of Eudoxus that $x = y$ should be defined instead as " $x < r$ if and only if $y < r$, for all rational r .") See O. Becker, *Quellen und Studien zur Geschichte Math., Astron., Physik* (B) 2 (1933), 311–333.

When X is a rational number, the regular continued fraction corresponds in a natural way to Euclid's algorithm. Let us assume that $X = v/u$, where $u > v \geq 0$. The regular continued fraction process starts with $X_0 = X$; let us define $U_0 = u$, $V_0 = v$. Assuming that $X_n = V_n/U_n \neq 0$, (10) becomes

$$\begin{aligned}A_{n+1} &= [U_n/V_n], \\ X_{n+1} &= U_n/V_n - A_{n+1} = (U_n \bmod V_n)/V_n.\end{aligned}\tag{14}$$

Therefore, if we define

$$U_{n+1} = V_n, \quad V_{n+1} = U_n \bmod V_n,\tag{15}$$

the condition $X_n = V_n/U_n$ holds throughout the process. Furthermore, (15) is precisely the transformation made on the variables u and v in Euclid's algorithm (see Algorithm 4.5.2A, step A2). For example, since $\frac{8}{29} = [3, 1, 1, 1, 2]$, we know that Euclid's algorithm applied to $u = 29$ and $v = 8$ will require exactly five division steps, and the quotients $[u/v]$ in step A2 will be successively 3, 1, 1, 1, and 2. Note that the last partial quotient A_n must be 2 or more when $X_n = 0$ and $n \geq 1$, since X_{n-1} is less than unity.

From this correspondence with Euclid's algorithm we can see that the regular continued fraction for X terminates at some step with $X_n = 0$ if and only if X is rational; for it is obvious that X_n cannot be zero if X is irrational, and, conversely, we know that Euclid's algorithm always terminates. If the partial quotients obtained during Euclid's algorithm are A_1, A_2, \dots, A_n , then we have,

by (5),

$$\frac{v}{u} = \frac{Q_{n-1}(A_2, \dots, A_n)}{Q_n(A_1, A_2, \dots, A_n)}. \quad (16)$$

This formula holds also if Euclid's algorithm is applied for $u < v$, when $A_1 = 0$. Furthermore, because of (8), $Q_{n-1}(A_2, \dots, A_n)$ and $Q_n(A_1, A_2, \dots, A_n)$ are relatively prime, and the fraction on the right-hand side of (16) is in lowest terms; therefore

$$u = Q_n(A_1, A_2, \dots, A_n)d, \quad v = Q_{n-1}(A_2, \dots, A_n)d, \quad (17)$$

where $d = \gcd(u, v)$.

The worst case. We can now apply these observations to determine the behavior of Euclid's algorithm in the “worst case,” or in other words to give an upper bound on the number of division steps. The worst case occurs when the inputs are consecutive Fibonacci numbers:

Theorem F (G. Lamé, 1845). *For $n \geq 1$, let u and v be integers with $u > v > 0$ such that Euclid's algorithm applied to u and v requires exactly n division steps, and such that u is as small as possible satisfying these conditions. Then $u = F_{n+2}$ and $v = F_{n+1}$.*

Proof. By (17), we must have $u = Q_n(A_1, A_2, \dots, A_n)d$, where A_1, A_2, \dots, A_n , and d are positive integers and $A_n \geq 2$. Since Q_n is a polynomial with nonnegative coefficients, involving all of the variables, the minimum value is achieved only when $A_1 = 1, \dots, A_{n-1} = 1, A_n = 2, d = 1$. Putting these values in (17) yields the desired result. ■

(This theorem has the historical claim of being the first practical application of the Fibonacci sequence; since then many other applications of Fibonacci numbers to algorithms and to the study of algorithms have been discovered.)

As a consequence of Theorem F we have an important corollary:

Corollary L. *If $0 \leq u, v < N$, the number of division steps required when Algorithm 4.5.2A is applied to u and v is at most $\lceil \log_\phi(\sqrt{5}N) \rceil - 2$.*

Proof. By Theorem F, the maximum number of steps, n , occurs when $u = F_n$ and $v = F_{n+1}$, where n is as large as possible with $F_{n+1} < N$. (The first division step in this case merely interchanges u and v when $n > 1$.) Since $F_{n+1} < N$, we have $\phi^{n+1}/\sqrt{5} < N$ (see Eq. 1.2.8–15), so $n+1 < \log_\phi(\sqrt{5}N)$. This completes the proof. ■

Note that $\log_\phi(\sqrt{5}N)$ is approximately $2.078 \ln N + 1.672 \approx 4.785 \log_{10} N + 1.672$. See exercises 31 and 36 for extensions of Theorem F.

An approximate model. Now that we know the maximum number of division steps that can occur, let us attempt to find the average number. Let $T(m, n)$ be the number of division steps that occur when $u = m$ and $v = n$ are input to Euclid's algorithm. Thus

$$T(m, 0) = 0; \quad T(m, n) = 1 + T(n, m \bmod n) \quad \text{if } n \geq 1. \quad (18)$$

Let T_n be the average number of division steps when $v = n$ and when u is chosen at random; since only the value of $u \bmod v$ affects the algorithm after the first division step, we may write

$$T_n = \frac{1}{n} \sum_{0 \leq k < n} T(k, n). \quad (19)$$

For example, $T(0, 5) = 1$, $T(1, 5) = 2$, $T(2, 5) = 3$, $T(3, 5) = 4$, $T(4, 5) = 3$, so

$$T_5 = \frac{1}{5}(1 + 2 + 3 + 4 + 3) = 2\frac{3}{5}.$$

In order to estimate T_n for large n , let us first try an approximation suggested by R. W. Floyd: We might assume that, for $0 \leq k < n$, the value of n is essentially "random" modulo k , so that we can set

$$T_n \approx 1 + \frac{1}{n} (T_0 + T_1 + \cdots + T_{n-1}).$$

Then $T_n \approx S_n$, where the sequence $\langle S_n \rangle$ is the solution to the recurrence relation

$$S_0 = 0, \quad S_n = 1 + \frac{1}{n} (S_0 + S_1 + \cdots + S_{n-1}), \quad n \geq 1. \quad (20)$$

(This approximation is analogous to the "lattice-point model" used to investigate Algorithm B in Section 4.5.2.)

The recurrence (20) is readily solved by the use of generating functions. A more direct way to solve it, analogous to our solution of the lattice-point model, is by noting that

$$\begin{aligned} S_{n+1} &= 1 + \frac{1}{n+1} (S_0 + S_1 + \cdots + S_{n-1} + S_n) \\ &= 1 + \frac{1}{n+1} (n(S_n - 1) + S_n) = S_n + \frac{1}{n+1}; \end{aligned}$$

hence S_n is $1 + \frac{1}{2} + \cdots + \frac{1}{n} = H_n$, a harmonic number. The approximation $T_n \approx S_n$ now suggests that $T_n \approx \ln n + O(1)$.

Comparison of this approximation with tables of the true value of T_n show, however, that $\ln n$ is too large; T_n does not grow this fast. One way to account for the fact that this approximation is too pessimistic is to observe that the average

value of $n \bmod k$ is less than the average value of $\frac{1}{2}k$, in the range $1 \leq k \leq n$:

$$\begin{aligned} \frac{1}{n} \sum_{1 \leq k \leq n} (n \bmod k) &= \frac{1}{n} \sum_{\substack{1 \leq q \leq n \\ \lfloor n/(q+1) \rfloor < k \leq \lfloor n/q \rfloor}} (n - qk) \\ &= n - \frac{1}{n} \sum_{1 \leq q \leq n} q \left(\binom{\lfloor n/q \rfloor + 1}{2} - \binom{\lfloor n/(q+1) \rfloor + 1}{2} \right) \\ &= n - \frac{1}{n} \sum_{1 \leq q \leq n} \binom{\lfloor n/q \rfloor + 1}{2} \\ &= \left(1 - \frac{\pi^2}{12} \right) n + O(\log n) \end{aligned} \quad (21)$$

(cf. exercise 4.5.2–10(c)). This is only about $.1775n$, not $.25n$; so the value of $n \bmod k$ tends to be smaller than the above model predicts, and Euclid's algorithm works faster than we might expect.

A continuous model. The behavior of Euclid's algorithm with $v = N$ is essentially determined by the behavior of the regular continued fraction process when $X = 0/N, 1/N, \dots, (N-1)/N$. Assuming that N is very large, we are led naturally to a study of regular continued fractions when X is a random real number uniformly distributed in $[0, 1]$. Therefore let us define the distribution function

$$F_n(x) = \text{probability that } X_n \leq x, \quad \text{for } 0 \leq x \leq 1, \quad (22)$$

given a uniform distribution of $X = X_0$. By the definition of regular continued fractions, we have $F_0(x) = x$, and

$$\begin{aligned} F_{n+1}(x) &= \sum_{k \geq 1} \text{probability that } (k \leq 1/X_n \leq k+x) \\ &= \sum_{k \geq 1} \text{probability that } (1/(k+x) \leq X_n \leq 1/k) \\ &= \sum_{k \geq 1} (F_n(1/k) - F_n(1/(k+x))). \end{aligned} \quad (23)$$

If the distributions $F_0(x), F_1(x), \dots$ defined by these formulas approach a limiting distribution $F_\infty(x) = F(x)$, we will have

$$F(x) = \sum_{k \geq 1} (F(1/k) - F(1/(k+x))). \quad (24)$$

One function that satisfies this relation is $F(x) = \log_b(1+x)$, for any base $b > 1$; see exercise 19. The further condition $F(1) = 1$ implies that we should take $b = 2$. Thus it is reasonable to make a guess that $F(x) = \lg(1+x)$, and that $F_n(x)$ approaches this behavior.

We might conjecture, for example, that $F(\frac{1}{2}) = \lg(\frac{3}{2}) \approx 0.58496$; let us see how close $F_n(\frac{1}{2})$ comes to this value for small n . We have $F_0(\frac{1}{2}) = \frac{1}{2}$, and

$$\begin{aligned} F_1(\frac{1}{2}) &= \frac{1}{1} - \frac{1}{1 + \frac{1}{2}} + \frac{1}{2} - \frac{1}{2 + \frac{1}{2}} + \dots \\ &= 2\left(\frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \dots\right) = 2(1 - \ln 2) \approx 0.6137; \\ F_2(\frac{1}{2}) &= \sum_{m \geq 1} \frac{2}{m} \left(\frac{1}{2m+2} - \frac{1}{3m+2} + \frac{1}{4m+2} - \frac{1}{5m+2} + \dots \right) \\ &= \sum_{m \geq 1} \frac{2}{m^2} \left(\frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \dots \right) \\ &\quad - \sum_{m \geq 1} \frac{4}{m} \left(\frac{1}{2m(2m+2)} - \frac{1}{3m(3m+2)} + \dots \right) \\ &= \frac{1}{3}\pi^2(1 - \ln 2) - \sum_{m \geq 1} \frac{4S_m}{m^2}, \end{aligned}$$

where $S_m = 1/(4m+4) - 1/(9m+6) + 1/(16m+8) - \dots$. Using the values of H_x for fractional x found in Table 3 of Appendix B, we find that

$$S_1 = \frac{1}{12}, \quad S_2 = \frac{3}{4} - \ln 2, \quad S_3 = \frac{19}{20} - \pi/(2\sqrt{3}),$$

etc.; a numerical evaluation yields $F_2(\frac{1}{2}) \approx 0.5748$. Although $F_1(x) = H_x$, it is clear that $F_n(x)$ is difficult to calculate exactly when n is large.

The distributions $F_n(x)$ were first studied by K. F. Gauss, who thought of the problem in 1800. His notebook for that year lists various recurrence relations and gives a brief table of values, including the four-place value for $F_2(\frac{1}{2})$ that has just been mentioned. After performing these calculations, Gauss wrote, "Tam complicatae evadunt, ut nulla spes superesse videatur," i.e., "They come out so complicated that no hope appears to be left." Twelve years later, Gauss wrote a letter to Laplace in which he posed the problem as one he could not resolve to his satisfaction. He said, "I found by very simple reasoning that, for n infinite, $F_n(x) = \log(1+x)/\log 2$. But the efforts that I made since then in my inquiries to assign $F_n(x) = \log(1+x)/\log 2$ for very large but not infinite values of n were fruitless." He never published his "very simple reasoning," and it is not completely clear that he had found a rigorous proof. More than 100 years went by before a proof was finally published, by R. O. Kuz'min [*Atti del Congresso internazionale dei matematici* 6 (Bologna, 1928), 83–89], who showed that

$$F_n(x) = \lg(1+x) + O(e^{-A\sqrt{n}})$$

for some positive constant A . The error term was improved to $O(e^{-An})$ by Paul Lévy shortly afterward [*Bull. Soc. Math. de France* 57 (1929), 178–194]*; but

*An exposition of Lévy's interesting proof appeared in the first edition of this book.

Gauss's problem, namely to find the asymptotic behavior of $F_n(x) - \lg(1+x)$, was not really resolved until 1974, when Eduard Wirsing published a beautiful analysis of the situation [*Acta Arithmetica* 24 (1974), 507–528]. We shall study the simplest aspects of Wirsing's approach here, since his method is an instructive use of linear operators.

If G is any function of x defined for $0 \leq x \leq 1$, let SG be the function defined by

$$SG(x) = \sum_{k \geq 1} \left(G\left(\frac{1}{k}\right) - G\left(\frac{1}{k+x}\right) \right). \quad (25)$$

Thus, S is an operator that changes one function into another. In particular, by (23) we have $F_{n+1}(x) = SF_n(x)$, hence

$$F_n = S^n F_0. \quad (26)$$

(In this discussion F_n stands for a distribution function, *not* for a Fibonacci number.) Note that S is a “linear operator”; i.e., $S(cG) = c(SG)$ for all constants c , and $S(G_1 + G_2) = SG_1 + SG_2$.

Now if G has a bounded first derivative, we can differentiate (25) term by term to show that

$$(SG)'(x) = \sum_{k \geq 1} \frac{1}{(k+x)^2} G'\left(\frac{1}{k+x}\right); \quad (27)$$

hence SG also has a bounded first derivative. (Term-by-term differentiation of a convergent series is justified when the series of derivatives is uniformly convergent; cf. K. Knopp, *Theory and Application of Infinite series* (Glasgow: Blackie, 1951), §47.)

Let $H = SG$, and let $g(x) = (1+x)G'(x)$, $h(x) = (1+x)H'(x)$. It follows that

$$\begin{aligned} h(x) &= \sum_{k \geq 1} \frac{1+x}{(k+x)^2} \left(1 + \frac{1}{k+x}\right)^{-1} g\left(\frac{1}{k+x}\right) \\ &= \sum_{k \geq 1} \left(\frac{k}{k+1+x} - \frac{k-1}{k+x} \right) g\left(\frac{1}{k+x}\right). \end{aligned}$$

In other words, $h = Tg$, where T is the linear operator defined by

$$Tg(x) = \sum_{k \geq 1} \left(\frac{k}{k+1+x} - \frac{k-1}{k+x} \right) g\left(\frac{1}{k+x}\right). \quad (28)$$

Continuing, we see that if g has a bounded first derivative, we can differentiate term by term to show that Tg does also:

$$\begin{aligned}(Tg)'(x) &= -\sum_{k \geq 1} \left(\left(\frac{k}{(k+1+x)^2} - \frac{k-1}{(k+x)^2} \right) g\left(\frac{1}{k+x}\right) \right. \\ &\quad \left. + \left(\frac{k}{k+1+x} - \frac{k-1}{k+x} \right) \frac{1}{(k+x)^2} g'\left(\frac{1}{k+x}\right) \right) \\ &= -\sum_{k \geq 1} \left(\frac{k}{(k+1+x)^2} \left(g\left(\frac{1}{k+x}\right) - g\left(\frac{1}{k+1+x}\right) \right) \right. \\ &\quad \left. + \frac{1+x}{(k+x)^3(k+1+x)} g'\left(\frac{1}{k+x}\right) \right).\end{aligned}$$

There is consequently a third linear operator, U , such that $(Tg)' = -U(g')$, namely

$$\begin{aligned}U\varphi(x) &= \sum_{k \geq 1} \left(\frac{k}{(k+1+x)^2} \int_{1/(k+1+x)}^{1/(k+x)} \varphi(t) dt \right. \\ &\quad \left. + \frac{1+x}{(k+x)^3(k+1+x)} \varphi\left(\frac{1}{k+x}\right) \right). \quad (29)\end{aligned}$$

What is the relevance of all this to our problem? Well, if we set

$$F_n(x) = \lg(1+x) + R_n(\lg(1+x)), \quad (30)$$

$$f_n(x) = (1+x)F'_n(x) = \frac{1}{\ln 2}(1+R'_n(\lg(1+x))), \quad (31)$$

we have

$$f'_n(x) = R''_n(\lg(1+x)) / ((\ln 2)^2(1+x)); \quad (32)$$

the effect of the $\lg(1+x)$ term disappears, after these transformations. Furthermore since $F_n = S^n F_0$ we have $f_n = T^n f_0$ and $f'_n = (-1)^n U^n f'_0$. Both F_n and f_n have bounded derivatives, by induction on n . Thus (32) becomes

$$(-1)^n R''_n(\lg(1+x)) = (1+x)(\ln 2)^2 U^n f'_0(x). \quad (33)$$

Now $F_0(x) = x$, $f_0(x) = 1+x$, and $f'_0(x)$ is the constant function 1. We are going to show that the operator U^n takes the constant function into a function with very small values, hence $|R''_n(x)|$ must be very small for $0 \leq x \leq 1$. Finally we can clinch the argument by showing that $R_n(x)$ itself is small: Since we have $R_n(0) = R_n(1) = 1$, it follows from a well-known interpolation formula (cf. exercise 4.6.4-15 with $x_0 = 0$, $x_1 = x$, $x_2 = 1$) that

$$R_n(x) = -\frac{x(1-x)}{2} R''_n(\xi(x)) \quad (34)$$

for some function $\xi(x)$, where $0 \leq \xi(x) \leq 1$ when $0 \leq x \leq 1$.

Thus everything hinges on our being able to prove that U^n produces small function values, where U is the linear operator defined in (29). Note that U is a positive operator, in the sense that $U\varphi(x) \geq 0$ for all x if $\varphi(x) \geq 0$ for all x . It follows that U is order-preserving: If $\varphi_1(x) \leq \varphi_2(x)$ for all x then we have $U\varphi_1(x) \leq U\varphi_2(x)$ for all x .

One way to exploit this property is to find a function φ for which we can calculate $U\varphi$ exactly and to use constant multiples of this function to bound the ones that we are really interested in. First let us look for a function g such that Tg is easy to compute. If we consider functions defined for all $x \geq 0$, instead of only on $[0, 1]$, it is easy to remove the summation from (25) by observing that

$$SG(x+1) - SG(x) = G\left(\frac{1}{1+x}\right) - \lim_{k \rightarrow \infty} G\left(\frac{1}{k+x}\right) = G\left(\frac{1}{1+x}\right) - G(0) \quad (35)$$

when G is continuous. Since $T((1+x)G') = (1+x)(SG)'$, it follows (see exercise 20) that

$$\frac{Tg(x)}{x+1} - \frac{Tg(x+1)}{x+2} = \left(\frac{1}{x+1} - \frac{1}{x+2}\right)g\left(\frac{1}{1+x}\right). \quad (36)$$

If we set $Tg(x) = 1/(x+1)$, we find that the corresponding value of $g(x)$ is $1+x - 1/(1+x)$. Let $\varphi(x) = g'(x) = 1 + 1/(1+x)^2$, so that $U\varphi(x) = -(Tg)'(x) = 1/(1+x)^2$; this is the function φ we have been looking for.

For this choice of φ we have $2 \leq \varphi(x)/U\varphi(x) = (1+x)^2 + 1 \leq 5$ for $0 \leq x \leq 1$, hence

$$\frac{1}{5}\varphi \leq U\varphi \leq \frac{1}{2}\varphi.$$

By the positivity of U and φ we can apply U to this inequality again, obtaining $\frac{1}{25}\varphi \leq \frac{1}{5}U\varphi \leq U^2\varphi \leq \frac{1}{2}U\varphi \leq \frac{1}{4}\varphi$; and after $n-1$ applications we have

$$5^{-n}\varphi \leq U^n\varphi \leq 2^{-n}\varphi \quad (37)$$

for this particular φ . Let $\chi(x) = f'_0(x) = 1$ be the constant function; then for $0 \leq x \leq 1$ we have $\frac{5}{4}\chi \leq \varphi \leq 2\chi$, hence

$$\frac{5}{8}5^{-n}\chi \leq \frac{1}{2}5^{-n}\varphi \leq \frac{1}{2}U^n\varphi \leq U^n\chi \leq \frac{4}{5}U^n\varphi \leq \frac{4}{5}2^{-n}\varphi \leq \frac{8}{5}2^{-n}\chi.$$

It follows by (33) that

$$\frac{5}{8}(\ln 2)^2 5^{-n} \leq (-1)^n R''_n(x) \leq \frac{16}{5}(\ln 2)^2 2^{-n}, \quad \text{for } 0 \leq x \leq 1;$$

hence by (30) and (34) we have proved the following result:

Theorem W. *The distribution $F_n(x)$ equals $\lg(1+x) + O(2^{-n})$ as $n \rightarrow \infty$. In fact, $F_n(x) - \lg(1+x)$ lies between $\frac{5}{16}(-1)^{n+1}5^{-n}(\ln(1+x))(\ln 2/(1+x))$ and $\frac{8}{5}(-1)^{n+1}2^{-n}(\ln(1+x))(\ln 2/(1+x))$, for $0 \leq x \leq 1$. ■*

With a slightly different choice of φ , we can obtain tighter bounds (see exercise 21). In fact, Wirsing went much further in his paper, proving that

$$F_n(x) = \lg(1+x) + (-\lambda)^n \Psi(x) + O(x(1-x)(\lambda - 0.031)^n), \quad (38)$$

where

$$\begin{aligned} \lambda &= 0.30366\,30028\,98732\,65860\dots \\ &= 1/3, 3, 2, 2, 3, 13, 1, 174, 1, 1, 1, 2, 2, 2, 1, 1, 1, 2, 2, 1, \dots \end{aligned} \quad (39)$$

is a fundamental constant (apparently unrelated to more familiar constants), and where Ψ is an interesting function that is analytic in the entire complex plane except for the negative real axis from -1 to $-\infty$. Wirsing's function satisfies $\Psi(0) = \Psi(1) = 0$, $\Psi'(0) < 0$, and $S\Psi = -\lambda\Psi$; thus by (35) it satisfies the identity

$$\Psi(z) - \Psi(z+1) = \frac{1}{\lambda} \Psi\left(\frac{1}{1+z}\right). \quad (40)$$

Furthermore, Wirsing demonstrated that

$$\Psi\left(-\frac{u}{v} + \frac{i}{N}\right) = c\lambda^{-n} \log N + O(1) \quad \text{as } N \rightarrow \infty, \quad (41)$$

where c is a constant and $n = T(u, v)$ is the number of iterations when Euclid's algorithm is applied to the integers $u > v > 0$.

A complete solution to Gauss's problem was found a few years later by K. I. Babenko [*Doklady Akad. Nauk SSSR* **238** (1978), 1021–1024], who used powerful techniques of functional analysis to prove that

$$F_n(x) = \lg(1+x) + \sum_{j \geq 2} \lambda_j^n \Psi_j(x) \quad (42)$$

for all $0 \leq x \leq 1$, $n \geq 1$. Here $|\lambda_2| > |\lambda_3| \geq |\lambda_4| \geq \dots$, and each $\Psi_j(z)$ is an analytic function in the complex plane except for a cut at $[-\infty, -1]$. The function Ψ_2 is Wirsing's Ψ , and $\lambda_2 = -\lambda$, while $\lambda_3 = 0.1009$, $\lambda_4 = -0.0408$, $\lambda_5 = -0.0355$, $\lambda_6 = 0.0128$. Babenko also established further properties of the eigenvalues λ_j , proving in particular that they are exponentially small as $j \rightarrow \infty$, and that the sum for $j \geq k$ in (42) is bounded by $(\pi^2/6)|\lambda_k|^{n-1} \min(x, 1-x)$. [Further information appears in a paper by Babenko and Fur'ev, *Doklady Akad. Nauk SSSR* **240** (1978), 1273–1276.]

From continuous to discrete. We have now derived results about the probability distributions for continued fractions when X is a real number uniformly distributed in the interval $[0, 1]$. But a real number is rational with probability zero (almost all numbers are irrational), so these results do not apply directly to Euclid's algorithm. Before we can apply Theorem W to our problem, some technicalities must be overcome. Consider the following observation based on elementary measure theory:

Lemma M. Let $I_1, I_2, \dots, J_1, J_2, \dots$ be pairwise disjoint intervals contained in the interval $[0, 1)$, and let

$$\mathcal{I} = \bigcup_{k \geq 1} I_k, \quad \mathcal{J} = \bigcup_{k \geq 1} J_k, \quad \mathcal{K} = [0, 1] \setminus (\mathcal{I} \cup \mathcal{J}).$$

Assume that \mathcal{K} has measure zero. Let P_n be the set $\{0/n, 1/n, \dots, (n-1)/n\}$. Then

$$\lim_{n \rightarrow \infty} \frac{\|\mathcal{I} \cap P_n\|}{n} = \mu(\mathcal{I}). \quad (43)$$

Here $\mu(\mathcal{I})$ is the Lebesgue measure of \mathcal{I} , namely, $\sum_{k \geq 1} \text{length}(I_k)$; and $\|\mathcal{I} \cap P_n\|$ denotes the number of elements in the set $\mathcal{I} \cap P_n$.

Proof. Let $\mathcal{I}_N = \bigcup_{1 \leq k \leq N} I_k$ and $\mathcal{J}_N = \bigcup_{1 \leq k \leq N} J_k$. Given $\epsilon > 0$, find N large enough so that $\mu(\mathcal{I}_N) + \mu(\mathcal{J}_N) \geq 1 - \epsilon$, and let

$$\mathcal{K}_N = \mathcal{K} \cup \bigcup_{k > N} I_k \cup \bigcup_{k > N} J_k.$$

If I is an interval, having any of the forms (a, b) or $[a, b)$ or $(a, b]$ or $[a, b]$, it is clear that $\mu(I) = b - a$ and

$$n\mu(I) - 1 \leq \|\mathcal{I} \cap P_n\| \leq n\mu(I) + 1.$$

Now let $r_n = \|\mathcal{I}_N \cap P_n\|$, $s_n = \|\mathcal{J}_N \cap P_n\|$, $t_n = \|\mathcal{K}_N \cap P_n\|$; we have

$$\begin{aligned} r_n + s_n + t_n &= n; \\ n\mu(\mathcal{I}_N) - N &\leq r_n \leq n\mu(\mathcal{I}_N) + N; \\ n\mu(\mathcal{I}_N) - N &\leq s_n \leq n\mu(\mathcal{J}_N) + N. \end{aligned}$$

Hence

$$\begin{aligned} \mu(\mathcal{I}) - \frac{N}{n} - \epsilon &\leq \mu(\mathcal{I}_N) - \frac{N}{n} \leq \frac{r_n + t_n}{n} \\ &= 1 - \frac{s_n}{n} \leq 1 - \mu(\mathcal{J}_N) + \frac{N}{n} \leq \mu(\mathcal{I}) + \frac{N}{n} + \epsilon. \end{aligned}$$

This holds for all n and for all ϵ ; hence $\lim_{n \rightarrow \infty} r_n/n = \mu(\mathcal{I})$. ■

Exercise 25 shows that Lemma M is not trivial, in the sense that some rather restrictive hypotheses are needed to prove (43).

Distribution of partial quotients. Now we can put Theorem W and Lemma M together to derive some solid facts about Euclid's algorithm.

Theorem E. Let n and k be positive integers, and let $p_k(a, n)$ be the probability that the $(k+1)$ st quotient A_{k+1} in Euclid's algorithm is equal to a , when $v = n$ and u is chosen at random. Then

$$\lim_{n \rightarrow \infty} p_k(a, n) = F_k\left(\frac{1}{a}\right) - F_k\left(\frac{1}{a+1}\right),$$

where $F_k(x)$ is the distribution function (21).

Proof. The set I of all X in $[0, 1)$ for which $A_{k+1} = a$ is a union of disjoint intervals, and so is the set J of all X for which $A_{k+1} \neq a$. Lemma M therefore applies, with K the set of all X for which A_{k+1} is undefined. Furthermore, $F_k(1/a) - F_k(1/(a+1))$ is the probability that $1/(a+1) < X_k \leq 1/a$, which is $\mu(I)$, the probability that $A_{k+1} = a$. ■

As a consequence of Theorems E and W, we can say that a quotient equal to a occurs with the approximate probability

$$\lg(1 + 1/a) - \lg(1 + 1/(a+1)) = \lg((a+1)^2 / ((a+1)^2 - 1)).$$

Thus

- a quotient of 1 occurs about $\lg(\frac{1}{3}) = 41.504$ percent of the time;
- a quotient of 2 occurs about $\lg(\frac{9}{8}) = 16.992$ percent of the time;
- a quotient of 3 occurs about $\lg(\frac{16}{15}) = 9.311$ percent of the time;
- a quotient of 4 occurs about $\lg(\frac{25}{24}) = 5.890$ percent of the time.

Actually, if Euclid's algorithm produces the quotients A_1, A_2, \dots, A_t , the nature of the proofs above will guarantee this behavior only for A_k when k is comparatively small with respect to t ; the values A_{t-1}, A_{t-2}, \dots are not covered by this proof. But we can in fact show that the distribution of the last quotients A_{t-1}, A_{t-2}, \dots is essentially the same as the first.

For example, consider the regular continued fraction expansions for the set of all proper fractions whose denominator is 29:

$\frac{1}{29} = [29]$	$\frac{8}{29} = [3, 1, 1, 1, 2]$	$\frac{15}{29} = [1, 1, 14]$	$\frac{22}{29} = [1, 3, 7]$
$\frac{2}{29} = [14, 2]$	$\frac{9}{29} = [3, 4, 2]$	$\frac{16}{29} = [1, 1, 4, 3]$	$\frac{23}{29} = [1, 3, 1, 5]$
$\frac{3}{29} = [9, 1, 2]$	$\frac{10}{29} = [2, 1, 9]$	$\frac{17}{29} = [1, 1, 2, 2, 2]$	$\frac{24}{29} = [1, 4, 1, 4]$
$\frac{4}{29} = [7, 4]$	$\frac{11}{29} = [2, 1, 1, 1, 3]$	$\frac{18}{29} = [1, 1, 1, 1, 3]$	$\frac{25}{29} = [1, 6, 4]$
$\frac{5}{29} = [5, 1, 4]$	$\frac{12}{29} = [2, 2, 2, 2]$	$\frac{19}{29} = [1, 1, 1, 9]$	$\frac{26}{29} = [1, 8, 1, 2]$
$\frac{6}{29} = [4, 1, 5]$	$\frac{13}{29} = [2, 4, 3]$	$\frac{20}{29} = [1, 2, 4, 2]$	$\frac{27}{29} = [1, 13, 2]$
$\frac{7}{29} = [4, 7]$	$\frac{14}{29} = [2, 14]$	$\frac{21}{29} = [1, 2, 1, 1, 1, 2]$	$\frac{28}{29} = [1, 28]$

Several things can be observed in this table.

a) As mentioned earlier, the last quotient is always 2 or more. Furthermore, we have the obvious identity

$$|x_1, \dots, x_{n-1}, x_n + 1| = |x_1, \dots, x_{n-1}, x_n, 1|, \quad (44)$$

and this shows how partial fractions whose last quotient is unity are related to regular continued fractions.

b) The values in the right-hand columns have a simple relationship to the values in the left-hand columns; can the reader see the correspondence before reading any further? The relevant identity is

$$1 - |x_1, x_2, \dots, x_n| = |1, x_1 - 1, x_2, \dots, x_n|; \quad (45)$$

see exercise 9.

c) There is symmetry between left and right in the first two columns: If $|A_1, A_2, \dots, A_t|$ occurs, so does $|A_t, \dots, A_2, A_1|$. This will always be the case (see exercise 26).

d) If we examine all of the quotients in the table, we find that there are 96 in all, of which $\frac{39}{96} = 40.6$ percent are equal to 1, $\frac{21}{96} = 21.9$ percent are equal to 2, $\frac{8}{96} = 8.3$ percent are equal to 3; this agrees reasonably well with the probabilities listed above.

The number of division steps. Let us now return to our original problem and investigate T_n , the average number of division steps when $v = n$. (See Eq. (19).) Here are some sample values of T_n :

$n =$	95	96	97	98	99	100	101	102	103	104	105
$T_n =$	5.0	4.4	5.3	4.8	4.7	4.6	5.3	4.6	5.3	4.7	4.6
$n =$	996	997	998	999	1000	1001	...	9999	10000	10001	
$T_n =$	6.5	7.3	7.0	6.8	6.4	6.7	...	8.6	8.3	9.1	
$n =$	49999	50000	50001	...	99999	100000	100001				
$T_n =$	10.6	9.7	10.0	...	10.7	10.3	11.0				

Note the somewhat erratic behavior; T_n tends to be higher than its neighbors when n is prime, and it is correspondingly lower when n has many divisors. (In this list, 97, 101, 103, 997, and 49999 are primes; $10001 = 73 \cdot 137$, $50001 = 3 \cdot 7 \cdot 2381$, $99999 = 3 \cdot 3 \cdot 41 \cdot 271$, and $100001 = 11 \cdot 9091$.) It is not difficult to understand why this happens: if $\gcd(u, v) = d$, Euclid's algorithm applied to u and v behaves essentially the same as if it were applied to u/d and v/d . Therefore, when $v = n$ has several divisors, there are many choices of u for which n behaves as if it were smaller.

Accordingly let us consider another quantity, τ_n , which is the average number of division steps when $v = n$ and when u is relatively prime to n . Thus

$$\tau_n = \frac{1}{\varphi(n)} \sum_{\substack{0 \leq m < n \\ \gcd(m, n) = 1}} T(m, n). \quad (46)$$

It follows that

$$T_n = \frac{1}{n} \sum_{d|n} \varphi(d) \tau_d. \quad (47)$$

Here is a table of τ_n for the same values of n considered above:

$n =$	95	96	97	98	99	100	101	102	103	104	105
$\tau_n =$	5.4	5.3	5.3	5.6	5.2	5.2	5.4	5.3	5.4	5.3	5.6
$n =$	996	997	998	999	1000	1001	...	9999	10000	10001	
$\tau_n =$	7.2	7.3	7.3	7.3	7.3	7.4	...	9.21	9.21	9.22	
$n =$	49999	50000	50001	...	99999	100000	100001				
$\tau_n =$	10.58	10.57	10.59	...	11.170	11.172	11.172				

Clearly τ_n is much more well-behaved than T_n , and it should be more susceptible to analysis. Inspection of a table of τ_n for small n reveals some curious anomalies; for example, $\tau_{50} = \tau_{100}$ and $\tau_{60} = \tau_{120}$. But as n grows, the values of τ_n behave quite regularly indeed, as the above table indicates, and they show no significant relation to the factorization properties of n . If the reader will plot the values of τ_n versus $\ln n$ on graph paper, for the values of τ_n given above, he will see that the values lie very nearly on a straight line, and that the formula

$$\tau_n \approx 0.843 \ln n + 1.47 \quad (48)$$

is a very good approximation.

We can account for this behavior if we study the regular continued fraction process a little further. Note that in Euclid's algorithm as expressed in (15) we have

$$\frac{V_0}{U_0} \frac{V_1}{U_1} \cdots \frac{V_{t-1}}{U_{t-1}} = \frac{V_{t-1}}{U_0},$$

since $U_{k+1} = V_k$; therefore if $U = U_0$ and $V = V_0$ are relatively prime, and if there are t division steps, we have

$$X_0 X_1 \dots X_{t-1} = 1/U.$$

Setting $U = N$ and $V = m < N$, we find that

$$\ln X_0 + \ln X_1 + \dots + \ln X_{t-1} = -\ln N. \quad (49)$$

We know the approximate distribution of X_0, X_1, X_2, \dots , so we can use this equation to estimate

$$t = T(N, m) = T(m, N) - 1.$$

Returning to the formulas preceding Theorem W, we find that the average value of $\ln X_n$, when X_0 is a real number uniformly distributed in $[0, 1]$, is

$$\int_0^1 \ln x F'_n(x) dx = \int_0^1 \ln x f_n(x) dx / (1+x), \quad (50)$$

where $f_n(x)$ is defined in (31). Now

$$f_n(x) = \frac{1}{\ln 2} + O(2^{-n}), \quad (51)$$

using the facts we have derived earlier (see exercise 23); hence the average value of $\ln X_n$ is very well approximated by

$$\begin{aligned} \frac{1}{\ln 2} \int_0^1 \frac{\ln x}{1+x} dx &= -\frac{1}{\ln 2} \int_0^\infty \frac{ue^{-u}}{1+e^{-u}} du \\ &= -\frac{1}{\ln 2} \sum_{k \geq 1} (-1)^{k+1} \int_0^\infty ue^{-ku} du \\ &= -\frac{1}{\ln 2} \left(1 - \frac{1}{4} + \frac{1}{9} - \frac{1}{16} + \frac{1}{25} - \dots \right) \\ &= -\frac{1}{\ln 2} \left(1 + \frac{1}{4} + \frac{1}{9} + \dots - 2 \left(\frac{1}{4} + \frac{1}{16} + \frac{1}{36} + \dots \right) \right) \\ &= -\frac{1}{2 \ln 2} \left(1 + \frac{1}{4} + \frac{1}{9} + \dots \right) \\ &= -\pi^2 / (12 \ln 2). \end{aligned}$$

By (49) we therefore expect to have the approximate formula

$$-t\pi^2 / (12 \ln 2) \approx -\ln N;$$

that is, t should be approximately equal to $((12 \ln 2)/\pi^2) \ln N$. This constant $(12 \ln 2)/\pi^2 = 0.842765913\dots$ agrees perfectly with the empirical formula (48) obtained earlier, so we have good reason to believe that the formula

$$\tau_n \approx \frac{12 \ln 2}{\pi^2} \ln n + 1.47 \quad (52)$$

indicates the true asymptotic behavior of τ_n as $n \rightarrow \infty$.

If we assume that (52) is valid, we obtain the formula

$$T_n \approx \frac{12 \ln 2}{\pi^2} \left(\ln n - \sum_{d \mid n} \Lambda(d)/d \right) + 1.47, \quad (53)$$

where $\Lambda(d)$ is *von Mangoldt's function* defined by the rules

$$\Lambda(n) = \begin{cases} \ln p, & \text{if } n = p^r \text{ for } p \text{ prime and } r \geq 1; \\ 0, & \text{otherwise.} \end{cases} \quad (54)$$

For example,

$$\begin{aligned} T_{100} &\approx \frac{12 \ln 2}{\pi^2} \left(\ln 100 - \frac{\ln 2}{2} - \frac{\ln 2}{4} - \frac{\ln 2}{5} - \frac{\ln 5}{25} \right) + 1.47 \\ &\approx (0.843)(4.605 - 0.347 - 0.173 - 0.322 - 0.064) + 1.47 \\ &\approx 4.59; \end{aligned}$$

the exact value of T_{100} is 4.56.

We can also estimate the average number of division steps when u and v are both uniformly distributed between 1 and N , by calculating

$$\frac{1}{N} \sum_{1 \leq n \leq N} T_n. \quad (55)$$

Assuming formula (53), exercise 27 shows that this sum has the form

$$\frac{12 \ln 2}{\pi^2} \ln N + O(1), \quad (56)$$

and empirical calculations with the same numbers used to derive Eq. 4.5.2–45 show good agreement with the formula

$$\frac{12 \ln 2}{\pi^2} \ln N + 0.06. \quad (57)$$

Of course we have not yet proved anything about T_n and τ_n in general; so far we have only been considering plausible reasons why the above formulas ought to hold. Fortunately it is now possible to supply rigorous proofs, based on a careful analysis by several mathematicians.

The leading coefficient $(12 \ln 2)/\pi^2$ in the above formulas was established first, in independent studies by John D. Dixon and Hans A. Heilbronn. Dixon [*J. Number Theory* 2 (1970), 414–422] developed the theory of the $F_n(x)$ distributions to show that individual partial quotients are essentially independent of each other in an appropriate sense, and proved that for all positive ϵ we have $|T(m, n) - ((12 \ln 2)/\pi^2) \ln n| < (\ln n)^{(1/2)+\epsilon}$ except for $\exp(-c(\epsilon)(\log N)^{\epsilon/2})N^2$ values of m and n in the range $1 \leq m < n \leq N$, where $c(\epsilon) > 0$. Heilbronn's approach was completely different, working entirely with integers instead of continuous variables. His idea, which is presented in slightly modified form in exercises 33 and 34, is based on the fact that τ_n can be related to the number of ways to represent n in a certain manner. Furthermore, his paper [*Number Theory and Analysis*, ed. by Paul Turán (New York: Plenum, 1969), 87–96] shows that the distribution of individual partial quotients 1, 2, ... that we have discussed above actually applies to the entire collection of partial quotients belonging to the fractions having a given denominator; this is a sharper form of Theorem E. A still sharper result was obtained several years later by J. W. Porter [*Mathematika* 22

(1975), 20–28], who established that

$$\tau_n = \frac{12 \ln 2}{\pi^2} \ln n + C + O(n^{-1/6+\epsilon}), \quad (58)$$

where

$$\boxed{\dots - 1/2 \cdot 1/2 \cdot 1/2 \cdot 1/2 \cdots}$$

see D. E. Knuth, *Computers and Math. with Applic.* 2 (1976), 137–139. Thus the conjecture (48) is fully proved.

The average running time for Euclid's algorithm on multiple-precision integers, using classical algorithms for arithmetic, was shown to be of order

$$(1 + \log(\max(u, v)/\gcd(u, v))) \log \min(u, v)$$

by G. E. Collins, in *SIAM J. Computing* 3 (1974), 1–10.

Summary. We have found that the worst case of Euclid's algorithm occurs when its inputs u and v are consecutive Fibonacci numbers (Theorem F); the number of division steps when $v = n$ will never exceed $\lceil 4.8 \log_{10} N - 0.32 \rceil$. We have determined the frequency of the values of various partial quotients, showing, for example, that the division step finds $\lfloor u/v \rfloor = 1$ about 41 percent of the time (Theorem E). And, finally, the theorems of Heilbronn and Porter prove that the average number T_n of division steps when $v = n$ is approximately

$$((12 \ln 2)/\pi^2) \ln n \approx 1.9405 \log_{10} n,$$

minus a correction term based on the divisors of n as shown in Eq. (53).

↑

EXERCISES

- 1. [20] Since the quotient $\lfloor u/v \rfloor$ is equal to unity over 40 percent of the time in Algorithm 4.5.2A, it may be advantageous on some computers to make a test for this case and to avoid the division when the quotient is unity. Is the following MIX program for Euclid's algorithm more efficient than Program 4.5.2A?

```

LDX U      rX ← u.
JMP 2F
1H STX V      v ← rX.
SUB V      rA ← u - v.
CMPA V
SRAX 5      rAX ← rA.
JL 2F      Is u - v < v?
DIV V      rX ← rAX mod v.
2H LDA V      rA ← v.
JXNZ 1B      Done if rX = 0. ■

```

2. [M21] Evaluate the matrix product

$$\begin{pmatrix} x_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_2 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} x_n & 1 \\ 1 & 0 \end{pmatrix}.$$

3. [M21] What is the value of

$$\det \begin{pmatrix} x_1 & 1 & 0 & \dots & 0 \\ -1 & x_2 & 1 & & 0 \\ 0 & -1 & x_3 & 1 & \vdots \\ \vdots & & -1 & \ddots & 1 \\ 0 & 0 & \dots & -1 & x_n \end{pmatrix}?$$

4. [M20] Prove Eq. (8).

5. [HM25] Let x_1, x_2, \dots be a sequence of real numbers that are each greater than some positive real number ϵ . Prove that the infinite continued fraction $|x_1, x_2, \dots| = \lim_{n \rightarrow \infty} |x_1, \dots, x_n|$ exists. Show also that $|x_1, x_2, \dots|$ need not exist if we assume only that $x_j > 0$ for all j .

6. [M23] Prove that the regular continued fraction expansion of a number is *unique* in the following sense: If B_1, B_2, \dots are positive integers, then the infinite continued fraction $|B_1, B_2, \dots|$ is an irrational number X between 0 and 1 whose regular continued fraction has $A_n = B_n$ for all $n \geq 1$; and if B_1, \dots, B_m are positive integers with $B_m > 1$, then the regular continued fraction for $X = |B_1, \dots, B_m|$ has $A_n = B_n$ for $1 \leq n \leq m$.

7. [M26] Find all permutations $p(1)p(2)\dots p(n)$ of the integers $\{1, 2, \dots, n\}$ such that $Q_n(x_1, x_2, \dots, x_n) = Q_n(x_{p(1)}, x_{p(2)}, \dots, x_{p(n)})$ holds for all x_1, x_2, \dots, x_n .

8. [M20] Show that $-1/X_n = |A_n, \dots, A_1, -X|$, whenever X_n is defined, in the regular continued fraction process.

9. [M21] Show that continued fractions satisfy the following identities:

a) $|x_1, \dots, x_n| = |x_1, \dots, x_k + |x_{k+1}, \dots, x_n||, \quad 1 \leq k \leq n;$

b) $|0, x_1, x_2, \dots, x_n| = x_1 + |x_2, \dots, x_n|, \quad n \geq 1;$

c) $|x_1, \dots, x_{k-1}, x_k, 0, x_{k+1}, x_{k+2}, \dots, x_n|$
 $= |x_1, \dots, x_{k-1}, x_k + x_{k+1}, x_{k+2}, \dots, x_n|, \quad 1 \leq k < n;$

d) $1 - |x_1, x_2, \dots, x_n| = |1, x_1 - 1, x_2, \dots, x_n|, \quad n \geq 1.$

10. [M28] By the result of exercise 6, every irrational real number X has a unique regular continued fraction representation of the form

$$X = A_0 + |A_1, A_2, A_3, \dots|,$$

where A_0 is an integer and A_1, A_2, A_3, \dots are positive integers. Show that if X has this representation then the regular continued fraction for $1/X$ is

$$1/X = B_0 + |B_1, \dots, B_m, A_5, A_6, \dots|$$

for suitable integers B_0, B_1, \dots, B_m . (The case $A_0 < 0$ is, of course, the most interesting.) Explain how to determine the B 's in terms of A_0, A_1, A_2, A_3 , and A_4 .

11. [M30] (J. Lagrange.) Let $X = A_0 + \lfloor A_1, A_2, \dots \rfloor$, $Y = B_0 + \lfloor B_1, B_2, \dots \rfloor$ be the regular continued fraction representations of two real numbers X and Y , in the sense of exercise 10. Show that these representations ~~eventually agree~~ in the sense that $A_{m+k} = B_{n+k}$ for some m and n and for all $k \geq 0$, if and only if we have $X = (qY + r)/(sY + t)$ for some integers q, r, s, t with $|qt - rs| = 1$. (This theorem is the analog, for continued fraction representations, of the simple result that the representations of X and Y in the decimal system eventually agree if and only if ~~they are equal~~ for some integers q, r , and s .)

► 12. [M30] A quadratic irrationality is a number of the form $(\sqrt{D} - U)/V$, where D , U , and V are integers, $D > 0$, $V \neq 0$, and D is not a perfect square. We may assume without loss of generality that V is a divisor of $D - U^2$, for otherwise the number may be rewritten as $(\sqrt{DV^2} - U|V|)/V|V|$.

- a) Prove that the regular continued fraction expansion (in the sense of exercise 10) of a quadratic irrationality $X = (\sqrt{D} - U)/V$ is obtained by the following formulas:

$$\begin{aligned} V_0 &= V, & A_0 &= \lfloor X \rfloor, & U_0 &= U + A_0 V; \\ V_{n+1} &= (D - U_n^2)/V_n, & A_{n+1} &= \lfloor (\sqrt{D} + U_n)/V_{n+1} \rfloor, \\ U_{n+1} &= A_{n+1} V_{n+1} - U_n. \end{aligned}$$

[Note: An algorithm based on this process has many applications to the solution of quadratic equations in integers; see, for example, H. Davenport, *The Higher Arithmetic* (London: Hutchinson, 1952); W. J. LeVeque, *Topics in Number Theory* (Reading, Mass.: Addison-Wesley, 1956); and see also Section 4.5.4. By exercise 1.2.4-35, we have $A_{n+1} = \lfloor (\lfloor \sqrt{D} \rfloor + U_n)/V_{n+1} \rfloor$ when $V_{n+1} > 0$, and $A_{n+1} = \lfloor (\lfloor \sqrt{D} \rfloor + 1 + U_n)/V_{n+1} \rfloor$ when $V_{n+1} < 0$; hence such an algorithm need only work with the positive integer $\lfloor \sqrt{D} \rfloor$.]

b) Prove that $0 < U_n < \sqrt{D}$, $0 < V_n < 2\sqrt{D}$, for all $n > N$, where N is some integer depending on X ; hence the regular continued fraction representation of every quadratic irrationality is eventually periodic. [Hint: Show that $(-\sqrt{D} - U)/V = A_0 + \lfloor A_1, \dots, A_n, -V_n/(\sqrt{D} + U_n) \rfloor$, and use Eq. (5) to prove that $(\sqrt{D} + U_n)/V_n$ is positive when n is large.]

c) Letting $p_n = Q_{n+1}(A_0, A_1, \dots, A_n)$ and $q_n = Q_n(A_1, \dots, A_n)$, prove the identity $V p_n^2 + 2U p_n q_n + ((U^2 - D)/V) q_n^2 = (-1)^{n+1} V_{n+1}$.

d) Prove that the regular continued fraction representation of an irrational number X is eventually periodic if and only if X is a quadratic irrationality. (This is the continued fraction analog of the fact that the decimal expansion of a real number X is eventually periodic if and only if X is rational.)

13. [M40] (J. Lagrange, 1797.) Let $f(x) = a_n x^n + \dots + a_0$, $a_n > 0$, be a polynomial with integer coefficients, having no rational roots, and having exactly one real root $\xi > 1$. Design a computer program to find the first thousand or so partial quotients of ξ , using the following algorithm (which essentially involves only addition):

L1. Set $A \leftarrow 1$.

L2. For $k = 0, 1, \dots, n-1$ (in this order) and for $j = n-1, \dots, k$ (in this order), set $a_j \leftarrow a_{j+1} + a_j$. (This step replaces $f(x)$ by $g(x) = f(x+1)$, a polynomial whose roots are one less than those of f .)

L3. If $a_n + a_{n-1} + \dots + a_0 < 0$, set $A \leftarrow A + 1$ and return to L2.

- L4.** Output A (which is the value of the next partial quotient). Replace the coefficients $(a_n, a_{n-1}, \dots, a_0)$ by $(-a_0, -a_1, \dots, -a_n)$ and return to L1. (This step replaces $f(x)$ by a polynomial whose roots are reciprocals of those of f .)

For example, starting with $f(x) = x^3 - 2$, the algorithm will output “1” (changing $f(x)$ to $x^3 - 3x^2 - 3x - 1$); then “3” (changing $f(x)$ to $10x^3 - 6x^2 - 6x - 1$); etc.

- 14. [M22]** (A. Hurwitz, 1891.) Show that the following rules make it possible to find the regular continued fraction expansion of $2X$, given the partial quotients of X :

$$\begin{aligned} 2\lceil 2a, b, c, \dots \rceil &= \lceil a, 2b + 2\lceil c, \dots \rceil \rceil; \\ 2\lceil 2a + 1, b, c, \dots \rceil &= \lceil a, 1, 1 + 2\lceil b - 1, c, \dots \rceil \rceil. \end{aligned}$$

Use this idea to find the regular continued fraction expansion of $\frac{1}{2}e$, given the expansion of e in (13).

- **15. [M31]** (R. W. Gosper.) Generalizing exercise 14, design an algorithm that computes the continued fraction $X_0 + \lceil X_1, X_2, \dots \rceil$ for $(ax + b)/(cx + d)$, given the continued fraction $x_0 + \lceil x_1, x_2, \dots \rceil$ for x , and given integers a, b, c, d with $ad \neq bc$. Make your algorithm an “on-line coroutine” that outputs as many X_k as possible before inputting each x_j . Demonstrate how your algorithm computes $(97x + 39)/(-62x - 25)$ when $x = -1 + \lceil 5, 1, 1, 1, 2, 1, 2 \rceil$.

- 16. [HM30]** (L. Euler, 1731.) Let $f_0(z) = (e^z - e^{-z})/(e^z + e^{-z}) = \tanh z$, and let $f_{n+1}(z) = 1/f_n(z) - (2n + 1)/z$. Prove that, for all n , $f_n(z)$ is an analytic function of the complex variable z in a neighborhood of the origin, and it satisfies the differential equation $f'_n(z) = 1 - f_n(z)^2 - 2nf_n(z)/z$. Use this fact to prove that

$$\tanh z = \lceil z^{-1}, 3z^{-1}, 5z^{-1}, 7z^{-1}, \dots \rceil.$$

Then apply Hurwitz’s rule (exercise 14) to prove that

$$e^{-1/n} = \lceil 1, (2m + 1)n - 1, 1 \rceil, \quad m \geq 0.$$

(This notation denotes the infinite continued fraction $\lceil 1, n - 1, 1, 1, 3n - 1, 1, 1, 5n - 1, 1, \dots \rceil$.) Also find the regular continued fraction expansion of $e^{-2/n}$ when $n > 0$ is odd.

- **17. [M29]** (a) Prove that $\lceil x_1, -x_2 \rceil = \lceil x_1 - 1, 1, x_2 - 1 \rceil$. (b) Generalize this identity, obtaining a formula for $\lceil x_1, -x_2, x_3, -x_4, \dots, x_{2n-1}, -x_{2n} \rceil$ in which all partial quotients are positive integers when the x ’s are large positive integers. (c) The result of exercise 16 implies that $\tan 1 = \lceil 1, -3, 5, -7, \dots \rceil$. Find the regular continued fraction expansion of $\tan 1$.

- 18. [M40]** Develop a computer program to find as many partial quotients of x as possible, when x is a real number given with high precision. Use your program to calculate the first one thousand or so partial quotients of Euler’s constant γ , based on D. W. Sweeney’s 3566-place value [Math. Comp. 17 (1963), 170–178]. (According to the theory in the text, we expect to get about 0.97 partial quotients per decimal digit. Cf. Algorithm 4.5.2L and the article by J. W. Wrench, Jr. and D. Shanks, Math. Comp. 20 (1966), 444–447.)

- 19. [M20]** Prove that $F(x) = \log_b(1 + x)$ satisfies Eq. (24).

- 20. [HM20]** Derive (36) from (35).

21. [HM29] (E. Wirsing.) The bounds (37) were obtained for a function φ corresponding to g with $Tg(x) = 1/(x+1)$. Show that the function corresponding to $Tg(x) = 1/(x+c)$ yields better bounds, when $c > 0$ is an appropriate constant.

22. [HM46] (K. I. Babenko.) Develop efficient means to calculate accurate approximations to the quantities λ_j and $\Psi_j(x)$ in (42), for small $j \geq 3$ and for $0 \leq x \leq 1$.

23. [HM23] Prove (51), using results from the proof of Theorem W.

24. [M22] What is the average value of a partial quotient A_n in the regular continued fraction expansion of a random real number?

25. [HM25] Find an example of a set $I = I_1 \cup I_2 \cup I_3 \cup \dots \subseteq [0, 1]$, where the I 's are disjoint intervals, for which (43) does not hold.

26. [M23] Show that if the numbers $\{1/n, 2/n, \dots, \lfloor n/2 \rfloor /n\}$ are expressed as regular continued fractions, the result is symmetric between left and right, in the sense that $|A_t, \dots, A_2, A_1|$ appears whenever $|A_1, A_2, \dots, A_t|$ does.

27. [M21] Derive (53) from (47) and (52).

28. [M23] Prove the following identities involving the three number-theoretic functions $\varphi(n)$, $\mu(n)$, $\Lambda(n)$:

$$\begin{aligned} \text{a)} \sum_{d \mid n} \mu(d) &= \delta_{n1}. & \text{b)} \ln n &= \sum_{d \mid n} \Lambda(d), & n &= \sum_{d \mid n} \varphi(d). \\ \text{c)} \Lambda(n) &= \sum_{d \mid n} \mu\left(\frac{n}{d}\right) \ln d, & \varphi(n) &= \sum_{d \mid n} \mu\left(\frac{n}{d}\right) d. \end{aligned}$$

29. [M23] Assuming that T_n is given by (53), show that (55) equals (56).

► **30.** [HM32] The following variant of Euclid's algorithm is often suggested: Instead of replacing v by $u \bmod v$ during the division step, replace it by $|(u \bmod v) - v|$ if $u \bmod v > \frac{1}{2}v$. Thus, for example, if $u = 26$ and $v = 7$, we have $\gcd(26, 7) = \gcd(-2, 7) = \gcd(7, 2)$; -2 is the *remainder of smallest magnitude* when multiples of 7 are subtracted from 26. Compare this procedure with Euclid's algorithm; estimate the number of division steps this method saves, on the average.

► **31.** [M35] Find the “worst case” of the modification of Euclid's algorithm suggested in exercise 30; what are the smallest inputs $u > v > 0$ that require n division steps?

32. [20] (a) A Morse code sequence of length n is a string of r dots and s dashes, where $r + 2s = n$. For example, the Morse code sequences of length 4 are

$$\dots, \quad \cdots, \quad \cdots, \quad \cdots, \quad \cdots.$$

Noting that the continuant $Q_4(x_1, x_2, x_3, x_4)$ is $x_1x_2x_3x_4 + x_1x_2 + x_1x_4 + x_3x_4 + 1$, find and prove a simple relation between $Q_n(x_1, \dots, x_n)$ and Morse code sequences of length n . (b) (L. Euler, *Novi Comm. Acad. Sci. Pet.* 9 (1762), 53–69.) Prove that

$$\begin{aligned} Q_{m+n}(x_1, \dots, x_{m+n}) &= Q_m(x_1, \dots, x_m)Q_n(x_{m+1}, \dots, x_{m+n}) \\ &\quad + Q_{m-1}(x_1, \dots, x_{m-1})Q_{n-1}(x_{m+2}, \dots, x_{m+n}). \end{aligned}$$

33. [M32] Let $h(n)$ be the number of representations of n in the form

$$n = xx' + yy', \quad x > y > 0, \quad x' > y' > 0, \quad \gcd(x, y) = 1, \quad \text{integer } x, x', y, y'.$$

(a) Show that if the conditions are relaxed to allow $x' = y'$, the number of representations is $h(n) + \lfloor (n-1)/2 \rfloor$. (b) Show that for fixed $y > 0$ and $0 < t \leq y$, where $\gcd(t, y) = 1$, and for each fixed x' in the range $0 < x' < n/(y+t)$ such that $x't \equiv n$ (modulo y), there is exactly one representation of n satisfying the restrictions of (a) and the condition $x \equiv t$ (modulo y). (c) Consequently

$$h(n) = \sum \left[\left(\frac{n}{y+t} - t' \right) \frac{1}{y} \right] - \lfloor (n-1)/2 \rfloor,$$

where the sum is over all positive integers y, t, t' such that $\gcd(t, y) = 1, t \leq y, t' \leq y, tt' \equiv n$ (modulo y). (d) Show that each of the $h(n)$ representations can be expressed uniquely in the form

$$\begin{aligned} x &= Q_m(x_1, \dots, x_m), & y &= Q_{m-1}(x_1, \dots, x_{m-1}), \\ x' &= Q_k(x_{m+1}, \dots, x_{m+k})d, & y' &= Q_{k-1}(x_{m+2}, \dots, x_{m+k})d, \end{aligned}$$

where m, k, d , and the x_j are positive integers with $x_1 \geq 2, x_{m+k} \geq 2$, and d is a divisor of n . The identity of exercise 32 now implies that $n/d = Q_{m+k}(x_1, \dots, x_{m+k})$. Conversely, any given sequence of positive integers x_1, \dots, x_{m+k} such that $x_1 \geq 2, x_{m+k} \geq 2$, and $Q_{m+k}(x_1, \dots, x_{m+k})$ divides n , corresponds in this way to $m+k-1$ representations of n . (e) Therefore $nT_n = \lfloor (5n-3)/2 \rfloor + 2h(n)$.

34. [HM40] (H. Heilbronn.) (a) Let $h_d(n)$ be the number of representations of n as in exercise 33 such that $xd < x'$, plus half the number of representations with $xd = x'$. Let $g(n)$ be the number of representations without the requirement that $\gcd(x, y) = 1$. Prove that

$$h(n) = \sum_{d|n} \mu(d)g\left(\frac{n}{d}\right), \quad g(n) = 2 \sum_{d|n} h_d\left(\frac{n}{d}\right).$$

(b) Generalizing exercise 33(b), show that for $d \geq 1$, $h_d(n) = \sum(n/(y(y+t))) + O(n)$, where the sum is over all integers y and t such that $\gcd(t, y) = 1$ and $0 < t \leq y < \sqrt{n/d}$. (c) Show that $\sum_{1 \leq y \leq n} (y/(y+t)) = \varphi(y) \ln 2 + O(\sigma_{-1}(y))$, where the sum is over the range $0 < t \leq y, \gcd(t, y) = 1$; and where $\sigma_{-1}(y) = \sum_{d|y} (1/d)$. (d) Show that $\sum_{1 \leq y \leq n} \varphi(y)/y^2 = \sum_{1 \leq d \leq n} \mu(d)H_{\lfloor n/d \rfloor}/d^2$. (e) Hence we have the asymptotic formula $T_n = ((12 \ln 2)/\pi^2)(\ln n - \sum_{d|n} \Lambda(d)/d) + O(\sigma_{-1}(n)^2)$.

35. [HM41] (A. C. Yao and D. E. Knuth.) Prove that the sum of all partial quotients for the fractions m/n , for $1 \leq m < n$, is equal to $2(\sum \lfloor x/y \rfloor + \lfloor n/2 \rfloor)$, where the sum is over all representations $n = xx' + yy'$ satisfying the conditions of exercise 33(a). Show that $\sum \lfloor x/y \rfloor = 3\pi^{-2}n(\ln n)^2 + O(n \log n (\log \log n)^2)$, and apply this to the “ancient” form of Euclid’s algorithm that uses only subtraction instead of division.

36. [M35] (G. H. Bradley.) What is the smallest value of u_n such that the calculation of $\gcd(u_1, \dots, u_n)$ by steps C1 and C2 in Section 4.5.2 requires N divisions, if Euclid’s algorithm is used throughout? Assume that $N \geq n$.

37. [M38] (T. S. Motzkin and E. G. Straus.) Let a_1, \dots, a_n be positive integers. Show that $\max Q_n(a_{p(1)}, \dots, a_{p(n)})$, over all permutations $p(1) \dots p(n)$ of $\{1, 2, \dots, n\}$, occurs when $a_{p(1)} \geq a_{p(n)} \geq a_{p(2)} \geq a_{p(n-1)} \geq \dots$; and the minimum occurs when $a_{p(1)} \leq a_{p(n)} \leq a_{p(3)} \leq a_{p(n-2)} \leq a_{p(5)} \leq \dots \leq a_{p(6)} \leq a_{p(n-3)} \leq a_{p(4)} \leq a_{p(n-1)} \leq a_{p(2)}$.

38. [M25] (J. Mikusiński.) Let $K(n) = \max_{m \geq 0} T(m, n)$. Theorem F shows that $K(n) \leq \lfloor \log_\phi(\sqrt{5}n + 1) \rfloor - 2$; prove that $K(n) \geq \frac{1}{2}\lceil \log_\phi(\sqrt{5}n + 1) \rceil - 2$.

► **39.** [M25] (R. W. Gosper.) If a baseball player's batting average is .334, what is the fewest possible number of times he has been at bat? [Note for non-baseball-fans: Batting average = (number of hits)/(times at bat), rounded to three decimal places.]

► **40.** [M28] (The Stern–Peirce tree.) Consider an infinite binary tree in which each node is labeled with the fraction $(p_l + p_r)/(q_l + q_r)$, where p_l/q_l is the label of the node's nearest left ancestor and p_r/q_r is the label of the node's nearest right ancestor. (A left ancestor is one that precedes a node in symmetric order, while a right ancestor follows the node. See Section 2.3.1 for the definition of symmetric order.) If the node has no left ancestors, $p_l/q_l = 0/1$; if it has no right ancestors, $p_r/q_r = 1/0$. Thus the label of the root is $1/1$; the labels of its two sons are $1/2$ and $2/1$; the labels of the four nodes on level 2 are $1/3, 2/3, 3/2, \text{ and } 3/1$, from left to right; the labels of the eight nodes on level 3 are $1/4, 2/5, 3/5, 3/4, 4/3, 5/3, 5/2, 4/1$; and so on.

Prove that p is relatively prime to q in each label p/q ; furthermore, the node labeled p/q precedes the node labeled p'/q' in symmetric order if and only if the labels satisfy $p/q < p'/q'$. Find a connection between the continued fraction for the label of a node and the path to that node, thereby showing that each positive rational number appears as the label of exactly one node in the tree.

[M40] (J. Shallit, 1979.) Show that the regular continued fraction expansion of

$$\frac{1}{2^1} + \frac{1}{2^3} + \frac{1}{2^7} + \dots = \sum_{n \geq 1} \frac{1}{2^{2^n-1}}$$

contains only 1's and 2's and has a fairly [redacted]! Prove that the partial quotients of Liouville's numbers $\sum_{n \geq 1} l^{-n}$ also have a regular pattern, when l is any integer ≥ 2 . [The latter numbers, introduced by J. Liouville in *J. de Math. Pures et Appl.* 16 (1851), 133–142, were the first explicitly defined numbers to be proved transcendental. The former number and similar constants were first proved transcendental by A. J. Kempner, *Trans. Amer. Math. Soc.* 17 (1916), 476–482.]

42. [M30] (J. Lagrange, 1798.) Let X have the regular continued fraction expansion $/A_1, A_2, \dots, /$, and let $q_n = Q_n(A_1, \dots, A_n)$. Let $\|x\|$ denote the distance from x to the nearest integer, namely $\min_p |x - p|$. Show that $\|qx\| \geq \|q_{n-1}x\|$ for $1 \leq q < q_n$. (Thus the denominators q_n of the so-called convergents $p_n/q_n = /A_1, \dots, A_n/$ are the “record-breaking” integers that make $\|qx\|$ achieve new lows.)

43. [M30] (D. W. Matula.) Show that the “mediant rounding” rule for fixed-slash or floating-slash numbers, Eq. 4.5.1–1, can be implemented simply as follows, when the number $x > 0$ is not representable: Let the regular continued fraction expansion of x be $a_0 + /a_1, a_2, \dots, /$, and let $p_n = Q_{n+1}(a_0, \dots, a_n)$, $q_n = Q_n(a_1, \dots, a_n)$. Then $\text{round}(x) = (p_i/q_i)$, where (p_i/q_i) is representable but (p_{i+1}/q_{i+1}) is not. [Hint: See exercise 40.]

44. [M25] Suppose we are doing fixed slash arithmetic with mediant rounding, where the fraction (u/u') is representable if and only if $|u| < M$ and $0 \leq u' < N$ and $\gcd(u, u') = 1$. Prove or disprove the identity $((u/u') \oplus (v/v')) \ominus (v/v') = (u/u')$ for all representable (u/u') and (v/v') , provided that $u' < \sqrt{N}$ and no overflow occurs.

45. [HM48] Develop the analysis of algorithms for computing the greatest common divisor of three or more integers.

4.5.4. Factoring into Primes

Several of the computational methods we have encountered in this book rest on the fact that every positive integer n can be expressed in a unique way in the form

$$n = p_1 p_2 \dots p_t, \quad p_1 \leq p_2 \leq \dots \leq p_t, \quad (1)$$

where each p_k is prime. (When $n = 1$, this equation holds for $t = 0$.) It is unfortunately not a simple matter to find this prime factorization of n , or to determine whether or not n is prime. So far as anyone knows, it is a great deal harder to factor a large number n than to compute the greatest common divisor of two large numbers m and n ; therefore we should avoid factoring large numbers whenever possible. But several ingenious ways to speed up the factoring process have been discovered, and we will now investigate some of them.

Divide and factor. First let us consider the most obvious algorithm for factorization: If $n > 1$, we can divide n by successive primes $p = 2, 3, 5, \dots$ until discovering the smallest p for which $n \bmod p = 0$. Then p is the smallest prime factor of n , and the same process may be applied to $n \leftarrow n/p$ in an attempt to divide this new value of n by p and by higher primes. If at any stage we find that $n \bmod p \neq 0$ but $\lfloor n/p \rfloor \leq p$, we can conclude that n is prime; for if n is not prime, then by (1) we must have $n \geq p_1^2$, but $p_1 > p$ implies that $p_1^2 \geq (p+1)^2 > p(p+1) > p^2 + (n \bmod p) \geq \lfloor n/p \rfloor p + (n \bmod p) = n$. This leads us to the following procedure:

Algorithm A (Factoring by division). Given a positive integer N , this algorithm finds the prime factors $p_1 \leq p_2 \leq \dots \leq p_t$ of N as in Eq. (1). The method makes use of an auxiliary sequence of “trial divisors”

$$2 = d_0 < d_1 < d_2 < d_3 < \dots, \quad (2)$$

which includes all prime numbers $\leq \sqrt{N}$ (and which may also include values that are *not* prime, if it is convenient to do so). The sequence of d 's must also include at least one value such that $d_k \geq \sqrt{N}$.

A1. [Initialize.] Set $t \leftarrow 0$, $k \leftarrow 0$, $n \leftarrow N$. (During this algorithm the variables t , k , n are related by the following condition: “ $n = N/p_1 \dots p_t$, and n has no prime factors less than d_k .”)

A2. [$n = 1?$] If $n = 1$, the algorithm terminates.

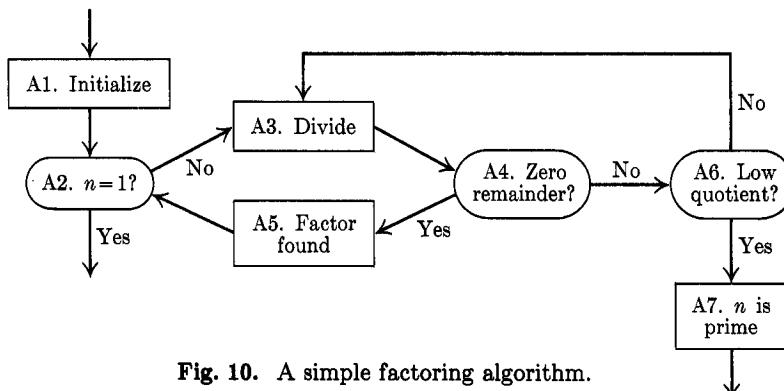


Fig. 10. A simple factoring algorithm.

A3. [Divide.] Set $q \leftarrow \lfloor n/d_k \rfloor$, $r \leftarrow n \bmod d_k$. (Here q and r are the quotient and remainder obtained when n is divided by d_k .)

A4. [Zero remainder?] If $r \neq 0$, go to step A6.

A5. [Factor found.] Increase t by 1, and set $p_t \leftarrow d_k$, $n \leftarrow q$. Return to step A2.

A6. [Low quotient?] If $q > d_k$, increase k by 1 and return to step A3.

A7. [n is prime.] Increase t by 1, set $p_t \leftarrow n$, and terminate the algorithm. ■

As an example of Algorithm A, consider the factorization of the number $N = 25852$. We immediately find that $N = 2 \cdot 12926$; hence $p_1 = 2$. Furthermore, $12926 = 2 \cdot 6463$, so $p_2 = 2$. But now $n = 6463$ is not divisible by 2, 3, 5, ..., 19; we find that $n = 23 \cdot 281$, hence $p_3 = 23$. Finally $281 = 12 \cdot 23 + 5$ and $12 \leq 23$; hence $p_4 = 281$. The determination of 25852's factors has therefore involved a total of 12 division operations; on the other hand, if we had tried to factor the slightly smaller number 25849 (which is prime), at least 38 division operations would have been performed. This illustrates the fact that Algorithm A requires a running time roughly proportional to $\max(p_{t-1}, \sqrt{p_t})$. (If $t = 1$, this formula is valid if we adopt the convention $p_0 = 1$.)

The sequence d_0, d_1, d_2, \dots of trial divisors used in Algorithm A can be taken to be simply 2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, ..., where we alternately add 2 and 4 after the first three terms. This sequence contains all numbers that are not multiples of 2 or 3; it also includes numbers such as 25, 35, 49, etc., which are not prime, but the algorithm will still give the correct answer. A further savings of 20 percent in computation time can be made by removing the numbers $30m \pm 5$ from the list for $m \geq 1$, thereby eliminating all of the spurious multiples of 5. The exclusion of multiples of 7 shortens the list by 14 percent more, etc. A compact bit table can be used to govern the choice of trial divisors.

If N is known to be small, it is reasonable to have a table of all the necessary primes as part of the program. For example, if N is less than a million, we need only include the 168 primes less than a thousand (followed by the value $d_{168} = 1000$, to terminate the list in case N is a prime larger than 997^2). Such a table

can be set up by means of a short auxiliary program, which builds the table just after the factoring program has been loaded into the computer; see Algorithm 1.3.2P, or see exercise 8.

How many trial divisions are necessary in Algorithm A? Let $\pi(x)$ be the number of primes $\leq x$, so that $\pi(2) = 1$, $\pi(10) = 4$; the asymptotic behavior of this function has been studied extensively by many of the world's greatest mathematicians, beginning with Legendre in 1798. Numerous advances made during the nineteenth century culminated in 1899, when Charles de la Vallée Poussin proved that, for some $A > 0$,

$$\pi(x) = \int_2^x \frac{dt}{\ln t} + O(xe^{-A\sqrt{\log x}}). \quad (3)$$

[Mém. Couronnés Acad. Roy. Belgique 59 (1899), 1–74.] Integrating by parts yields

$$\pi(x) = \frac{x}{\ln x} + \frac{x}{(\ln x)^2} + \frac{2! x}{(\ln x)^3} + \cdots + \frac{r! x}{(\ln x)^{r+1}} + O\left(\frac{x}{(\log x)^{r+2}}\right) \quad (4)$$

for all fixed $r \geq 0$. The error term in (3) has subsequently been improved; for example, it can be replaced by $O(x \exp(-A(\log x)^{3/5}/(\log \log x)^{1/5}))$. [See A. Walfisz, *Weyl'sche Exponentialsummen in der neueren Zahlentheorie* (Berlin, 1963), Chapter 5.] Bernhard Riemann conjectured in 1859 that

$$\pi(x) = \sum_{k \geq 1} \mu(k) L(\sqrt[k]{x})/k + O(1) = L(x) - \frac{1}{2}L(\sqrt{x}) - \frac{1}{3}L(\sqrt[3]{x}) + \cdots \quad (5)$$

where $L(x) = \int_2^x dt/\ln t$, and his formula agrees well with actual counts when x is of reasonable size. For example, we have the following table:

x	$\pi(x)$	$x/\ln x$	$L(x)$	Riemann's formula
10^3	168	144.8	176.6	168.36
10^6	78498	72382.4	78626.5	78527.40
10^9	50847534	48254942.4	50849233.9	50847455.43

However, the distribution of large primes is not that simple, and Riemann's conjecture (5) was disproved by J. E. Littlewood in 1914; see Hardy and Littlewood, *Acta Math.* 41 (1918), 119–196, where it is shown that there is a positive constant C such that $\pi(x) > L(x) + C\sqrt{x} \log \log x / \log x$ for infinitely many x . Littlewood's result shows that prime numbers are inherently somewhat mysterious, and it will be necessary to develop deep properties of mathematics before their distribution is really understood. Riemann made another much more plausible conjecture, the famous "Riemann hypothesis," which states that the complex function $\zeta(z)$ is zero only when the real part of z is equal to $\frac{1}{2}$, except in the trivial cases where z is a negative even integer. This hypothesis, if true,

would imply that $\pi(x) = L(x) + O(\sqrt{x} \log x)$; see exercise 25. Richard Brent has used a method of D. H. Lehmer to verify Riemann's hypothesis computationally for all "small" values of z , by showing that $\zeta(z)$ has exactly 75,000,000 zeros whose imaginary part is in the range $0 < \Im z < 32585736.4$; all of these zeros have $\Re z = \frac{1}{2}$ and $\zeta'(z) \neq 0$. [Math. Comp. 33 (1979), 1361–1372.]

In order to analyze the average behavior of Algorithm A, we would like to know how large the largest prime factor p_t will tend to be. This question was first investigated by Karl Dickman [Arkiv för Mat., Astron. och Fys. 22A, 10 (1930), 1–14], who studied the probability that a random integer between 1 and x will have its largest prime factor $\leq x^\alpha$. Dickman gave a heuristic argument to show that this probability approaches the limiting value $F(\alpha)$ as $x \rightarrow \infty$, where F can be calculated from the functional equation

$$F(\alpha) = \int_0^\alpha F\left(\frac{t}{1-t}\right) \frac{dt}{t}, \quad \text{for } 0 \leq \alpha \leq 1; \quad F(\alpha) = 1 \quad \text{for } \alpha \geq 1. \quad (6)$$

His argument was essentially this: Given $0 < t < 1$, the number of integers less than x whose largest prime factor is between x^t and x^{t+dt} is $x F'(t) dt$. The number of primes p in that range is $\pi(x^{t+dt}) - \pi(x^t) = \pi(x^t + (\ln x)x^t dt) - \pi(x^t) = x^t dt/t$. For every such p , the number of integers n such that " $np \leq x$ and the largest prime factor of n is $\leq p$ " is the number of $n \leq x^{1-t}$ whose largest prime factor is $\leq (x^{1-t})^{t/(1-t)}$, namely $x^{1-t} F(t/(1-t))$. Hence $x F'(t) dt = (x^t dt/t)(x^{1-t} F(t/(1-t)))$, and (6) follows by integration. This heuristic argument can be made rigorous; V. Ramaswami [Bull. Amer. Math. Soc. 55 (1949), 1122–1127] showed that the probability in question for fixed α is asymptotically $F(\alpha) + O(1/\log x)$, as $x \rightarrow \infty$, and many other authors have extended the analysis [see the survey by Karl K. Norton, Memoirs Amer. Math. Soc. 106 (1971), 9–27].

If $\frac{1}{2} \leq \alpha \leq 1$, formula (6) simplifies to

$$F(\alpha) = 1 - \int_\alpha^1 F\left(\frac{t}{1-t}\right) \frac{dt}{t} = 1 - \int_\alpha^1 \frac{dt}{t} = 1 + \ln \alpha.$$

Thus, for example, the probability that a random positive integer $\leq x$ has a prime factor $> \sqrt{x}$ is $1 - F(\frac{1}{2}) = \ln 2$, about 69 percent. In all such cases, Algorithm A must work hard.

The net result of this discussion is that Algorithm A will give the answer rather quickly if we want to factor a six-digit number; but for large N the amount of computer time for factorization by trial division will rapidly exceed practical limits, unless we are unusually lucky.

Later in this section we will see that there are fairly good ways to determine whether or not a reasonably large number n is prime, without trying all divisors up to \sqrt{n} . Therefore Algorithm A would often run faster if we inserted a primality test between steps A2 and A3; the running time for this improved algorithm would then be roughly proportional to p_{t-1} , the second-largest prime factor of N , instead of to $\max(p_{t-1}, \sqrt{p_t})$. By an argument analogous to

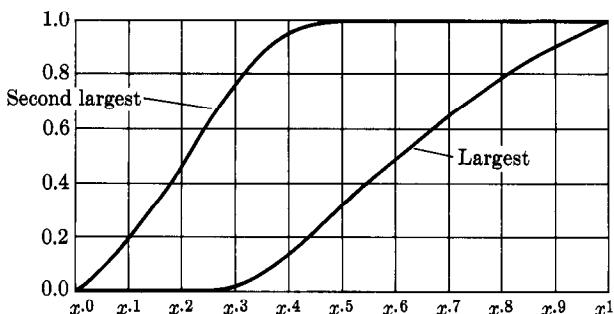


Fig. 11. Probability distribution functions for the two largest prime factors of a random integer $\leq x$.

Dickman's (see exercise 18), we can show that the second-largest prime factor of a random integer x will be $\leq x^\beta$ with approximate probability $G(\beta)$, where

$$G(\beta) = \int_0^\beta \left(G\left(\frac{t}{1-t}\right) - F\left(\frac{t}{1-t}\right) \right) \frac{dt}{t}, \quad \text{for } 0 \leq \beta \leq \frac{1}{2}. \quad (7)$$

Clearly $G(\beta) = 1$ for $\beta \geq \frac{1}{2}$. (See Fig. 11.) Numerical evaluation of (6) and (7) yields the following "percentage points":

$F(\alpha), G(\beta) =$.01	.05	.10	.20	.35	.50	.65	.80	.90	.95	.99
$\alpha =$.2697	.3348	.3785	.4430	.5220	.6065	.7047	.8187	.9048	.9512	.9900
$\beta =$.0056	.0273	.0531	.1003	.1611	.2117	.2582	.3104	.3590	.3967	.4517

Thus, the second-largest prime factor will be $\leq x^{.2117}$ about half the time, etc.

The total number of prime factors, t , has also been intensively analyzed. Obviously $1 \leq t \leq \lg N$, but these lower and upper bounds are seldom achieved. It is possible to prove that if N is chosen at random between 1 and x , the probability that $t \leq \ln \ln x + c\sqrt{\ln \ln x}$ approaches

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^c e^{-u^2/2} du \quad (8)$$

as $x \rightarrow \infty$, for any fixed c . In other words, the distribution of t is essentially normal, with mean and variance $\ln \ln x$; about 99.73 percent of all the large integers $\leq x$ have $|t - \ln \ln x| \leq 3\sqrt{\ln \ln x}$. Furthermore the average value of $t - \ln \ln x$ for $1 \leq N \leq x$ is known to approach

$$\gamma + \sum_{p \text{ prime}} (\ln(1 - 1/p) + 1/(p - 1)) = 1.03465\ 38818\ 97438.$$

[Cf. G. H. Hardy and E. M. Wright, *Introduction to the Theory of Numbers*, 4th ed. (Oxford, 1960), §22.11; see also D. E. Knuth and L. Trabb Pardo, *Theoretical Comp. Sci.* 3 (1976), 321–348.]

The size of prime factors has a remarkable connection with permutations: The average number of bits in the k th largest prime factor of a random n -bit integer is asymptotically the same as the average length of the k th largest cycle of a random n -element permutation, as $n \rightarrow \infty$. [See D. E. Knuth and L. Trabb Pardo, *Theoretical Comp. Sci.* 3 (1976), 321–348.] It follows that Algorithm A usually finds a few small factors and then begins a long-drawn-out search for the big ones that are left.

Factoring à la Monte Carlo. Near the beginning of Chapter 3, we observed that “a random number generator chosen at random isn’t very random.” This principle, which worked against us in that chapter, has the redeeming virtue that it leads to a surprisingly efficient method of factorization, discovered by J. M. Pollard [*BIT* 15 (1975), 331–334]. The number of computational steps in Pollard’s method is on the order of $\sqrt{p_{t-1}}$, so it is significantly faster than Algorithm A when N is large. According to (7) and Fig. 11, the running time will usually be well under $N^{1/4}$.

Let $f(x)$ be any polynomial with integer coefficients, and consider the two sequences defined by

$$x_0 = y_0 = A; \quad x_{m+1} = f(x_m) \bmod N, \quad y_{m+1} = f(y_m) \bmod p, \quad (9)$$

where p is any prime factor of N . It follows that

$$y_m = x_m \bmod p, \quad \text{for } m \geq 1. \quad (10)$$

Now exercise 3.1–7 shows that we will have $y_m = y_{l(m)-1}$ for some $m \geq 1$, where $l(m)$ is the greatest power of 2 that is $\leq m$. Thus $x_m - x_{l(m)-1}$ will be a multiple of p . Furthermore if $f(y) \bmod p$ behaves as a random mapping from the set $\{0, 1, \dots, p-1\}$ into itself, exercise 3.1–12 shows that the average value of the least such m will be of order \sqrt{p} . In fact, exercise 4 below shows that this average value for random mappings is less than $1.625 Q(p)$, where the function $Q(p) \approx \sqrt{\pi p}/2$ was defined in Section 1.2.11.3. If the different prime divisors of N correspond to different values of m (as they almost surely will, when N is large), we will be able to find them by calculating $\gcd(x_m - x_{l(m)-1}, N)$ for $m = 1, 2, 3, \dots$, until the unfactored residue is prime.

From the theory in Chapter 3, we know that a linear polynomial $f(x) = ax + c$ will not be sufficiently random for our purposes. The next-simplest case is quadratic, say $f(x) = x^2 + 1$; although we don’t know that this function is sufficiently random, our lack of knowledge tends to support the hypothesis of randomness, and empirical tests show that this f does work essentially as predicted. In fact, f is probably slightly better than random, since $x^2 + 1$ takes on only $\frac{1}{2}(p+1)$ distinct values mod p . Therefore the following procedure is reasonable:

Algorithm B (*Monte Carlo factorization*). This algorithm outputs the prime factors of a given integer $N \geq 2$, with high probability, although there is a chance that it will fail.

- B1. [Initialize.] Set $x \leftarrow 5$, $x' \leftarrow 2$, $k \leftarrow 1$, $l \leftarrow 1$, $n \leftarrow N$. (During this algorithm, n is the unfactored part of N , and the variables x and x' represent the quantities $x_m \bmod n$ and $x_{l(m)-1} \bmod n$ in (9), where $f(x) = x^2 + 1$, $A = 1$, $l = l(m)$, and $k = 2l - m$.)
- B2. [Test primality.] If n is prime (see the discussion below), output n ; the algorithm terminates.
- B3. [Factor found?] Set $g \leftarrow \gcd(x' - x, n)$. If $g = 1$, go on to step B4; otherwise output g . Now if $g = n$, the algorithm terminates (and it has failed, because we know that n isn't prime). Otherwise set $n \leftarrow n/g$, $x \leftarrow x \bmod n$, $x' \leftarrow x' \bmod n$, and return to step B2. (Note that g may not be prime; this should be tested. In the rare event that g isn't prime, its prime factors probably won't be determinable with this algorithm.)
- B4. [Advance.] Set $k \leftarrow k - 1$. If $k = 0$, set $x' \leftarrow x$, $l \leftarrow 2l$, $k \leftarrow l$. Set $x \leftarrow (x^2 + 1) \bmod n$ and return to B3. ■

As an example of Algorithm B, let's try to factor $N = 25852$ again. The third execution of step B3 will output $g = 4$ (which isn't prime). After six more iterations the algorithm finds the factor $g = 23$. Algorithm B has not distinguished itself in this example, but of course it was designed to factor *big* numbers. Algorithm A takes much longer to find large prime factors, but it can't be beat when it comes to removing the small ones. In practice, we should run Algorithm A awhile before switching over to Algorithm B.

We can get a better idea of Algorithm B's prowess by considering the ten largest six-digit primes. The number of iterations, $m(p)$, that Algorithm B needs to find the factor p is given in the following table:

$p =$	999863	999883	999907	999917	999931	999953	999959	999961	999979	999983
$m(p) =$	276	409	2106	1561	1593	1091	474	1819	395	814

Experiments indicate that $m(p)$ has an average value of about $2\sqrt{p}$, and it never exceeds $12\sqrt{p}$ when $p < 1000000$. The maximum $m(p)$ for $p < 10^6$ is $m(874771) = 7685$; and the maximum of $m(p)/\sqrt{p}$ occurs when $p = 290047$, $m(p) = 6251$. According to these experimental results, almost all 12-digit numbers can be factored in fewer than 2000 iterations of Algorithm B (compared to roughly 100,000 divisions in Algorithm A).

The time-consuming operations in each iteration of Algorithm B are the multiple-precision multiplication and division in step B4, and the gcd in step B3. If the gcd operation is slow, Pollard suggests gaining speed by accumulating the product mod n of, say, ten consecutive $(x' - x)$ values before taking each gcd; this replaces 90 percent of the gcd operations by a single multiplication and division while only slightly increasing the chance of failure. He also suggests starting with $m = q$ instead of $m = 1$ in step B1, where q is, say, $\frac{1}{10}$ the number of iterations you are planning to use.

In those rare cases where failure occurs for large N , we could try using $f(x) = x^2 + c$ for some $c \neq 0$ or 1. The value $c = -2$ should also be avoided, since the recurrence $x_{m+1} = x_m^2 - 2$ has solutions of the form $x_m = r^{2^m} + r^{-2^m}$. Other values of c do not seem to lead to simple relationships mod p , and they should all be satisfactory when used with suitable starting values.

Richard Brent used a modification of Algorithm B in 1980 to discover the prime factor 1238926361552897 of $2^{256} + 1$.

Fermat's method. Another approach to the factoring problem, which was used by Pierre de Fermat in 1643, is more suited to finding large factors than small ones. [Fermat's original description of his method, translated into English, can be found in L. E. Dickson's monumental *History of the Theory of Numbers* 1 (New York: Chelsea, 1952), 357.]

Assume that $N = uv$, where $u \leq v$. For practical purposes we may assume that N is odd; this means that u and v are odd, and we can let

$$x = (u + v)/2, \quad y = (v - u)/2, \quad (11)$$

$$N = x^2 - y^2, \quad 0 \leq y < x \leq N. \quad (12)$$

Fermat's method consists of searching systematically for values of x and y that satisfy Eq. (12). The following algorithm shows how factoring can therefore be done without using any division:

Algorithm C (Factoring by addition and subtraction). Given an odd number N , this algorithm determines the largest factor of N less than or equal to \sqrt{N} .

C1. [Initialize.] Set $x' \leftarrow 2\lfloor\sqrt{N}\rfloor + 1$, $y' \leftarrow 1$, $r \leftarrow \lfloor\sqrt{N}\rfloor^2 - N$. (During this algorithm x' , y' , r correspond respectively to $2x + 1$, $2y + 1$, $x^2 - y^2 - N$ as we search for a solution to (12); we will have $|r| < x'$ and $y' < x'$.)

C2. [Test r .] If $r \leq 0$, go to step C4.

C3. [Step y .] Set $r \leftarrow r - y'$, $y' \leftarrow y' + 2$, and return to C2.

C4. [Done?] If $r = 0$, the algorithm terminates; we have

$$N = ((x' - y')/2)((x' + y' - 2)/2),$$

and $(x' - y')/2$ is the largest factor of N less than or equal to \sqrt{N} .

C5. [Step x .] Set $r \leftarrow r + x'$, $x' \leftarrow x' + 2$, and return to C3. ■

The reader may find it amusing to find the factors of 377 by hand, using this algorithm. The number of steps needed to find the factors u and v of $N = uv$ is essentially proportional to $(x' + y' - 2)/2 - \lfloor\sqrt{N}\rfloor = v - \lfloor\sqrt{N}\rfloor$; this can, of course, be a very large number, although each step can be done very rapidly on most computers. An improvement that requires only $O(N^{1/3})$ operations in the worst case has been developed by R. S. Lehman [*Math. Comp.* 28 (1974), 637–646].

It is not quite correct to call Algorithm C "Fermat's method," since Fermat used a somewhat more streamlined approach. Algorithm C's main loop is quite

fast on computers, but it is not very suitable for hand calculation. Fermat actually did not keep the running value of y ; he would look at $x^2 - N$ and tell whether or not this quantity was a perfect square by looking at its least significant digits. (The last two digits of a perfect square must be 00, e1, e4, 25, o6, or e9, where e is an even digit and o is an odd digit.) Therefore he avoided the operations of steps C2 and C3, replacing them by an occasional determination that a certain number is not a perfect square.

Fermat's method of looking at the rightmost digits can, of course, be generalized by using other moduli. Suppose for clarity that $N = 11111$, and consider the following table:

m	if $x \bmod m$ is	$\text{then } x^2 \bmod m$ is	and $(x^2 - N) \bmod m$ is
3	0, 1, 2	0, 1, 1	1, 2, 2
5	0, 1, 2, 3, 4	0, 1, 4, 4, 1	4, 0, 3, 3, 0
7	0, 1, 2, 3, 4, 5, 6	0, 1, 4, 2, 2, 4, 1	5, 6, 2, 0, 0, 2, 6
8	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 4, 1, 0, 1, 4, 1	1, 2, 5, 2, 1, 2, 5, 2
11	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	0, 1, 4, 9, 5, 3, 3, 5, 9, 4, 1	10, 0, 3, 8, 4, 2, 2, 4, 8, 3, 0

If $x^2 - N$ is to be a perfect square y^2 , it must have a residue mod m consistent with this fact, for all m . For example, if $N = 11111$ and $x \bmod 3 \neq 0$, then $(x^2 - N) \bmod 3 = 2$, so $x^2 - N$ cannot be a perfect square; therefore x must be a multiple of 3 whenever $11111 = x^2 - y^2$. The table tells us, in fact, that

$$\begin{aligned} x \bmod 3 &= 0; \\ x \bmod 5 &= 0, 1, \text{ or } 4; \\ x \bmod 7 &= 2, 3, 4, \text{ or } 5; \\ x \bmod 8 &= 0 \text{ or } 4 \text{ (hence } x \bmod 4 = 0\text{);} \\ x \bmod 11 &= 1, 2, 4, 7, 9, \text{ or } 10. \end{aligned} \tag{13}$$

This narrows down the search for x considerably. For example, x must be a multiple of 12. We must have $x \geq \lceil \sqrt{N} \rceil = 106$, and it is easy to verify that the first value of $x \geq 106$ that satisfies all of the conditions in (13) is $x = 144$. Now $144^2 - 11111 = 9625$, and by attempting to take the square root of 9625 we find that it is not a square. The first value of $x > 144$ that satisfies (13) is $x = 156$. In this case $156^2 - 11111 = 13225 = 115^2$; so we have found the desired solution $x = 156$, $y = 115$. This calculation shows that $11111 = 41 \cdot 271$.

The hand calculations involved in the above example are comparable to the amount of work required to divide 11111 by 13, 17, 19, 23, 29, 31, 37, and 41, even though the factors 41 and 271 are not very close to each other; thus we can see the advantages of Fermat's method.

In place of the moduli considered in (13), we can use any powers of distinct primes. For example, if we had used 25 in place of 5, we would find that the only permissible values of $x \bmod 25$ are 0, 5, 6, 10, 15, 19, and 20. This gives more information than (13). In general, we will get more information modulo p^2 than we do modulo p , for odd primes p , whenever $x^2 - N \equiv 0 \pmod{p}$ has a solution x .

The modular method just used is called a sieve procedure, since we can imagine passing all integers through a “sieve” for which only those values with $x \bmod 3 = 0$ come out, then sifting these numbers through another sieve that allows only numbers with $x \bmod 5 = 0, 1$, or 4 to pass, etc. Each sieve by itself will remove about half of the remaining values (see exercise 6); and when we sieve with respect to moduli that are relatively prime in pairs, each sieve is independent of the others because of the Chinese remainder theorem (Theorem 4.3.2C). So if we sieve with respect to, say, 30 different primes, only about one value in every 2^{30} will need to be examined to see if $x^2 - N$ is a perfect square y^2 .

Algorithm D (Factoring with sieves). Given an odd number N , this algorithm determines the largest factor of N less than or equal to \sqrt{N} . The procedure uses moduli m_1, m_2, \dots, m_r that are relatively prime to each other in pairs and relatively prime to N . We assume that r “sieve tables” $S[i, j]$ for $0 \leq j < m_i$, $1 \leq i \leq r$, have been prepared, where

$$S[i, j] = \begin{cases} 1, & \text{if } j^2 - N \equiv y^2 \pmod{m_i} \text{ has a solution } y; \\ 0, & \text{otherwise.} \end{cases}$$

- D1. [Initialize.] Set $x \leftarrow \lceil \sqrt{N} \rceil$, and set $k_i \leftarrow (-x) \bmod m_i$ for $1 \leq i \leq r$. (Throughout this algorithm the index variables k_1, k_2, \dots, k_r will be set so that $(-x) \bmod m_i = k_i$.)
- D2. [Sieve.] If $S[i, k_i] = 1$ for $1 \leq i \leq r$, go to step D4.
- D3. [Step x .] Set $x \leftarrow x + 1$, and set $k_i \leftarrow (k_i - 1) \bmod m_i$ for $1 \leq i \leq r$. Return to step D2.
- D4. [Test $x^2 - N$.] Set $y \leftarrow \lfloor \sqrt{x^2 - N} \rfloor$ or to $\lceil \sqrt{x^2 - N} \rceil$. If $y^2 = x^2 - N$, then $(x - y)$ is the desired factor, and the algorithm terminates. Otherwise return to step D3. ■

There are several ways to make this procedure run fast. For example, we have seen that if $N \bmod 3 = 2$, then x must be a multiple of 3 ; we can set $x = 3x'$, and use a different sieve corresponding to x' , increasing the speed threefold. If $N \bmod 9 = 1, 4$, or 7 , then x must be congruent respectively to $\pm 1, \pm 2$, or ± 4 (modulo 9); so we run two sieves (one for x' and one for x'' , where $x = 9x' + a$ and $x = 9x'' - a$) to increase the speed by a factor of $4\frac{1}{2}$. If $N \bmod 4 = 3$, then $x \bmod 4$ is known and the speed is increased by an additional factor of 4 ; in the other case, when $N \bmod 4 = 1$, x must be odd so the speed may be doubled. Another way to double the speed of the algorithm (at the expense of storage space) is to combine pairs of moduli, using $m_{r-k} m_k$ in place of m_k for $1 \leq k < \frac{1}{2}r$.

An even more important method of speeding up Algorithm D is to use the “Boolean operations” found on most binary computers. Let us assume, for example, that MIX is a binary computer with 30 bits per word. The tables $S[i, k_i]$ can be kept in memory with one bit per entry; thus 30 values can be stored in a single word. The operation AND, which replaces the k th bit of the accumulator

by zero if the k th bit of a specified word in memory is zero, for $1 \leq k \leq 30$, can be used to process 30 values of x at once! For convenience, we can make several copies of the tables $S[i, j]$ so that the table entries for m_i involve $\text{lcm}(m_i, 30)$ bits; then the sieve tables for each modulus fill an integral number of words. Under these assumptions, 30 executions of the main loop in Algorithm D are equivalent to code of the following form:

D2 LD1 K1	$rI1 \leftarrow k'_1.$
LDA S1, 1	$rA \leftarrow S'[1, rI1].$
DEC1 1	$rI1 \leftarrow rI1 - 1.$
J1NN **2	
INC1 M1	If $rI1 < 0$, set $rI1 \leftarrow rI1 + \text{lcm}(m_1, 30).$
ST1 K1	$k'_1 \leftarrow rI1.$
LD1 K2	$rI1 \leftarrow k'_2.$
AND S2, 1	$rA \leftarrow rA \wedge S'[2, rI1].$
DEC1 1	$rI1 \leftarrow rI1 - 1.$
J1NN **2	
INC1 M2	If $rI1 < 0$, set $rI1 \leftarrow rI1 + \text{lcm}(m_2, 30).$
ST1 K2	$k'_2 \leftarrow rI1.$
LD1 K3	$rI1 \leftarrow k'_3.$
...	(m_3 through m_r are like m_2)
ST1 Kr	$k'_r \leftarrow rI1.$
INCX 30	$x \leftarrow x + 30.$
JAZ D2	Repeat if all sieved out. ■

The number of cycles for 30 iterations is essentially $2 + 8r$; if $r = 11$, this means three cycles are being used on each iteration, just as in Algorithm C, and Algorithm C involves $y = \frac{1}{2}(v - u)$ more iterations.

If the table entries for m_i do not come out to be an integral number of words, further shifting of the table entries would be necessary on each iteration in order to align the bits properly. This would add quite a lot of coding to the main loop and it would probably make the program too slow to compete with Algorithm C unless $v/u \leq 100$ (see exercise 7).

Sieve procedures can be applied to a variety of other problems, not necessarily having much to do with arithmetic. A survey of these techniques has been prepared by Marvin C. Wunderlich, *JACM* 14 (1967), 10–19.

Special sieve machines (of reasonably low cost) have been constructed by D. H. Lehmer and his associates over a period of many years; see, for example, *AMM* 40 (1933), 401–406. Lehmer's electronic delay-line sieve, which began operating in 1965, processes one million numbers per second. Thus, each iteration of the loop in Algorithm D can be performed in one microsecond on this device. Another way to factor with sieves is described by D. H. and Emma Lehmer in *Math. Comp.* 28 (1974), 625–635.

Primality testing. None of the algorithms we have discussed so far is an efficient way to determine that a large number n is prime. Fortunately, there are other methods available for settling this question; efficient methods have been devised

by É. Lucas and others, notably D. H. Lehmer [see *Bull. Amer. Math. Soc.* 33 (1927), 327–340].

According to Fermat's theorem (Theorem 1.2.4F), we have $x^{p-1} \bmod p = 1$ whenever p is prime and x is not a multiple of p . Furthermore, there are efficient ways to calculate $x^{n-1} \bmod n$, requiring only $O(\log n)$ operations of multiplication mod n . (We shall study these in Section 4.6.3 below.) Therefore we can often determine that n is not prime when this relationship fails.

For example, Fermat once verified that the numbers $2^1 + 1$, $2^2 + 1$, $2^4 + 1$, $2^8 + 1$, and $2^{16} + 1$ are prime. In a letter to Mersenne written in 1640, Fermat conjectured that $2^{2^n} + 1$ is always prime, but said he was unable to determine definitely whether the number $4294967297 = 2^{32} + 1$ is prime or not. Neither Fermat nor Mersenne ever resolved this problem, although they could have done it as follows: The number $3^{2^{32}} \bmod (2^{32} + 1)$ can be computed by doing 32 operations of squaring modulo $2^{32} + 1$, and the answer is 3029026160; therefore (by Fermat's own theorem, which he discovered in the same year 1640!) the number $2^{32} + 1$ is not prime. This argument gives us absolutely no idea what the factors are, but it answers Fermat's question.

Fermat's theorem is a powerful test for showing non-primality of a given number. When n is not prime, it is always possible to find a value of $x < n$ such that $x^{n-1} \bmod n \neq 1$; experience shows that, in fact, such a value can almost always be found very quickly. There are some rare values of n for which $x^{n-1} \bmod n$ is frequently equal to unity, but then n has a factor less than $\sqrt[3]{n}$; see exercise 9.

The same method can be extended to prove that a large prime number n really is prime, by using the following idea: *If there is a number x for which the order of x modulo n is equal to $n - 1$, then n is prime.* (The order of x modulo n is the smallest positive integer k such that $x^k \bmod n = 1$; see Section 3.2.1.2.) For this condition implies that the numbers $x^1 \bmod n, \dots, x^{n-1} \bmod n$ are distinct and relatively prime to n , so they must be the numbers 1, 2, ..., $n - 1$ in some order; thus n has no proper divisors. If n is prime, such a number x (called a “primitive root” of n) will always exist; see exercise 3.2.1.2–16. In fact, primitive roots are rather numerous. There are $\varphi(n - 1)$ of them, and this is quite a substantial number, since $n/\varphi(n - 1) = O(\log \log n)$.

It is unnecessary to calculate $x^k \bmod n$ for all $k \leq n - 1$ to determine if the order of x is $n - 1$ or not. The order of x will be $n - 1$ if and only if

- i) $x^{n-1} \bmod n = 1$;
- ii) $x^{(n-1)/p} \bmod n \neq 1$ for all primes p that divide $n - 1$.

For $x^s \bmod n = 1$ if and only if s is a multiple of the order of x modulo n . If the two conditions hold, and if k is the order of x modulo n , we therefore know that k is a divisor of $n - 1$, but not a divisor of $(n - 1)/p$ for any prime factor p of $n - 1$; the only remaining possibility is $k = n - 1$. This completes the proof that conditions (i) and (ii) suffice to establish the primality of n .

Exercise 10 shows that we can in fact use different values of x for each of the primes p , and n will still be prime. We may restrict consideration to prime

values of x , since the order of uv modulo n divides the least common multiple of the orders of u and v by exercise 3.2.1.2–15. Conditions (i) and (ii) can be tested efficiently by using the rapid methods for evaluating powers of numbers discussed in Section 4.6.3. But it is necessary to know the prime factors of $n - 1$, so we have an interesting situation in which the factorization of n depends on that of $n - 1$.

An example. The study of a reasonably typical large factorization will help to fix the ideas we have discussed so far. Let us try to find the prime factors of $2^{214} + 1$, a 65-digit number. The factorization can be initiated with a bit of clairvoyance if we notice that

$$2^{214} + 1 = (2^{107} - 2^{54} + 1)(2^{107} + 2^{54} + 1); \quad (14)$$

this identity is a special case of some factorizations discovered by A. Aurifeuille in 1873 [see Dickson's *History*, 1, p. 383]. The problem now boils down to examining each of the 33-digit factors in (14).

A computer program readily discovers that $2^{107} - 2^{54} + 1 = 5 \cdot 857 \cdot n_0$, where

$$n_0 = 37866809061660057264219253397 \quad (15)$$

is a 29-digit number having no prime factors less than 1000. A multiple-precision calculation using the “binary method” of Section 4.6.3 shows that

$$3^{n_0-1} \bmod n_0 = 1,$$

so we suspect that n_0 is prime. It is certainly out of the question to prove that n_0 is prime by trying the 10 million million or so potential divisors, but the method discussed above gives a feasible test for primality: our next goal is to factor $n_0 - 1$. With little difficulty, our computer will tell us that

$$n_0 - 1 = 2 \cdot 2 \cdot 19 \cdot 107 \cdot 353 \cdot n_1, \quad n_1 = 13191270754108226049301.$$

Here $3^{n_1-1} \bmod n_1 \neq 1$, so n_1 is not prime; by continuing Algorithm A or Algorithm B we find

$$n_1 = 91813 \cdot n_2, \quad n_2 = 143675413657196977.$$

This time $3^{n_2-1} \bmod n_2 = 1$, so we will try to prove that n_2 is prime. This requires the factorization $n_2 - 1 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 547 \cdot n_3$, where $n_3 = 1824032775457$. Since $3^{n_3-1} \bmod n_3 \neq 1$, we know that n_3 is composite, and Algorithm A finds that $n_3 = 1103 \cdot n_4$, where $n_4 = 1653701519$. The number n_4 behaves like a prime (i.e., $3^{n_4-1} \bmod n_4 = 1$), so we calculate

$$n_4 - 1 = 2 \cdot 7 \cdot 19 \cdot 23 \cdot 137 \cdot 1973.$$

Good; this is our first complete factorization. We are now ready to backtrack to the previous subproblem, proving that n_4 is prime. Using the procedure suggested by exercise 10, we compute the following values:

x	p	$x^{(n_4-1)/p} \bmod n_4$	$x^{n_4-1} \bmod n_4$	
2	2	1	(1)	
2	7	766408626	(1)	
2	19	332952683	(1)	
2	23	1154237810	(1)	(16)
2	137	373782186	(1)	
2	1973	490790919	(1)	
3	2	1	(1)	
5	2	1	(1)	
7	2	1653701518	1	

(Here “(1)” means a result of 1 that needn’t be computed since it can be deduced from previous calculations.) Thus n_4 is prime, and $n_2 - 1$ has been completely factored. A similar calculation shows that n_2 is prime, and this complete factorization of $n_0 - 1$ finally shows [after still another calculation like (16)] that n_0 is prime.

The last three lines of (16) represent a search for an integer x that satisfies $x^{(n_4-1)/2} \not\equiv x^{n_4-1} \equiv 1 \pmod{n_4}$. If n_4 is prime, we have only a 50-50 chance of success, so the case $p = 2$ is typically the hardest one to verify. We could streamline this part of the calculation by using the law of quadratic reciprocity (cf. exercise 23), which tells us for example that $5^{(q-1)/2} \equiv 1 \pmod{q}$ whenever q is a prime congruent to $\pm 1 \pmod{5}$. Merely calculating $n_4 \bmod 5$ would have told us right away that $x = 5$ could not possibly help in showing that n_4 is prime. In fact, however, the result of exercise 26 implies that the case $p = 2$ doesn’t really need to be considered at all when testing n for primality, unless $n - 1$ is divisible by a high power of 2, so we could have dispensed with the last three lines of (16) entirely.

The next quantity to be factored is the other half of (14), namely

$$n_5 = 2^{107} + 2^{54} + 1.$$

Since $3^{n_5-1} \bmod n_5 \neq 1$, we know that n_5 is not prime, and Algorithm B shows that $n_5 = 843589 \cdot n_6$, where $n_6 = 192343993140277293096491917$. Unfortunately, $3^{n_6-1} \bmod n_6 \neq 1$, so we are left with a 27-digit nonprime. Continuing Algorithm B might well exhaust our patience (not our budget—nobody is paying for this, we’re using idle time on a weekend rather than “prime time”). But the sieve method of Algorithm D will be able to crack n_6 into its two factors,

$$n_6 = 8174912477117 \cdot 23528569104401.$$

This result could not have been discovered by Algorithm A in a reasonable length of time. (A few million iterations of Algorithm B would probably have sufficed.)

Now the computation is complete: $2^{214} + 1$ has the prime factorization

$$5 \cdot 857 \cdot 843589 \cdot 8174912477117 \cdot 23528569104401 \cdot n_0,$$

where n_0 is the 29-digit prime in (15). A certain amount of good fortune entered into these calculations, for if we had not started with the known factorization (14) it is quite probable that we would first have cast out the small factors, reducing n to $n_6 n_0$. This 55-digit number would have been much more difficult to factor—Algorithm D would be useless and Algorithm B would have to work overtime because of the high precision necessary.

Dozens of further numerical examples can be found in an article by John Brillhart and J. L. Selfridge, *Math. Comp.* **21** (1967), 87–96.

Improved primality tests. Since the above procedure for proving that n is prime requires the complete factorization of $n - 1$, it will bog down for large n . Another technique, which uses the factorization of $n + 1$ instead, is described in exercise 15; if $n - 1$ turns out to be too hard, $n + 1$ might be easier.

Significant improvements are available for dealing with large n . For example, it is not difficult to prove a stronger converse of Fermat's theorem that requires only a partial factorization of $n - 1$. Exercise 26 shows that we could have avoided most of the calculations in (16); the three conditions $2^{n_4-1} \bmod n_4 = \gcd(2^{(n_4-1)/23} - 1, n_4) = \gcd(2^{(n_4-1)/1973} - 1, n_4) = 1$ are sufficient by themselves to prove that n_4 is prime. Brillhart, Lehmer, and Selfridge have in fact developed a method that works when the numbers $n - 1$ and $n + 1$ have been only partially factored [*Math. Comp.* **29** (1975), 620–647, Corollary 11]: Suppose $n - 1 = f^- r^-$ and $n + 1 = f^+ r^+$, where we know the complete factorizations of f^- and f^+ , and we also know that all factors of r^- and r^+ are $\geq b$. If the product $(b^3 f^- f^+ \max(f^-, f^+))$ is greater than $2n$, a small amount of additional computation, described in their paper, will determine whether or not n is prime. Therefore numbers of up to 35 digits can usually be tested for primality in 2 or 3 seconds, simply by casting out all prime factors < 30030 from $n \pm 1$ [see J. L. Selfridge and M. C. Wunderlich, *Proc. Fourth Manitoba Conf. Numer. Math.* (1974), 109–120]. The partial factorization of other quantities like $n^2 \pm n + 1$ and $n^2 + 1$ can be used to improve this method still further [see H. C. Williams and J. S. Judd, *Math. Comp.* **30** (1976), 157–172, 867–886].

In practice, when n has no small prime factors and $3^{n-1} \bmod n = 1$, it has almost always turned out that n is prime. (One of the rare exceptions in the author's experience is $n = \frac{1}{7}(2^{28} - 9) = 2341 \cdot 16381$.) On the other hand, some nonprime values of n are definitely bad news for the primality test we have discussed, because it might happen that $x^{n-1} \bmod n = 1$ for all x relatively prime to n (see exercise 9). One such number is $n = 3 \cdot 11 \cdot 17 = 561$; here $\lambda(n) = \text{lcm}(2, 10, 16) = 80$ in the notation of Eq. 3.2.1.2–9, so $x^{80} \bmod 561 = 1 = x^{560} \bmod 561$ whenever x is relatively prime to 561. Our procedure would repeatedly fail to show that such an n is prime, until we had stumbled across one of its divisors. To improve the method, we need a quick way to determine the nonprimality of nonprime n , even in such pathological cases.

The following surprisingly simple procedure is guaranteed to do the job with high probability:

Algorithm P (Probabilistic primality test). Given an odd integer n , this algorithm attempts to decide whether or not n is prime. By repeating the algorithm several times, as explained in the remarks below, it is possible to be extremely confident about the primality of n , in a precise sense, yet the primality will not be rigorously proved. Let $n = 1 + 2^k q$, where q is odd.

- P1. [Generate x .] Let x be a random integer in the range $1 < x < n$.
- P2. [Exponentiate.] Set $j \leftarrow 0$ and $y \leftarrow x^q \pmod{n}$. (As in our previous primality test, $x^q \pmod{n}$ should be calculated in $O(\log q)$ steps, cf. Section 4.6.3.)
- P3. [Done?] (Now $y = x^{2^j q} \pmod{n}$.) If $j = 0$ and $y = 1$, or if $y = n - 1$, terminate the algorithm and say “ n is probably prime.” If $j > 0$ and $y = 1$, go to step P5.
- P4. [Increase j .] Increase j by 1. If $j < k$, set $y \leftarrow y^2 \pmod{n}$ and return to step P3.
- P5. [Not prime.] Terminate the algorithm and say that “ n is definitely not prime.” ■

The idea underlying Algorithm P is that if $n = 1 + 2^k q$ is prime and $x^q \pmod{n} \neq 1$, the sequence of values

$$x^q \pmod{n}, \quad x^{2q} \pmod{n}, \quad x^{4q} \pmod{n}, \quad \dots, \quad x^{2^k q} \pmod{n}$$

will end with 1, and the value just preceding the first appearance of 1 will be $n - 1$. (The only solutions to $y^2 \equiv 1 \pmod{p}$ are $y \equiv \pm 1$, when p is prime, since $(y - 1)(y + 1)$ must be a multiple of p .)

Exercise 22 proves the basic fact that Algorithm P will be wrong at most $\frac{1}{4}$ of the time, for all n . Actually it will rarely fail at all, for most n ; but the crucial point is that the probability of failure is bounded regardless of the value of n .

Suppose we invoke Algorithm P repeatedly, choosing x independently and at random whenever we get to step P1. If the algorithm ever reports that n is nonprime, we can say that n definitely isn’t prime. But if the algorithm reports 25 times in a row that n is “probably prime,” we can say that n is “almost surely prime.” For the probability is less than $(1/4)^{25}$ that such a 25-times-in-a-row procedure gives the wrong information about n . This is less than one chance in a quadrillion; even if we certified a billion different primes with such a procedure, the expected number of mistakes would be less than $\frac{1}{1000000}$. It’s much more likely that our computer has dropped a bit in its calculations, due to hardware malfunctions or cosmic radiations, than that Algorithm P has repeatedly guessed wrong!

Probabilistic algorithms like this lead us to question our traditional standards of reliability. Do we really need to have a rigorous proof of primality? For people unwilling to abandon traditional notions of proof, Gary L. Miller

has demonstrated that if \sqrt{n} is not an integer for any integer $r \geq 2$ (this condition being easily checked), and if a certain well-known conjecture in number theory called the Generalized Riemann Hypothesis can be proved, then either n is prime or there is an $x < 4(\ln n)^2$ such that Algorithm P will discover the nonprimality of n . [See *J. Comp. System Sci.* **13** (1976), 300–317. The constant 4 in this upper bound is due to Peter Weinberger, whose paper on the subject is not yet published.] Thus, we would have a rigorous way to test primality in $O(\log n)^5$ elementary operations, as opposed to a probabilistic method whose running time is $O(\log n)^3$. But one might well ask whether any purported proof of the Generalized Riemann Hypothesis will ever be as reliable as repeated application of Algorithm P on random x 's.

A probabilistic test for primality was first proposed in 1974 by R. Solovay and V. Strassen, who devised the interesting but more complicated test described in exercise 23(b). [See *SIAM J. Computing* **6** (1977), 84–85; **7** (1978), 118.] Algorithm P is a simplified version of a procedure due to M. O. Rabin, based in part on ideas of Gary L. Miller [cf. *Algorithms and Complexity*, ed. by J. F. Traub (New York: Academic Press, 1976), 35–36].

A completely different approach to primality testing was discovered in 1980 by Leonard M. Adleman. His highly interesting method is based on the theory of algebraic integers, so it is beyond the scope of this book; but it leads to a non-probabilistic procedure that will decide the primality of any number of up to, say, 250 digits, in a few hours at most. [See L. M. Adleman and R. S. Rumely, to appear.]

Factoring via continued fractions. The factorization procedures we have discussed so far will often balk at numbers of 30 digits or more, and another idea is needed if we are to go much further. Fortunately there is such an idea; in fact, there were two ideas, due respectively to A. M. Legendre and M. Kraitchik, that D. H. Lehmer and R. E. Powers used to devise a new technique many years ago [*Bull. Amer. Math. Soc.* **37** (1931), 770–776]. However, the method was not used at the time because it was comparatively unsuitable for desk calculators. This negative judgment prevailed until the late 1960s, when John Brillhart found that the Lehmer–Powers approach deserved to be resurrected, since it was quite well suited to computer programming. In fact, he and Michael A. Morrison later developed it into the champion of all known methods for factoring large numbers: Their program would handle typical 25-digit numbers in about 30 seconds, and 40-digit numbers in about 50 minutes, on an IBM 360/91 computer [see *Math. Comp.* **29** (1975), 183–205]. In 1970 the method had its first triumphant success, discovering that $2^{128} + 1 = 59649589127497217 \cdot 5704689200685129054721$.

The basic idea is to search for numbers x and y such that

$$x^2 \equiv y^2 \pmod{N}, \quad 0 < x, y < N, \quad x \neq y, \quad x + y \neq N. \quad (17)$$

Fermat's method imposes the stronger requirement $x^2 - y^2 = N$, but actually the congruence (17) is enough to split N into factors: It implies that N is a divisor of $x^2 - y^2 = (x-y)(x+y)$, yet N divides neither $x-y$ nor $x+y$; hence

$\gcd(N, x - y)$ and $\gcd(N, x + y)$ are proper factors of N that can be found by the efficient methods of Section 4.5.2.

One way to discover solutions of (17) is to look for values of x such that $x^2 \equiv a \pmod{N}$, for small values of $|a|$. As we will see, it is often a simple matter to piece together solutions of this congruence to obtain solutions of (17). Now if $x^2 = a + kNd^2$ for some k and d , with small $|a|$, the fraction x/d is a good approximation to \sqrt{kN} ; conversely, if x/d is an especially good approximation to \sqrt{kN} , the difference $|x^2 - kNd^2|$ will be small. This observation suggests looking at the continued fraction expansion of \sqrt{kN} , since we have seen (in Eq. 4.5.3-12 and exercise 4.5.3-42) that continued fractions yield good rational approximations.

Continued fractions for quadratic irrationalities have many pleasant properties, which are proved in exercise 4.5.3-12. The algorithm below makes use of these properties to derive solutions to the congruence

$$x^2 \equiv (-1)^{e_0} p_1^{e_1} p_2^{e_2} \dots p_m^{e_m} \pmod{N}. \quad (18)$$

Here we use a fixed set of small primes $p_1 = 2, p_2 = 3, \dots$, up to p_m ; only primes p such that either $p = 2$ or $(kN)^{(p-1)/2} \pmod{p} \leq 1$ should appear in this list, since other primes will never be factors of the numbers generated by the algorithm (see exercise 14). If $(x_1, e_{01}, e_{11}, \dots, e_{m1}), \dots, (x_r, e_{0r}, e_{1r}, \dots, e_{mr})$ are solutions of (18) such that the vector sum

$$(e_{01}, e_{11}, \dots, e_{m1}) + \dots + (e_{0r}, e_{1r}, \dots, e_{mr}) = (2e'_0, 2e'_1, \dots, 2e'_m) \quad (19)$$

is even in each component, then

$$x = (x_1 \dots x_r) \pmod{N}, \quad y = ((-1)^{e'_0} p_1^{e'_1} \dots p_m^{e'_m}) \pmod{N} \quad (20)$$

yields a solution to (17), except for the possibility that $x \equiv \pm y$. Condition (19) essentially says that the vectors are linearly dependent modulo 2, so we must have a solution to (19) if we have found at least $m + 2$ solutions to (18).

Algorithm E (*Factoring via continued fractions*). Given a positive integer N and a positive integer k such that kN is not a perfect square, this algorithm attempts to discover solutions to the congruence (18) for fixed m , by analyzing the convergents of the continued fraction for \sqrt{kN} . (Another algorithm, which uses the outputs to discover factors of N , is the subject of exercise 12.)

E1. [Initialize.] Set $D \leftarrow kN$, $R \leftarrow \lfloor \sqrt{D} \rfloor$, $R' \leftarrow 2R$, $U \leftarrow U' \leftarrow R'$, $V \leftarrow 1$, $V' \leftarrow D - R^2$, $P \leftarrow R$, $P' \leftarrow 1$, $A \leftarrow 0$, $S \leftarrow 0$. (This algorithm follows the general procedure of exercise 4.5.3-12, finding the continued fraction expansion of \sqrt{kN} . The variables U , U' , V , V' , P , P' , A , and S represent, respectively, what that exercise calls $R + U_n$, $R + U_{n-1}$, V_n , V_{n-1} , $p_n \pmod{N}$, $p_{n-1} \pmod{N}$, A_n , and $n \pmod{2}$. We will always have

$$0 < V \leq U \leq R',$$

so the highest precision is needed only for P and P' .)

- E2.** [Advance $U, V, S.$] Set $T \leftarrow V, V \leftarrow A(U' - U) + V', V' \leftarrow T, A \leftarrow \lfloor U/V \rfloor, U' \leftarrow U, U \leftarrow R' - (U \bmod V), S \leftarrow 1 - S.$
- E3.** [Factor $V.$] (Now we have $P^2 - kNQ^2 = (-1)^S V$, for some Q relatively prime to P , by exercise 4.5.3-12(c).) Set $(e_0, e_1, \dots, e_m) \leftarrow (S, 0, \dots, 0)$, $T \leftarrow V$. Now do the following, for $1 \leq j \leq m$: If $T \bmod p_j = 0$, set $T \leftarrow T/p_j$ and $e_j \leftarrow e_j + 1$, and repeat this process until $T \bmod p_j \neq 0$.
- E4.** [Solution?] If $T = 1$, output the values $(P, e_0, e_1, \dots, e_m)$, which comprise a solution to (18). (If enough solutions have been generated, we may terminate the algorithm now.)
- E5.** [Advance $P, P'.$] If $V \neq 1$, set $T \leftarrow P, P \leftarrow (AP + P') \bmod N, P' \leftarrow T$, and return to step E2. Otherwise the continued fraction process has started to repeat its cycle, except perhaps for S , so the algorithm terminates. The cycle will usually be so long that this doesn't happen.) ■

We can illustrate the application of Algorithm E to relatively small numbers by considering the case $N = 197209$, $k = 1$, $m = 3$, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$. The computation proceeds as follows:

	U	V	A	P	S	T	Output
After E1:	888	1	0	444	0	—	
After E4:	876	73	12	444	1	73	
After E4:	882	145	6	5329	0	29	
After E4:	857	37	23	32418	1	37	
After E4:	751	720	1	159316	0	1	$159316^2 \equiv +2^4 \cdot 3^2 \cdot 5^1$
After E4:	852	143	5	191734	1	143	
After E4:	681	215	3	131941	0	43	
After E4:	863	656	1	193139	1	41	
After E4:	883	33	26	127871	0	11	
After E4:	821	136	6	165232	1	17	
After E4:	877	405	2	133218	0	1	$133218^2 \equiv +2^0 \cdot 3^4 \cdot 5^1$
After E4:	875	24	36	37250	1	1	$37250^2 \equiv -2^3 \cdot 3^1 \cdot 5^0$
After E4:	490	477	1	93755	0	53	

Continuing the computation gives 25 outputs in the first 100 iterations; in other words, the algorithm is finding solutions quite rapidly. But some of the solutions are trivial. For example, if the above computation were continued 13 more times, we would obtain the output $197197^2 \equiv 2^4 \cdot 3^2 \cdot 5^0$, which is of no interest since $197197 \equiv -12$. The first two solutions above are already enough to complete the factorization: We have found that

$$(159316 \cdot 133218)^2 \equiv (2^2 \cdot 3^3 \cdot 5^1)^2 \pmod{197209};$$

thus (17) holds with $x = (159316 \cdot 133218) \bmod 197209 = 126308$, $y = 540$. By Euclid's algorithm, $\gcd(126308 - 540, 197209) = 199$; hence we obtain the pretty factorization

$$197209 = 199 \cdot 991.$$