Students were sectioned according to both the nonpreference and preference algorithms, and the number of times they received their preferred sections in courses with multiple sections was tallied. The results show that when the nonpreference algorithm is used a student receives about 40 percent of the sections for which he has a preference, and when the preference algorithm is used this percentage increases to around 72 percent.

Both algorithms meet the original two requirements of machine sectioning equally well. A subjective analysis of the resulting balance of sections when the preference algorithm was used shows it to be almost as good as the nonpreference algorithm. Rarely was the imbalance more than one student.

5. Conclusions

Because of the small number of courses where second preferences were permitted, the consequences of alternate courses on the algorithm cannot be fully determined. The only evidence that substantiates its effectiveness is that PE sections were balanced as well as sections in non-PE courses. Further testing will probably show that ordering according to the number of seats left is not sufficient but that the ordering should be done according to the ratio of the number of seats left to seats originally available. Because of the relative unimportance of second preferences in the data tested, ordering by ratio was not tried.

A larger percentage of requests will probably decrease performance achieved if most requests are centered on only a few sections in each course, since section sizes are limited. In the test data used this seemed to be the case for some courses. Popular time periods may also cause trouble. For instance, there were only 811 requests for MWF at 8 o'clock classes while there were 1,390 requests for MWF at 9 o'clock classes. These numbers are only indicative of the trend, since the effects of the time schedule on requests is not readily analyzed.

From the test case tried, it appears that it is possible to allow students to have section preference with machine sectioning. The results achieved show that a student receives around 32 percent more of his preferred sections using the preference algorithm—yet the principle objectives of machine registration are still satisfied.

Acknowledgments. The author would like to thank Martin Faulkner and James King for their assistance in modifying and using the Washington State University machine registration program and for helpful discussions during development of the preference algorithm.

RECEIVED FEBRUARY, 1967; REVISED MAY, 1967

REFERENCES

1. FAULKNER, M. Computer sectioning and class scheduling. *Datamation 11* (June 1965), 35–37.
2. STOCKMAN, J. W. JR. A guide to automated class scheduling. *Data Proc. Mag. 6* (Oct. 1964), 30–33.
3. MACON, N., AND WALKER, E. E. A Monte Carlo algorithm for assigning students to classes. *Comm. ACM 9*, 5 (May 1966), 339–340.

Algorithms

J. G. HERRIOT, Editor

ALGORITHM 310
PRIME NUMBER GENERATOR 1 [A1]

B. A. CHARTRES (Recd. 25 Oct. 1966 and 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

```
integer procedure sieve1(m, p); value m; integer m; integer array p;
comment sieve1(m, p) generates the prime numbers less than or equal to m, and places them in the array p, setting p[1] = 2, p[2] = 3, p[3] = 5, ···, p[k] = (largest prime found). The value of the procedure is k, the number of primes less than or equal to m.
    The method used is a modification of the Sieve of Eratosthenes. In its customary form this method requires a repeated sweeping over m numbers (or m/2 odd numbers), crossing out all multiples of the ith prime on the ith sweep. The variation of the method used here condenses all these sweeps into one. When the odd integer n is being tested ("if n=q[i]") to see whether it should be crossed out ("t := false"), q[i], for i = 3, 4, ···, j, contains the smallest odd multiple of p[i] which is no smaller than either n or p[i] ↑ 2. The sequence of values taken on by q[i] defines the set of numbers crossed out because they are multiples of p[i]. The initial value of q[i] is p[i] ↑ 2 because all smaller odd multiples of p[i] have at least one other odd prime factor smaller than p[i]. For the same reason, q[j+1] does not become active ("j := j+1") until n has become equal to p[j] ↑ 2. The dimension of the arrays q and dq is therefore the number of primes less than or equal to the square root of m. Thus we have replaced repeated sweeps over the array p by (many more) repeated sweeps over part of the much smaller array q. This does not reduce the amount of computation, but does lead to a much more efficient computer implementation, as only the arrays q and dq need be held in a random access store.;
begin
  integer array q, dq[2 : 2.7×sqrt(m)/ln(m)];
  integer i, j, k, n;
  Boolean t;
  p[1] := j := k := 2;  p[2] := 3;  q[2] := 9;  dq[2] := 6;
  for n := 5 step 2 until m do
  begin
    t := true;
    for i := 2 step 1 until j do
    begin
      if n = q[i] then
      begin
        q[i] := n + dq[i];  t := false;
        if i = j then
        begin
          j := j + 1;  q[j] := p[j] ↑ 2;
          dq[j] := 2 × p[j];  go to A
        end
      end
    end;
    if t then
    begin
      k := k + 1;  p[k] := n
    end;
A: end;
  sieve1 := k
end sieve1
```

ALGORITHM 311
PRIME NUMBER GENERATOR 2 [A1]

B. A. CHARTRES (Recd. 25 Oct. 1966 and 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

```
integer procedure sieve2(m, p); value m;
  integer m; integer array p;
comment sieve2 is a faster version of sieve1. Two changes were
  made to obtain higher speed.
    (1) The multiples q[i] are sorted, smallest first, so that each
  value of n does not need to be compared with every q[i]. The
  sorted order of the q[i] is indicated by an index array r. The
  ith sorted element of q is q[r[i]]. It was found empirically that
  greater speed is obtained when the q[r[i]] are not kept con-
  stantly sorted, but are re-sorted only at the time a new prime is
  discovered. The integer jj indicates which of the q[r[i]] are sorted:
  q[r[3]] through q[r[jj-1]] are out of order, whereas q[r[jj]] through
  q[r[j]] are in order. Sorting is performed in two stages. A sift
  sort first rearranges r[3] through r[jj-1] into rr[3] through
  rr[jj-1]. Then a single merge sort combines rr[3] through rr[jj-1]
  and r[jj] through r[j] into r[1] through r[j].
    (2) All multiples of 3 are automatically excluded from con-
  sideration by stepping n alternately by 2 and 4, and, in a similar
  way, by stepping q[i] alternately by 2 × p[i] and 4 × p[i].;
begin
  integer array q, dq, sq, r, rr[2: 2.7×sqrt(m)/ln(m)];
  integer i, j, jj, k, n, ir, jr, dn;
  Boolean t;
  p[1] := dn := 2; p[2] := j := jj := k := r[3] := 3;
  p[3] := 5; q[3] := 25; dq[3] := 10; sq[3] := 30;
  for n := 7 step dn until m do
  begin
    t := true; dn := 6 − dn;
    for i := 3 step 1 until jj do
    begin
      ir := r[i];
      if n = q[ir] then
      begin
        q[ir] := n + dq[ir];
        dq[ir] := sq[ir] − dq[ir];
        t := false;
        if i = jj then
        begin
          jj := jj + 1;
          if ir = j then
          begin
            j := j + 1; r[j] := j;
            q[j] := p[j] ↑ 2;
            sq[j] := 6 × p[j];
            dq[j] := sq[j] × (1+(p[j] ÷ 3)) − 2× q[j]
          end
        end
      end
    end;
    if t then
    begin
      k := k + 1; p[k] := n;
A:    if jj = 3 then go to F;
      jj := jj − 1;
      if q[r[jj]] < q[r[jj+1]] then go to A;
      comment sift sort;
      rr[3] := r[3];
      for ir := 4 step 1 until jj do
      begin
        i := ir − 1;
B:      if q[r[ir]] < q[rr[i]] then
        begin
          rr[i+1] := rr[i]; i := i − 1;
          if i ⩾ 3 then go to B
        end;
        rr[i+1] := r[ir]
      end;
      comment merge sort;
      i := ir := 3; jr := jj + 1;
C:    if q[rr[ir]] ⩽ q[r[jr]] then
      begin
        r[i] := rr[ir]; ir := ir + 1;
        if ir > jj then go to E
      end
      else
      begin
        r[i] := r[jr]; jr := jr + 1;
        if jr > j then go to D
      end;
      i := i + 1; go to C;
D:    i := i + 1; r[i] := rr[ir]; ir := ir + 1;
      if ir ⩽ jj then go to D;
E:    jj := 3
    end;
F: end;
  sieve2 := k
end sieve2
```

REMARKS ON:

ALGORITHM 35 [A1]
SIEVE [T. C. Wood, *Comm. ACM 4* (Mar. 1961), 151]
ALGORITHM 310 [A1]
PRIME NUMBER GENERATOR 1 [B. A. Chartres, *Comm. ACM 10* (Sept. 1967), 569]
ALGORITHM 311 [A1]
PRIME NUMBER GENERATOR 2 [B. A. Chartres, *Comm. ACM 10* (Sept. 1967), 570]

B. A. CHARTRES (Recd. 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

The three procedures *Sieve*(*m*,*p*), *sieve*1(*m*,*p*), and *sieve*2(*m*,*p*), which all perform the same operation of putting the primes less than or equal to *m* into the array *p*, were tested and compared for speed on the Burroughs B5500 at the University of Virginia. The modification of *Sieve* suggested by J. S. Hillmore [*Comm. ACM 5* (Aug. 1962), 438] was used. It was also found that *Sieve* could be speeded up by a factor of 1.95 by avoiding the repeated evaluation of *sqrt*(*n*). The modification required consisted of declaring an integer variable *s*, inserting the statement *s* := *sqrt*(*n*) immediately after *i* := 3, and replacing *p*[*i*]≤*sqrt*(*n*) by *p*[*i*]≤*s*.

The running times for the computation of the first 10,000 primes were:

Sieve (Algorithm 35)	845 sec
Sieve (modified)	434 sec
*sieve*1	220 sec
*sieve*2	91 sec

The time required to compute the first k primes was found to be, for each algorithm, remarkably accurately represented by a power law throughout the range $500 \leq k \leq 50{,}000$. The running time of *Sieve* varied as $k^{1.40}$, that of *sieve*1 as $k^{1.53}$, and that of *sieve*2 as $k^{1.35}$. Thus the speed advantage of *sieve*2 over the other algorithms increases with increasing k. However, it should be noted that *sieve*2 took approximately 33 minutes to find the first 100,000 primes, and, if the power law can be trusted for extrapolation past this point (there is no reason known why it should be), it would take about 12 hours to find the first million primes.