12. Integers

Learning objectives:

- integers and their operations
- Euclidean algorithm
- Sieve of Eratosthenes
- large integers
- modular arithmetic
- Chinese remainder theorem
- random numbers and their generators

Operations on integers

Five basic operations account for the lion's share of integer arithmetic:

+ − · div mod

The product 'x · y', the quotient 'x div y', and the remainder 'x mod y' are related through the following *div-mod identity*:

(1) (x div y) · y + (x mod y) = x for y ≠ 0.

Many programming languages provide these five operations, but unfortunately, 'mod' tends to behave differently not only between different languages but also between different implementations of the same language. How come have we not learned in school what the remainder of a division is?

The div-mod identity, a cornerstone of number theory, defines 'mod' assuming that all the other operations are defined. It is mostly used in the context of nonnegative integers x ≥ 0, y > 0, where everything is clear, in particular the convention 0 ≤ x mod y < y. One half of the domain of integers consists of negative numbers, and there are good reasons for extending all five basic operations to the domain of all integers (with the possible exception of y = 0), such as:

- Any operation with an undefined result hinders the portability and testing of programs: if the "forbidden" operation does get executed by mistake, the computation may get into nonrepeatable states. Example: from a practical point of view it is better not to leave 'x div 0' undefined, as is customary in mathematics, but to define the result as '= overflow', a feature typically supported in hardware.
- Some algorithms that we usually consider in the context of nonnegative integers have natural extensions into the domain of all integers (see the following sections on 'gcd' and modular number representations).

Unfortunately, the attempt to extend 'mod' to the domain of integers runs into the problem mentioned above: How should we define 'div' and 'mod'? Let's follow the standard mathematical approach of listing desirable properties these operations might possess. In addition to the "sacred" div-mod identity (1) we consider:

(2) Symmetry of div: (−x) div y = x div (−y) = −(x div y).

The most plausible way to extend 'div' to negative numbers.

(3) A constraint on the possible values assumed by 'x mod y', which, for y > 0, reduces to the convention of nonnegative remainders:

0 ≤ x mod y < y.

This is important because a standard use of 'mod' is to partition the set of integers into y residue classes. We consider a weak and a strict requirement:

(3') Number of residue classes = |y|: for given y and varying x, 'x mod y' assumes exactly |y| distinct values.

(3") In addition, we ask for nonnegative remainders: 0 ≤ x mod y < |y|.

Pondering the consequences of these desiderata, we soon realize that 'div' cannot be extended to negative arguments by means of symmetry. Even the relatively innocuous case of positive denominator y > 0 makes it impossible to preserve both (2) and (3"), as the following failed attempt shows:

((−3) div 2) · 2 + ((−3) mod 2) ?=? −3 Preserving (1)

(−(3 div 2)) · 2 + 1 ?=? −3 and using (2) and (3")

(−1) · 2 + 1 ≠ −3 ... fails!

Even the weak condition (3'), which we consider essential, is incompatible with (2). For y = −2, it follows from (1) and (2) that there are three residue classes modulo (−2): x mod (−2) yields the values 1, 0, −1; for example,

1 mod (−2) = 1, 0 mod (−2) = 0, (−1) mod (−2) = −1.

This does not go with the fact that 'x mod 2' assumes only the two values 0, 1. Since a reasonable partition into residue classes is more important than the superficially appealing symmetry of 'div', we have to admit that (2) was just wishful thinking.

Without giving any reasons, [Knu 73a] (see the chapter "Reducing a task to given primitives; programming motion) defines 'mod' by means of the div-mod identity (1) as follows:

x mod y = x − y · ⌊x / y⌋, if y ≠ 0; x mod 0 = x;

Thus he implicitly defines x div y = ⌊x / y⌋, where ⌊z⌋, the "floor" of z, denotes the largest integer ≤ z; the "ceiling" ⌈z⌉ denotes the smallest integer ≥ z. Knuth extends the domain of 'mod' even further by defining "x mod 0 = x". With the exception of this special case y = 0, Knuth's definition satisfies (3'): Number of residue classes = |y|. The definition does not satisfy (3"), but a slightly more complicated condition. For given y ≠ 0, we have 0 ≤ x mod y < y, if y > 0; and 0 ≥ x mod y > y, if y < 0. Knuth's definition of 'div' and 'mod' has the added advantage that it holds for real numbers as well, where 'mod' is a useful operation for expressing the periodic behavior of functions [e.g. tan x = tan (x mod π)].

Exercise: another definition of 'div' and 'mod'

1. Show that the definition

$$x \operatorname{div} y := \begin{cases} \lfloor x/y \rfloor & \text{if } y > 0 \\ \lceil x/y \rceil & \text{if } y < 0 \end{cases}$$

in conjunction with the div-mod identity (1) meets the strict requirement (3").

Solution

x \ y	-5	-4	-3	-2	-1	1	2	3	4	5
5	-1	-1	-1	-2	-5	5	2	1	1	1
4	0	-1	-1	-2	-4	4	2	1	1	0
3	0	0	-1	-1	-3	3	1	1	0	0
2	0	0	0	-1	-2	2	1	0	0	0
1	0	0	0	0	-1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
-1	1	1	1	1	1	-1	-1	-1	-1	-1
-2	1	1	1	1	2	-2	-1	-1	-1	-1
-3	1	1	1	2	3	-3	-2	-1	-1	-1
-4	1	1	2	2	4	-4	-2	-2	-1	-1
-5	1	2	2	3	5	-5	-3	-2	-2	-1

x div y

x \ y	-5	-4	-3	-2	-1	1	2	3	4	5
5	0	1	2	1	0	0	1	2	1	0
4	4	0	1	0	0	0	0	1	0	4
3	3	3	0	1	0	0	1	0	3	3
2	2	2	2	0	0	0	0	2	2	2
1	1	1	1	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0
-1	4	3	2	1	0	0	1	2	3	4
-2	3	2	1	0	0	0	0	1	2	3
-3	2	1	0	1	0	0	1	0	1	2
-4	1	0	2	0	0	0	0	2	0	1
-5	0	3	1	1	0	0	1	1	3	0

x mod y

Exercise

Fill out comparable tables of values for Knuth's definition of 'div' and 'mod'.

Solution

x \ y	-5	-4	-3	-2	-1	1	2	3	4	5
5	-1	-2	-2	-3	-5	5	2	1	1	1
4	-1	-1	-2	-2	-4	4	2	1	1	0
3	-1	-1	-1	-2	-3	3	1	1	0	0
2	-1	-1	-1	-1	-2	2	1	0	0	0
1	-1	-1	-1	-1	-1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
-1	0	0	0	0	1	-1	-1	-1	-1	-1
-2	0	0	0	1	2	-2	-1	-1	-1	-1
-3	0	0	1	1	3	-3	-2	-1	-1	-1
-4	0	1	1	2	4	-4	-2	-2	-1	-1
-5	1	1	1	2	5	-5	-3	-2	-2	-1

x div y

x \ y	-5	-4	-3	-2	-1	1	2	3	4	5
5	0	-3	-1	-1	0	0	1	2	1	0
4	-1	0	-2	0	0	0	0	1	0	4
3	-2	-1	0	-1	0	0	1	0	3	3
2	-3	-2	-1	0	0	0	0	2	2	2
1	-4	-3	-2	-1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0
-1	-1	-1	-1	-1	0	0	1	2	3	4
-2	-2	-2	-2	0	0	0	0	1	2	3
-3	-3	-3	0	-1	0	0	1	0	1	2
-4	-4	0	-1	0	0	0	0	2	0	1
-5	0	-1	-2	-1	0	0	1	1	3	0

x mod y

The Euclidean algorithm

A famous algorithm for computing the greatest common divisor (gcd) of two natural numbers appears in Book 7 of Euclid's Elements (ca. 300 BC). It is based on the identity gcd(u, v) = gcd(u – v, v), which can be used for u > v to reduce the size of the arguments, until the smaller one becomes 0.

We use these properties of the greatest common divisor of two integers u and v > 0:

gcd(u, 0) = u By convention this also holds for u = 0.

gcd(u, v) = gcd(v, u) Permutation of arguments, important for the termination of the following procedure.

gcd(u, v) = gcd(v, u – q · v) For any integer q.

The formulas above translate directly into a recursive procedure:

```
function gcd(u, v: integer): integer;
begin
  if  v = 0  then  return(u)  else  return(gcd(v, u mod v))
end;
```

A test for the relative size of u and v is unnecessary. If initially u < v, the first recursive call permutes the two arguments, and thereafter the first argument is always larger than the second.

This simple and concise solution has a relatively high implementation cost. A stack, introduced to manage the recursive procedure calls, consumes space and time. In addition to the operations visible in the code (test for equality, assignment, and 'mod'), hidden stack maintenance operations are executed. There is an equally concise iterative version that requires a bit more thinking and writing, but is significantly more efficient:

```
function gcd(u, v: integer): integer;
var  r: integer;
begin
  while  v ≠ 0  do  { r := u mod v;  u := v;  v := r };
  return(u)
end;
```

The prime number sieve of Eratosthenes

The oldest and best-known algorithm of type *sieve* is named after Eratosthenes (ca. 200 BC). A set of elements is to be separated into two classes, the "good" ones and the "bad" ones. As is often the case in life, bad elements are easier to find than good ones. A sieve process successively eliminates elements that have been recognized as bad; each element eliminated helps in identifying further bad elements. Those elements that survive the epidemic must be good.

Sieve algorithms are often applicable when there is a striking asymmetry in the complexity or length of the proofs of the two assertions "p is a good element" and "p is a bad element". This theme occurs prominently in the complexity theory of problems that appear to admit only algorithms whose time requirement grows faster than polynomially in the size of the input (NP completeness). Let us illustrate this asymmetry in the case of prime numbers, for which Eratosthenes' sieve is designed. In this analogy, "prime" is "good" and "nonprime" is "bad".

A prime is a positive integer greater than 1 that is divisible only by 1 and itself. Thus primes are defined in terms of their *lack* of an easily verified property: a prime has no factors other than the two trivial ones. To prove that 1 675 307 419 is not prime, it suffices to exhibit a pair of factors:

$$1\,675\,307\,419 = 1\,234\,567 \cdot 1\,357.$$

This verification can be done by hand. The proof that $2^{17} - 1$ is prime, on the other hand, is much more elaborate. In general (without knowledge of any special property this particular number might have) one has to verify, for each and every number that qualifies as a candidate factor, that it is not a factor. This is obviously more time consuming than a mere multiplication.

Exhibiting factors through multiplication is an example of what is sometimes called a "one-way" or "trapdoor" function: the function is easy to evaluate (just one multiplication), but its inverse is hard. In this context, the inverse of multiplication is not division, but rather factorization. Much of modern cryptography relies on the difficulty of factorization.

The prime number sieve of Eratosthenes works as follows. We mark the smallest prime, 2, and erase all of its multiples within the desired range 1 .. n. The smallest remaining number must be prime; we mark it and erase its

multiples. We repeat this process for all numbers up to √n: If an integer c < n can be factored, c = a · b, then at least one of the factors is <√n.

```
{ sieve of Eratosthenes marks all the primes in 1 .. n }
const  n = … ;
var  sieve: packed array [2 .. n] of boolean;
       p, sqrtn, i: integer;
…
begin
  for  i := 2  to  n  do  sieve[i] := true;  { initialize the
sieve }
  sqrtn := trunc(sqrt(n));
  { it suffices to consider as divisors the numbers up to √ n }
  p := 2;
  while  p ≤ sqrtn  do  begin
    i := p · p;
    while  i ≤ n  do  { sieve[i] := false;  i := i + p };
    repeat  p := p + 1  until  sieve[p];
  end;
end;
```

Large integers

The range of numbers that can be represented directly in hardware is typically limited by the word length of the computer. For example, many small computers have a word length of 16 bits and thus limit integers to the range $-2^{15} \le a < +2^{15} = 32768$. When the built-in number system is insufficient, a variety of software techniques are used to extend its range. They differ greatly with respect to their properties and intended applications, but all of them come at an additional cost in memory and, above all, in the time required for performing arithmetic operations. Let us mention the most common techniques.

Double-length or double-precision integers. Two words are used to hold an integer that squares the available range as compared to integers stored in one word. For a 16-bit computer we get 32-bit integers, for a 32-bit computer we get 64-bit integers. Operations on double-precision integers are typically slower by a factor of 2 to 4.

Variable precision integers. The idea above is extended to allocate as many words as necessary to hold a given integer. This technique is used when the size of intermediate results that arise during the course of a computation is unpredictable. It calls for list processing techniques to manage memory. The time of an operation depends on the size of its arguments: linearly for addition, mostly quadratically for multiplication.

Packed BCD integers. This is a compromise between double precision and variable precision that comes from commercial data processing. The programmer defines the maximal size of every integer variable used, typically by giving the maximal number of decimal digits that may be needed to express it. The compiler allocates an array of bytes to this variable that contains the following information: maximal length, current length, sign, and the digits. The latter are stored in BCD (binary-coded decimal) representation: a decimal digit is coded in 4 bits, two of them are packed into a byte. Packed BCD integers are expensive in space because most of the time there is unused allocated space; and even more so in time, due to digit-by-digit arithmetic. They are unsuitable for lengthy scientific/technical computations, but OK for I/O-intensive data processing applications.

Modular number systems: the poor man's large integers

Modular arithmetic is a special-purpose technique with a narrow range of applications, but is extremely efficient where it applies—typically in combinatorial and number-theoretic problems. It handles addition, and particularly multiplication, with unequaled efficiency, but lacks equally efficient algorithms for division and comparison. Certain combinatorial problems that require high precision can be solved without divisions and with few comparisons; for these, modular numbers are unbeatable.

Chinese Remainder Theorem: Let $m_1, m_2, \dots, m_k$ be pairwise relatively prime positive integers, called *moduli*. Let $m = m_1 \cdot m2 \cdot \dots \cdot m_k$ be their product. Given k positive integers $r_1, r_2, \dots, r_k$, called *residues*, with $0 \le r_i < m_i$ for $1 \le i \le r_k$, there exists exactly one integer r, $0 \le r < m$, such that $r \bmod m_i = r_i$ for $1 \le i \le k$.

The Chinese remainder theorem is used to represent integers in the range $0 \le r < m$ uniquely as k-tuples of their residues modulo m_i. We denote this number representation by

$r \sim [r_1, r_2, \dots, r_k]$.

The practicality of modular number systems is based on the following fact: The arithmetic operations $(+, -, \cdot)$ on integers r in the range $0 \le r < m$ are represented by the same operations, applied componentwise to k-tuples $[r_1, r_2, \dots, r_k]$. A modular number system replaces a single +, –, or · in a large range by k operations of the same type in small ranges.

If $r \sim [r_1, r_2, \dots, r_k]$, $s \sim [s_1, s_2, \dots, s_k]$, $t \sim [t_1, t_2, \dots, t_k]$,
then:

$(r + s) \bmod m = t \Leftrightarrow (r_i + s_i) \bmod m_i = t_i$ for $1 \le i \le k$,

$(r - s) \bmod m = t \Leftrightarrow (r_i - s_i) \bmod m_i = t_i$ for $1 \le i \le k$,

$(r \cdot s) \bmod m = t \Leftrightarrow (r_i \cdot s_i) \bmod m_i = t_i$ for $1 \le i \le k$.

Example

$m_1 = 2$ and $m_2 = 5$, hence $m = m_1 \cdot m_2 = 2 \cdot 5 = 10$. In the following table the numbers r in the range 0 .. 9 are represented as pairs modulo 2 and modulo 5.

r	0	1	2	3	4	5	6	7	8	9
r mod 2	0	1	0	1	0	1	0	1	0	1
r mod 5	0	1	2	3	4	0	1	2	3	4

Let r = 2 and s = 3, hence $r \cdot s = 6$. In modular representation: $r \sim [0, 2]$, $s \sim [1, 3]$, hence $r \cdot s \sim [0, 1]$.

A useful modular number system is formed by the moduli

$m_1 = 99$, $m_2 = 100$, $m_3 = 101$, hence $m = m_1 \cdot m_2 \cdot m_3 = 999900$.

Nearly a million integers in the range $0 \le r < 999900$ can be represented. The conversion of a decimal number to its modular form is easily computed by hand by adding and subtracting pairs of digits as follows:

r mod 99: Add pairs of digits, and take the resulting sum mod 99.

r mod 100: Take the least significant pair of digits.

r mod 101: Alternatingly add and subtract pairs of digits, and take the result mod 101.

The largest integer produced by operations on components is $100^2 \sim 2^{13}$; it is smaller than $2^{15} = 32768 \sim 32k$ and thus causes no overflow on a computer with 16-bit arithmetic.

Example

r = 123456

r mod 99 = (56 + 34 + 12) mod 99 = 3

r mod 100 = 56

r mod 101 = (56 – 34 + 12) mod 101 = 34

r ~ [3, 56, 34]

s = 654321

s mod 99 = (21 + 43 + 65) mod 99 = 30

s mod 100 = 21

s mod 101 = (21 – 43 + 65) mod 101 = 43

s ~ [30, 21, 43]

r + s ~ [3, 56, 34] + [30, 21, 43] = [33, 77, 77]

Modular arithmetic has some shortcomings: division, comparison, overflow detection, and conversion to decimal notation trigger intricate computations.

Exercise: Fibonacci numbers and modular arithmetic

The sequence of Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

is defined by

$x_0 = 0, x_1 = 1, x_n = x_{n-1} + x_{n-2}$ for $n \geq 2$.

Write (a) a recursive function (b) an iterative function that computes the n-th element of this sequence. Using modular arithmetic, compute Fibonacci numbers up to 10^8 on a computer with 16-bit integer arithmetic, where the largest integer is $2^{15} - 1 = 32767$.

(c) Using moduli $m_1 = 999$, $m_2 = 1000$, $m_3 = 1001$, what is the range of the integers that can be represented uniquely by their residues $[r_1, r_2, r_3]$ with respect to these moduli?

(d) Describe in words and formulas how to compute the triple $[r_1, r_2, r_3]$ that uniquely represents a number r in this range.

(e) Modify the function in (b) to compute Fibonacci numbers in modular arithmetic with the moduli 999, 1000, and 1001. Use the declaration

```
type  triple = array [1 .. 3] of integer;
```

and write the procedure

```
procedure modfib(n: integer; var r: triple);
```

Solution

(a)

```
function fib(n: integer): integer;
begin
  if  n ≤ 1  then  return(n)  else  return(fib(n – 1) + fib(n – 2))
end;
```

(b)

```
function fib(n: integer): integer;
var  p, q, r, i: integer;
begin
  if  n ≤ 1  then  return(n)
```

```
    else  begin
      p := 0;  q := 1;
      for  i := 2  to  n  do  { r := p + q;  p := q;  q := r };
      return(r)
    end
  end;
```

(c) The range is 0 .. m – 1 with $m = m_1 \cdot m_2 \cdot m_3 = 999\,999\,000$.

(d) $r = d_1 \cdot 1\,000\,000 + d_2 \cdot 1000 + d_3$ with $0 \le d_1, d_2, d_3 \le 999$

$1\,000\,000 = 999\,999 + 1 = 1001 \cdot 999 + 1$

$1000 = 999 + 1 = 1001 - 1$

$r_1 = r \bmod 999 = (d_1 + d_2 + d_3) \bmod 999$

$r_2 = r \bmod 1000 = d_3$

$r_3 = r \bmod 1001 = (d_1 - d_2 + d_3) \bmod 1001$

(e)
```
procedure modfib(n: integer; var r: triple);
var  p, q: triple;
       i, j: integer;
begin
  if  n ≤ 1  then
    for  j := 1  to  3  do  r[j] := n
  else  begin
    for  j := 1  to  3  do  { p[j] := 0;  q[j] := 1 };
    for  i := 2  to  n  do  begin
      for  j := 1  to  3  do  r [j] := (p[j] + q[j]) mod (998 + j);
      p := q;  q := r
    end
  end
end;
```

Random numbers

The colloquial meaning of the term *at random* often implies "unpredictable". But *random numbers* are used in scientific/technical computing in situations where unpredictability is neither required nor desirable. What is needed in simulation, in sampling, and in the generation of test data is *not* unpredictability but certain statistical properties. A *random number generator* is a program that generates a sequence of numbers that passes a number of specified statistical tests. Additional requirements include: it runs fast and uses little memory; it is portable to computers that use a different arithmetic; the sequence of random numbers generated can be reproduced (so that a test run can be repeated under the same conditions).

In practice, random numbers are generated by simple formulas. The most widely used class, linear congruential generators, given by the formula

$$r_{i+1} = (a \cdot r_i + c) \bmod m$$

are characterized by three integer constants: the multiplier a, the increment c, and the modulus m. The sequence is initialized with a seed r_0.

All these constants must be chosen carefully. Consider, as a bad example, a formula designed to generate random days in the month of February:

$$r_0 = 0,\ r_{i+1} = (2 \cdot r_i + 1) \bmod 28.$$

It generates the sequence 0, 1, 3, 7, 15, 3, 7, 15, 3, ... Since $0 \le r_i < m$, each generator of the form above generates a sequence with a prefix of length < m which is followed by a period of length ≤ m. In the example, the

prefix 0, 1 of length 2 is followed by a period 3, 7, 15 of length 3. Usually we want a long period. Results from number theory assert that a period of length m is obtained if the following conditions are met:

- m is chosen as a prime number.
- (a – 1) is a multiple of m.
- m does not divide c.

Example

$r_0 = 0, \quad r_{i+1} = (8 \cdot r_i + 1) \bmod 7$

generates a sequence: 0, 1, 2, 3, 4, 5, 6, 0, … with a period of length 7.

Shall we accept this as a sequence of random integers, and if not, why not? Should we prefer the sequence 4, 1, 6, 2, 3, 0, 5, 4, … ?

For each application of random numbers, the programmer/analyst has to identify the important statistical properties required. Under normal circumstances these include:

No periodicity over the length of the sequence actually used. *Example:* to generate a sequence of 100 random weekdays ∈ {Su, Mo, … , Sat}, do not pick a generator with modulus 7, which can generate a period of length at most 7; pick one with a period much longer than 100.

A desired distribution, most often the uniform distribution. If the range 0 .. m – 1 is partitioned into k equally sized intervals $I_1, I_2, \ldots, I_k$, the numbers generated should be uniformly distributed among these intervals; this must be the case not only at the end of the period (this is trivially so for a generator with maximal period m), but for any initial part of the sequence.

Many well-known statistical tests are used to check the quality of random number generators. The *run test* (the lengths of monotonically increasing and monotonically decreasing subsequences must occur with the right frequencies); the *gap test* (given a test interval called the "gap", how many consecutively generated numbers fall outside?); the *permutation test* (partition the sequence into subsequences of t elements; there are t! possible relative orderings of elements within a subsequence; each of these orderings should occur about equally often).

Exercise: visualization of random numbers

Write a program that lets its user enter the constants a, c, m, and the seed r_0 for a linear congruential generator, then displays the numbers generated as dots on the screen: A pair of consecutive random numbers is interpreted as the (x, y)-coordinates of the dot. You will observe that most generators you enter have obvious flaws: our visual system is an excellent detector of regular patterns, and most regularities correspond to undesirable statistical properties.

The point made above is substantiated in [PM 88].

The following simple random number generator and some of its properties are easily memorized:

$r_0 = 1, \; r_{i+1} = 125 \cdot r_i \bmod 8192.$

1. $8192 = 2^{13}$, hence the remainder mod 8192 is represented by the 13 least significant bits.
2. $125 = 127 - 2 = (1111101)$ in binary representation.
3. Arithmetic can be done with 16-bit integers without overflow and without regard to the representation of negative numbers.

4. The numbers r_k generated are exactly those in the range $0 \le r_k < 8192$ with $r_k \bmod 4 = 1$ (i.e. the period has length $2_{11} = 2048$).
5. Its statistical properties are described in [Kru 69], [Knu 81] contains the most comprehensive treatment of the theory of random number generators.

As a conclusion of this brief introduction, remember an important rule of thumb:

Never choose a random number generator at random!

Exercises

1. Work out the details of implementing double-precision, variable-precision, and BCD integer arithmetic, and estimate the time required for each operation as compared to the time of the same operation in single precision. For variable precision and BCD, introduce the length L of the representation as a parameter.
2. The least common multiple (lcm) of two integers u and v is the smallest integer that is a multiple of u and v. Design an algorithm to compute lcm(u, v).
3. The prime decomposition of a natural number $n > 0$ is the (unique) multiset PD(n) = [p1, p2, ... , pk] of primes pi whose product is n. A multiset differs from a set in that elements may occur repeatedly (e.g. PD(12) = [2, 2, 3]). Design an algorithm to compute PD(n) for a given $n > 0$.
4. Work out the details of modular arithmetic with moduli 9, 10, 11.
5. Among the 95 linear congruential random number generators given by the formula $r_{i+1} = a \cdot r_i \bmod m$, with prime modulus m = 97 and $1 < a < 97$, find out how many get disqualified "at first sight" by a simple visual test. Consider that the period of these RNGs is at most 97.