of $2^{35} - 31$, the sequence repeats after $2^{35} - 32$ integers have been generated.

While testing the Lehmer generator we tried reversing each 35-bit integer after it came out of the generator. That is to say, bits 1 and 35 were interchanged, bits 2 and 34 were interchanged, etc. Hence the sequence of reversed integers was tested for randomness. The impressive thing about the Lehmer generator is that the sequence of reversed integers yields test results which are not significantly different from the unreversed integers, indicating that the least-significant bits of the integers are as random (at least from the viewpoint of satisfying our test criteria) as the most-significant bits. In both modulus 2^k methods the least-significant bits are periodically nonrandom.

The recipe for the Lehmer method is: (1) find the largest prime p less than register capacity; (2) find a positive primitive root A of p, which has sufficiently (to be determined by statistical tests) many digits; (3) start with any positive integer $X_0 < p$ and generate the sequence of pseudorandom integers by the recursion relation:

$$X_{i+1} = AX_i \pmod{p}.$$

The sequence will repeat after $p-1$ integers have been generated.

The Lehmer generator for the IBM 704/9/90/94 is:

```
LDQ = 3125              Multiplier = 5^5.
MPY X                   ACMQ = 5^5 X_i .
DVP = Ø377777777741     AC = 5^5 X_i[mod (2^35-31)] = X_{i+1} .
STO X
ARS 8
ORA = O200000000001     Insert characteristic and roundoff
FAD = 0.
TRA 1, 4
```

Time on a 7094 Model I: 21 or 22 cycles.

3. Test Results

The generator was rather extensively tested. (A tabulation of the test results is presented in [5]) and passed the usual statistical tests (and some new tests) for random number generators. However, the particular application must be the ultimate criterion of the suitability of a sequence of pseudorandom numbers. Hence the user would do very well to formulate his own tests depending on the application and to test the sequence of pseudorandom numbers used.

RECEIVED FEBRUARY, 1966

REFERENCES

1. HULL, T. E., AND A. R. DOBELL. Mixed congruential random number generators for binary machines. *J. ACM 11* (1964), 31–40.
2. HULL, T. E., AND A. R. DOBELL. Random number generators. *SIAM Rev. 4* (1962), 230–254.
3. PEACH, PAUL. Bias in pseudo random numbers. *J. Amer. Statist. Ass. 56*, (1961), 610–618.
4. LEHMER, D. H. Mathematical methods in large-scale computing units. *Ann. Comp. Lab. Harvard U. 26* (1951), 141–146.
5. HUTCHINSON, DAVID W. A new uniform pseudo-random number generator. File No. 651, April 27, 1965, Dept. Computer Sciences, U. of Illinois, Urbana, Ill.

Algorithms

J. G. HERRIOT, Editor

ALGORITHM 286
EXAMINATION SCHEDULING [ZH]

J. E. L. PECK AND M. R. WILLIAMS (Recd. 17 Mar. 1964, 25 Jan. 1965 and 1 Mar. 1966)
University of Alberta, Calgary, Alta., Canada

```
procedure partition (incidence) graph of order : (m) into : (n)
  parts using weights : (w) bound : (max) preassignment :
  (preassign) of number : (pren);
  Boolean array incidence; integer array w, preassign;
  integer m, n, max, pren;
comment This is an heuristic examination time-tabling pro-
  cedure for scheduling m courses in n time periods. It is essen-
  tially the problem of graph partitioning and map coloring.
    In the terminology of graph theory: Given a graph of m ver-
      texes with a positive integer weight w[i] at the ith vertex,
      partition this graph into no more than n disjoint sets such
      that each set contains no two vertexes joined by an edge,
      and such that the total weight of each set is less than the
      prescribed bound max.
  We represent the graph as an m×m symmetric Boolean matrix
  incidence whose i,jth element is true if and only if vertex i is
  joined to vertex j by an edge (if a student is taking both course i
  and course j), diagonal elements being assigned the value true.
  The weight assigned to the ith vertex (number of students in the
  ith course) is w[i]. We shall see below that preassignment is
  permitted. The number of courses to be preassigned is given in
  pren and the course preassign [i, 1] is to be placed at the time
  preassign [i, 2].
    This procedure does not minimize the second order incidence
  i.e. a vertex i being assigned to the set k, where the set k-1
  contains a vertex j joined to i (a student writing two consecutive
  examinations), but this may be done by rearranging the sets
  after the partitioning is completed. The procedure contains its
  own output statements, but its driver should provide the input;
begin integer array row [1:m], number [1:n];
  integer i, j, sum, course, time;
  Boolean preset, completed;
INITIALIZE: preset := false;
  for j := 1 step 1 until n do number [j] := 0;
  for i := 1 step 1 until m do
  begin sum := 0;
    for j := 1 step 1 until m do
    if incidence [i, j] then sum := sum + 1;
    row [i] := sum
  end INITIALIZE. Note that row [i] now contains the multi-
    plicity of, or number of edges at the vertex i (number
    of courses which conflict with the course i). Of course since the
    incidence matrix is symmetric, less than half (i > j) need be
    stored. However, this procedure, for the sake of simplicity,
    is written for the whole matrix. Also note that row [i] will
    eventually contain the negative of the set number to which
    the ith vertex is assigned (examination time for the ith course)
    and number [j] will contain the weight of the jth set (number of
    candidates at time j). From here on we drop the allusions to
    graph theory in the comments;
THE PREASSIGNMENT: for j := 1 step 1 until pren do
  begin comment preassignment of courses to times is now car-
```

```
    ried out. If pren = 0, then there are no preassignments;
  course := preassign [j,1]; time:= preassign [j,2];
  comment We now attempt to assign this course to the given
    time;
SCRUTINIZE: if row [course] < 0 then
  begin outstring (1, 'This course'); outinteger (1, course);
    outstring (1, 'is already scheduled at time');
    outinteger (1, -row[course]);  go to NEXT
  end;
  if number [time] + w[course] > max then
  begin outstring (1, 'Space is not available for course');
    outinteger (1, course); outstring (1, 'at time');
    outinteger (1, time);  go to NEXT
  end;
  for i := 1 step 1 until m do
    if row [i] = - time then
    begin if incidence [i, course] then
      begin outstring (1, 'course number');
        outinteger (1, course); outstring (1, 'conflicts with');
        outinteger (1,i);
        outstring (1, 'which is already scheduled at');
        outinteger (1, time),
        go to NEXT
      end if incidence
    end if row;
SATISFACTORY: row[course] := -time;
  number [time] := number [time] + w [course];
  preset := true;
NEXT:
 end THE PREASSIGNMENT;
MAIN PROGRAM: begin Boolean array available [1:m];
  integer next;
  procedure check (course); integer course;
  begin integer j; comment This procedure renders un-
    available those courses conflicting with the given course;
    for j := 1 step 1 until m do
      if incidence [course,j] then available [j] := false
  end of procedure check.
    For each of the n time periods we select a suitable set of non-
    conflicting courses whose students will fit the examination
    room;
START OF MAIN PROGRAM:
  for time := 1 step 1 until n do
    if preset ≡ number[time] > 0 then
    begin comment The preceding Boolean equivalence di-
        rects the attention of the program initially only to
        those times where prescheduling has occurred. We now
        determine the available courses (i.e. unscheduled and
        nonconflicting). If course i is already scheduled, then
        row[i] is negative;
      completed := true;
      for i := 1 step 1 until m do if row [i] > 0 then
      begin available [i] := true; completed := false end
        else available [i] := false;
      if completed then go to OUTPUT;
      if preset then
      begin comment Some courses were prescheduled at
          this time. It is necessary to render their conflicts un-
          available;
        for i := 1 step 1 until m do
          if row[i] = -time then check (i)
      end prescheduled courses.
        We now select the available course with the most con-
        flicts. This is essentially the heuristic step and there-
        fore the place where variations on the method may be
        made;
AGAIN:
      sum := 0;
      for i := 1 step 1 until m do
        if available [i] ∧ row [i] > sum then
        begin next := i;  sum := row [i] end most conflicts;
      if sum > 0 then
      begin comment There exists an available course, so
          we test it (viz next) for size. If it does not fit we look
          for another;
        available [next] := false;
        if number [time] + w[next] > max then go to AGAIN;
        comment If we are here the course will fit so we use it;
        row [next] := -time;
        number [time] := number [time] + w[next];
        check (next); go to AGAIN
      end sum > 0
    end of the time loop;
    if preset then
      begin preset := false; go to START OF MAIN
        PROGRAM end
        In case of prescheduling this takes us back to try the re-
        maining time periods.
          If we have reached here with completed true then all
        courses are scheduled, but the converse may not be true,
        therefore;
    if ¬ completed then
    begin completed := true;
      for i := 1 step 1 until m do
        if row [i] > 0 then completed := false
    end ¬ completed and
  end of the main program;
OUTPUT: if ¬ completed then
  begin comment The following for statement outputs the
      courses that were not scheduled;
    outstring (1, 'courses not scheduled');
    for i := 1 step 1 until m do
      if row [i] > 0 then outinteger (1,i)
  end not scheduled.
    The following outputs the time period j, the number of stu-
    dents number[j] and the courses i written at time j;
TIMETABLE: outstring(1, 'time  enrolment  courses');
  for j := 1 step 1 until n do
  begin outinteger (1,j); outinteger (1, number[j]);
    for i := 1 step 1 until m do
      if row[i] = -j then outinteger (1,i)
  end j.
    The following outputs the courses, the times at which they are
    written, and their enrolment;
  outstring (1, 'course  time  enrolment');
  for i := 1 step 1 until m do
    if row [i]< 0 then outinteger (1, i); outinteger (1, row [i]);
      outinteger (1, w[i])
    else
    begin outinteger(1,i); outstring(1, 'unscheduled');
      outinteger (1, w[i])
    end
end of the procedure
```

REMARK ON ALGORITHM 279
CHEBYSHEV QUADRATURE [D1]

F. R. A. Hopgood and C. Litherland

[*Comm. ACM 9* (Apr. 1966), 270]

The 33rd line of the second column on page 270 should read:

if $m \neq 4 \wedge m \neq mmax \wedge r \geq m - 4$ **then**

A printing error showed $\wedge$ as 7433.