Algorithm 123

SINGLE TRANSFERABLE VOTE BY MEEK'S METHOD

I. D. HILL[1], B. A. WICHMANN[2] and D. R. WOODALL[3]

[1] *Clinical Research Centre, Harrow, Middlesex, HA1 3UJ*

[2] *5 Ellis Farm Close, Woking, GU22 9QN*

[3] *Department of Mathematics, University of Nottingham, Nottingham NG7 2RD*

Received 17 October 1985, in final form 13 May 1986

1. INTRODUCTION

The single transferable vote (STV) method of conducting an election exists in a number of different formulations in different countries. Most of the methods are designed to be practicable when counting is by hand, and this necessarily enforces simplicity even at the expense of not always getting the best possible answer.

Meek[1,2] considered the question of the best possible method, within the STV framework, when a computer is available to do the counting, and it is his method that we present here. The method was rediscovered, in a different formulation, by Woodall.[4] However, neither Meek nor Woodall dealt with certain detailed points, such as how to resolve a 'tie', so we have had to extend the system to be complete. The algorithm as given here has been adopted by the Royal Statistical Society for its Council elections.

The basis of any STV system consists of the following. (1) Voting by order of preference of candidates, the first choice being marked 1, the second 2, and so on, on the ballot papers. (Meek also considered an alternative formulation in which voters would be allowed to indicate equal preference for some candidates instead of a strict ordering; we have not implemented this alternative.) (2) A quota for election, calculated from the number of votes and the number of seats to be filled. (3) A first counting by first preferences only, and the election of any candidate who equals or exceeds the quota (except in the special case of a multi-way tie). (4) Redistribution of surplus votes (above the quota) for any candidate, in accordance with the voters' further preferences, and election of any who now reach the quota. (5) When no further redistribution of surpluses is possible, the exclusion of the candidate who then has the fewest votes, and redistribution of those papers. (6) Further counting, election, redistribution of surpluses and exclusion as necessary, until all seats are filled.

In the Meek formulation the rule for redistributing surpluses is that, at every stage, if a candidate has votes totalling k times the quota, then he (or she) keeps $1/k$ of each of those votes and passes $(k-1)/k$ on to the next candidate on the voter's list. This same fraction applies also to portions of votes received as parts of other surpluses. This requires the iterative solution of non-linear equations. It is proved in Section 4 below that a solution always exists and is unique.

It should be emphasised that the results will not always be the same as by manual counting methods. The algorithm deliberately uses the power of the computer to get better results than are easily achievable by hand.

2. THE SPECIFICATION

2.1. At each stage, each candidate is in one of three states, designated as 'elected', 'excluded' and 'hopeful'. At the start every candidate is in the hopeful state.

2.2. At each stage the votes are scanned, and the one vote allowed to each voter may be split into parts that are assigned to the various candidates according to the voter's choices. At the first stage the whole of the vote goes to the first choice – this follows automatically from the operation of rules 2.1 and 2.3.

2.3. Each candidate, x, has an associated weight, w_x, and keeps a proportion w_x of each vote or part of a vote received, while passing on to another candidate (as specified by the voter's choices) a proportion $1-w_x$. Every hopeful candidate has weight 1, and therefore keeps everything received and passes nothing on. Every excluded candidate has weight 0, and therefore keeps nothing and passes everything on. Elected candidates have weights between 0 and 1, to be calculated by rule 2.5.

2.4. Thus if someone has voted for candidate a as first choice, b as second, c as third, and no more:

a receives from that voter w_a of a vote
b receives from that voter $(1-w_a)w_b$ of a vote
c receives from that voter $(1-w_a)(1-w_b)w_c$ of a vote

A fraction $(1-w_a)(1-w_b)(1-w_c)$ remains and this goes to 'excess'. (Note that if a hopeful candidate appears in the list, all the fractions beyond that point automatically become 0).

2.5. The quota is defined as (total votes – total excess)/(number of seats + 1), and the weights for elected candidates are found such that the total vote remaining with each of them equals the quota. This is done by the convergent iterative scheme specified in rule 2.9.

2.6. The weights having been found, the resulting total votes for each hopeful candidate are examined, and any candidate whose total votes equal or exceed the quota changes state from hopeful to elected (except in the special case where all the hopeful candidates either have zero votes or exactly equal the quota. In this case all those with zero votes are excluded, one other is excluded by a pseudo-random choice and the others are elected).

2.7. If no candidate were elected under rule 2.6, then the hopeful candidate with the fewest votes changes state from hopeful to excluded. Any tie is resolved by a pseudo-random choice.

2.8. If the total number of elected candidates is equal to the number of seats, the election is complete. Otherwise the process is repeated from rule 2.2.

2.9. The convergent iterative scheme is as follows: set w_j equal to 0 for excluded candidates, 1 for hopeful candidates, and their last calculated values w_j^0 for elected candidates. (Immediately after election of any candidate the last calculated value is 1 initially.) Applying rule 2.3, using these weights, let v_j be the total value of votes received by candidate j and let e be the total excess. Using this value for e, calculate the new quota q using rule 2.5. Finally update the weights for elected candidates to values $w_j^1 = w_j^0 q/v_j$. Repeat the process of successively updating v_j, e, q and w_j until every fraction q/v_j, for elected candidates, lies within the limits 0.99999 and 1.00001 (inclusive).

3. THE PROGRAM (by I. D. HILL and B. A. WICHMANN)

We have allowed for up to 40 candidates, but the necessary change to allow a larger number is trivial.

3.1. The data

The data file should be held on disc, or other device that allows quick 'rewinding', because it has to be read many times during program execution.

Its form should be as follows:

```
4 2
-2
3 1 3 4 0
4 1 3 2 0
2 4 1 3 0
1 2 0
2 2 4 3 1 0
1 3 4 2 0
0
"Adam"
"Basil"
"Charlotte"
"Donald"
"Title"
```

The first line means that there are 4 candidates for 2 seats. The second line means that candidate number 2 withdrew before the count. As many candidates as necessary may be included in this line, each preceded by a minus sign. If no candidate withdrew, the line should be omitted entirely. The third line means that 3 voters put candidate 1 first, candidate 3 second, candidate 4 third, and no more. Each such list must end with a zero. The final zero ends the votes. The subsequent lines name the candidates, in the order of candidate numbers as used in the votes, and finally give a title for the election. If any of these names, or the title, is longer than 20 characters, only the first 20 will be used.

For elections on any substantial scale, further programs are desirable to get the data into this required form. Machine-readable ballot papers would obviously be a great help if a suitable system can be devised.

3.2. Ties

The only ties that can occur in this system are as follows. (1) If $n+1$ candidates all exactly equal the quota, where only n seats are available. One of these candidates must then be excluded (together with all other candidates, who necessarily have zero votes) and the other n elected. (2) If the candidate with fewest votes must be excluded and two or more have equal fewest. In both these cases a pseudo-random procedure is used, on the grounds that 'if they are equal, they are equal' and any procedure to choose one must be arbitrary. Alternatives are sometimes recommended, such as excluding the one who had fewer votes the first time they were different, or the last time they were different, or whatever, but such rules add much complication for no real advantage, so simplicity is preferable.

The pseudo-random generator is derived, with permission, from *Applied Statistics* algorithm AS 183.[3] This needs three seeds to initialise it, and these are formed from data items for the particular election. This leaves it sufficiently nearly random that nobody can manipulate it to favour a particular candidate, yet has the advantage that, for a given election, there is always a unique result. Running it on a different day, or using a different computer, will make no change – in the unlikely event that a random choice is needed, the same thing will always happen for any given data set. If a tie does occur and a random choice has to be made, a warning message is printed.

It should be emphasised that a tie that actually influences the result is a very rare event.

3.3. Partial abstentions

There is no compulsion on voters to give a complete listing of candidates. They may stop short if desired. If they do so and the use of their vote 'runs off the end' we allow it to do so, but adjust the quota to allow for the fact that there are now fewer remaining usable votes. This treats the partial abstention in such a way as to be fair to all remaining candidates.

This usage is different from that adopted in most manual counting systems where, under such circumstances, votes are divided into 'transferable' and 'non-transferable' and no quota adjustment is made. We are convinced that, within Meek's system, our approach is right, but it has to be made clear that we are in dispute over this with the council of the Electoral Reform Society. We have held up publication of the algorithm in the hope of resolving the difficulty, but now feel that we can wait no longer. Unfortunately, it is therefore necessary to warn potential users that they may be told by others that our method is undesirable in this particular.

3.4. Language

The algorithm is presented in standard Pascal. On some machines small, non-standard, changes may be required in the method of accessing the data file. We have used upper-case letters for Pascal word-symbols, lower-case or mixed-case for identifiers.

4. PROOF OF EXISTENCE AND UNIQUENESS (by D. R. WOODALL)

We prove in this section that the equations that need to be solved at each stage of Meek's method have a unique solution.

At each stage, each candidate is in one of three states, called '**elected**', '**excluded**' and '**hopeful**'. It is explained in Section 2 how a candidate arrives in one of these states; but for the purposes of the formal proof it is irrelevant: we may suppose that each candidate is assigned to one of these states at random, subject to the condition that the number m of 'elected' candidates is non-zero and does not exceed the number s of seats to be filled: $1 \leqslant m \leqslant s$. We also require the following **non-triviality condition**: there is at least one ballot paper that contains the name of a 'hopeful' candidate in its list of preferences. These conditions would certainly be fulfilled in a real election (in which no equation needs to be solved until some candidate is declared 'elected').

Let the 'elected' candidates be $C_1, \ldots, C_m$. Let the weight assigned to candidate X (as in Section 2.3) be w_X ($0 \leqslant w_X \leqslant 1$). Since each 'excluded' candidate always receives weight 0 and each 'hopeful' candidate receives weight 1, the assigned weights are specified uniquely by the m-vector $\boldsymbol{w} = (w_1, \ldots, w_m)$, in which w_j is the weight assigned to C_j for each j ($j = 1, \ldots, m$).

In the situation described by the m-vector $\boldsymbol{w}$, let $V_X(\boldsymbol{w})$ denote the vote for candidate X (that is, the sum of the part-votes that X receives from all the electors); for convenience, write $V_{C_j}(\boldsymbol{w})$ as $V_j(\boldsymbol{w})$. Let $E(\boldsymbol{w})$ denote the total excess vote, and define the **quota** $Q(\boldsymbol{w})$ to be $(V-E(\boldsymbol{w}))/(s+1)$, where V is the total number of votes (ballot papers). The effect of the non-triviality condition mentioned above is to ensure that $Q(\boldsymbol{w}) \geqslant 1/(s+1) > 0$ for all $\boldsymbol{w}$, since if a ballot paper contains the name of a 'hopeful' candidate among its preferences then no part of that vote can be lost to the excess vote, and so $V-E(\boldsymbol{w}) \geqslant 1$.

We shall make extensive use of the following facts, which are obvious from the above definitions and from Section 2.4, and in which we use the terms 'increases' and 'decreases' in the weak sense (that is, both terms correctly describe a number that does not change): if one component w_j of $\boldsymbol{w}$ is decreased whilst all the other components are unchanged, then:

(1) $V_j(\boldsymbol{w})$ *decreases*, in exact proportion to the decrease in w_j;
(2) each $V_k(\boldsymbol{w})$ $(k \neq j)$ *increases*;
(3) the sum of the votes for all the 'elected' candidates *decreases* by an amount $v \geqslant 0$ (since the contribution from each ballot paper decreases);
(4) the excess vote *increases*, by at most v;
(5) the quota *decreases*, by at most $v/(s+1)$.

Let an m-vector $\boldsymbol{w}$ be called **feasible** if $0 \leqslant w_j \leqslant 1$ and $V_j(\boldsymbol{w}) \geqslant Q(\boldsymbol{w})$ for each j, and be called a **solution vector** if $0 \leqslant w_j \leqslant 1$ and $V_j(\boldsymbol{w}) = Q(\boldsymbol{w})$ for each j ($j = 1, \ldots, m$). The purpose of this section is to prove that *if there is a feasible vector, then there is a unique solution vector*. We note in passing that,

```
PROGRAM stvpas(datafile, output);

{This program counts the votes in a Single Transferable Vote election,
    using Meek's method, and reports the results}

{If there are more than 40 candidates, an increase in the size of
    MaxCandidates is the only change needed}

CONST MaxCandidates = 40;
      NameLength = 20;

TYPE  Candidates = 1 .. MaxCandidates;
      CandRange = 0 .. MaxCandidates;
      name = PACKED ARRAY [1 .. NameLength] OF char;

VAR   NumCandidates, NumSeats: Candidates;
      candidate, NumElected, NumExcluded,
            multiplier, ignored: CandRange;
      Droop, excess, quota, total: real;
      faulty, SomeoneElected, RandomUsed: Boolean;
      FracDigits: 1 .. 4;
      table, seed1, seed2, seed3: integer;
      datafile: text;
      title: name;
      votes, weight: ARRAY [Candidates] OF real;
      status: ARRAY [Candidates] OF (Hopeful, Elected, NewlyElected,
                    Almost, Excluded, ToBeExcluded, NotUsed, Used);
      names: ARRAY [Candidates] OF name;

FUNCTION InInteger: integer;

{Reads the next integer from datafile and returns its value}

    VAR i: integer;
    BEGIN
    read(datafile, i);
    InInteger := i
    END; {InInteger}

PROCEDURE PrintOut;

{Updates the table number and prints out the current results}

    VAR arg: real;
    cand: Candidates;
    BEGIN
    table := table + 1;
    writeln;
    writeln(' ': 20, title);
    writeln;
    write('Table: ', table: 1);
    writeln('   Quota: ', quota: 1: FracDigits);
    writeln;

    {The numbers of blanks following Candidate, Retain and
        Transfer are 12, 3 and 3 respectively}

    writeln('Candidate            Retain   Transfer   Votes');
    writeln;

    FOR cand := 1 TO NumCandidates DO
        BEGIN
        write(names[cand]);
        IF status[cand] = ToBeExcluded THEN
            arg := 100.0 ELSE arg := 100.0 * weight[cand];
        write(arg: 6: 1, '%');
        write(100.0 - arg: 8: 1, '%');

        {If it is valid to do so,  print quota instead of votes[cand]
            because the latter might have a small rounding error that
            would confuse unsophisticated users}

        IF status[cand] = Elected THEN arg := votes[cand] / quota
            ELSE arg := 0.0;
        IF (arg >= 0.99999) AND (arg <= 1.00001) THEN arg := quota
            ELSE arg := votes[cand];
        write(arg: 10: FracDigits, '  ');
        IF status[cand] = Excluded THEN write('Excluded')
        ELSE IF status[cand] = Elected THEN write('Elected')
        ELSE IF status[cand] = NewlyElected THEN write('Newly Elected')
        ELSE IF status[cand] = ToBeExcluded THEN
            BEGIN
            write('To be Excluded');
            status[cand] := Excluded
            END;
        writeln;
        IF (NumCandidates > 9) AND (cand MOD 5 = 0) AND
            (cand <> NumCandidates) THEN writeln
        END;

    writeln;
    writeln('Excess', excess: 40: FracDigits);
    writeln;
    writeln('Total ', total: 40: FracDigits);
    writeln;
    writeln
    END; {PrintOut}

PROCEDURE elect(cand: Candidates);
    BEGIN
    status[cand] := NewlyElected;
    NumElected := NumElected + 1
    END; {elect}

PROCEDURE exclude(cand: Candidates);
    BEGIN
    status[cand] := ToBeExcluded;
    weight[cand] := 0.0;
    NumExcluded := NumExcluded + 1;
    IF RandomUsed THEN
        BEGIN
        writeln;
        writeln;
        writeln('Random choice used to exclude ', names[cand])
        END
    END; {exclude}

FUNCTION LowestCandidate: CandRange;

{Returns the candidate number of the candidate who currently has the
    lowest number of votes.  If two or more are equal lowest, then a
    pseudo-random choice is made between them}

    VAR cand: Candidates;
        LowCand: CandRange;

FUNCTION random: real;

{Returns a pseudo-random number, rectangularly distributed
    between 0 and 1. Based on Wichmann and Hill, Algorithm
    AS 183, Appl. Statist. (1982) 31, 188 - 190}

    VAR rndm: real;
    BEGIN

    {If seeds have not been set, then set them}

    IF seed1 = 0 THEN
        BEGIN
        seed1 := NumCandidates;
        seed2 := NumSeats + 10000;
        rndm := total + 20000.0;
        WHILE rndm > 30322.5 DO rndm := rndm - 30322.0;
        seed3 := round(rndm)
        END;

    seed1 := 171 * (seed1 MOD 177) - 2 * (seed1 DIV 177);
    seed2 := 172 * (seed2 MOD 176) - 35 * (seed2 DIV 176);
    seed3 := 170 * (seed3 MOD 178) - 63 * (seed3 DIV 178);
    IF seed1 < 0 THEN seed1 := seed1 + 30269;
    IF seed2 < 0 THEN seed2 := seed2 + 30307;
    IF seed3 < 0 THEN seed3 := seed3 + 30323;
    rndm := seed1 / 30269.0 + seed2 / 30307.0 + seed3 / 30323.0;
    random := rndm - trunc(rndm)
    END; {random}

FUNCTION lower(cand, lowest: CandRange): Boolean;

{Finds whether cand has fewer votes than lowest, and also
    reports whether a random choice had to be made}

    VAR lowly: Boolean;
    BEGIN
    IF lowest = 0 THEN
        BEGIN
        RandomUsed := false;
        lower := true
        END
    ELSE IF votes[cand] = votes[lowest] THEN
        BEGIN
        RandomUsed := true;

        {Multiplier is used to make all equally-lowest candidates
            equally likely to be chosen, even though they are
            considered serially and not simultaneously}

        lower := (multiplier * random < 1.0)
        END
    ELSE
        BEGIN
        lowly := (votes[cand] < votes[lowest]);
        lower := lowly;
        IF lowly THEN RandomUsed := false
        END;
    IF RandomUsed THEN multiplier := multiplier + 1
        ELSE multiplier := 2
    END; {lower}

BEGIN
LowCand := 0;
FOR cand := 1 TO NumCandidates DO
    IF (status[cand] = Hopeful) OR (status[cand] = Almost) THEN
        IF lower(cand, LowCand) THEN LowCand := cand;
LowestCandidate := LowCand
END; {LowestCandidate}

PROCEDURE compute;

{This is the heart of the program, which counts the votes, taking
    the current weights into account, and adjusts the weights and
    the quota iteratively to attain the required solution}

    {MaxIterations is the maximum number of iterations allowed in
        calculating the weights. It is unlikely that so many will
        ever be used, but its value may be increased if desired}

    CONST MaxIterations = 500;
    VAR temp, value: real;
        count, iteration: integer;
        cand: CandRange;
        converged, ended: Boolean;

    PROCEDURE Rewind;

    {Returns to the beginning of datafile, and ignores the first two
        numbers on it.    These are the number of candidates and the
        number of seats, whose values are not needed again.  Numbers
        indicating withdrawn candidates are also ignored}

        VAR ig, ignore: integer;
        BEGIN
        reset(datafile);
        FOR ig := -1 TO ignored DO ignore := InInteger
        END; {Rewind}

    BEGIN
    iteration := 1;

        REPEAT
        Rewind;
        excess := 0.0;
        FOR cand := 1 TO NumCandidates DO votes[cand] := 0.0;
        count := InInteger;

        WHILE count > 0 DO
            BEGIN
            value := count;
            cand := InInteger;
            ended := false;

            WHILE cand>0 DO
                BEGIN
                IF NOT ended AND (weight[cand] > 0.0) THEN
                    BEGIN
                    ended := (status[cand] = Hopeful);
                    IF ended THEN
                        BEGIN
                        votes[cand] := votes[cand] + value;
                        value := 0.0
                        END
                    ELSE
                        BEGIN
                        votes[cand] := votes[cand] + value * weight[cand];
                        value := value * (1.0 - weight[cand])
                        END
                    END;
                cand := InInteger
                END;

            excess := excess + value;
            count := InInteger
            END;

        quota := (total - excess) * Droop;
```

```
        {The next statement is unlikely ever to be used, but is a
            safeguard against certain pathological test data}

        IF quota < 0.0001 THEN quota := 0.0001;
        converged := true;

        FOR cand := 1 TO NumCandidates DO
            IF status[cand] = Elected THEN
                BEGIN
                temp := quota / votes[cand];
                IF (temp > 1.00001) OR (temp < 0.99999) THEN
                    converged := false;
                temp := weight[cand] * temp;
                weight[cand] := temp;

                {The next statement is unlikely ever to be used, but is
                    a safeguard against certain pathological test data}

                IF temp > 1.0 THEN weight[cand] := 1.0
                END;

        iteration := iteration + 1
        UNTIL (iteration = MaxIterations) OR converged;

    IF NOT converged THEN
        BEGIN

        {The "Failure to converge" message is unlikely ever to appear.
            If it does, increasing MaxIterations will probably cure it}

        writeln;
        writeln;
        writeln('Failure to converge');
        writeln
        END;
    count := 0;

    FOR cand := 1 TO NumCandidates DO
        IF (status[cand] = Hopeful) AND (votes[cand] >= quota) THEN
            BEGIN
            status[cand] := Almost;
            count := count + 1
            END;

    {Allow for the special case where there is a multi-way tie and
            too many candidates reach the quota simultaneously}

    WHILE NumElected + count > NumSeats DO
        BEGIN
        PrintOut;
        RandomUsed := false;
        FOR cand := 1 TO NumCandidates DO
            IF status[cand] = Hopeful THEN exclude(cand);
        exclude(LowestCandidate);
        count := count - 1
        END;

    SomeoneElected := false;
    FOR cand := 1 TO NumCandidates DO
        IF status[cand] = Almost THEN
            BEGIN
            elect(cand);
            SomeoneElected := true
            END;

    IF SomeoneElected THEN PrintOut;
    FOR cand := 1 TO NumCandidates DO
        IF status[cand] = NewlyElected THEN
            BEGIN
            IF NumElected < NumSeats THEN
                weight[cand] := quota / votes[cand];
            status[cand] := Elected
            END
    END; {compute}

PROCEDURE complete;

{Used to elect all remaining candidates if the number
    remaining equals the number of seats remaining}

    VAR cand: Candidates;
    BEGIN
    FOR cand := 1 TO NumCandidates DO
        IF status[cand] = Hopeful THEN elect(cand)
    END; {complete}

PROCEDURE Preliminaries;

{Checks datafile for errors and sets initial values of variables}

    VAR cand, count, LineNo: integer;

    PROCEDURE error(cand: integer; TooBig: Boolean);
        BEGIN
        writeln;
        write('On line ', LineNo: 1, ', Candidate ', cand: 1);
        IF TooBig THEN write (' exceeds maximum')
            ELSE write (' is repeated');
        writeln;
        faulty := true
        END; {error}

    PROCEDURE ReadName(VAR n: name);

        {Reads the name of a candidate, or reads a title, and stores
            it for later use.   If the name has more than NameLength
            characters the excess ones will be disregarded.    If it
            has fewer than NameLength characters blanks will be used
            to extend it}

        VAR i: integer;
            ch: char;
        BEGIN

            REPEAT
            read(datafile, ch)
            UNTIL ch = '"';

        i := 0;
        read(datafile, ch);
        WHILE ch <> '"' DO
            BEGIN
            IF i < NameLength THEN
                BEGIN
                i := i + 1;
                n[i] := ch
                END;
            read(datafile, ch)
            END;

        WHILE i < NameLength DO
            BEGIN
            i := i + 1;
            n[i] := ' '
            END
        END; {ReadName}

    BEGIN
    Droop := 1.0/(NumSeats + 1);
    LineNo := 1;
    seed1 := 0;
    total := 0.0;
    table := 0;
    NumElected := 0;
    NumExcluded := 0;
    ignored := 0;
    FOR cand := 1 TO NumCandidates DO weight[cand] := 1.0;
    count := InInteger;

    {Deal with withdrawals, if any}

    WHILE count < 0 DO
        BEGIN
        weight[-count] := 0.0;
        count := InInteger
        END;
    WHILE count > 0 DO
        BEGIN
        LineNo := LineNo + 1;
        total := total + count;
        FOR cand := 1 TO NumCandidates DO status[cand] := NotUsed;
        cand := InInteger;

        WHILE cand > 0 DO
            BEGIN
            IF cand > NumCandidates THEN error(cand, true)
                ELSE IF status[cand] = Used THEN error(cand, false)
                    ELSE status[cand] := Used;
            cand := InInteger
            END;

        count := InInteger
        END;

    FOR cand := 1 TO NumCandidates DO
        BEGIN
        ReadName(names[cand]);
        status[cand] := Hopeful;
        IF weight[cand] < 0.5 THEN
            BEGIN
            status[cand] := Excluded;
            NumExcluded := NumExcluded + 1;
            ignored := ignored + 1
            END
        END;
    ReadName(title);
    IF NOT faulty THEN
        BEGIN

        {FracDigits controls the number of digits beyond the decimal
            point that will be printed in the output tables}

        FracDigits := 4;
        IF total > 999.5 THEN FracDigits := FracDigits - 1;
        IF total > 99.5 THEN FracDigits := FracDigits - 1;
        IF total > 9.5 THEN FracDigits := FracDigits - 1
        END
    END; {Preliminaries}

{Start of main program}

BEGIN
reset(datafile);
NumCandidates := InInteger;
NumSeats := InInteger;
writeln;
writeln;
writeln('Number of Candidates = ', NumCandidates: 1);
writeln('Number of Seats = ', NumSeats: 1);
IF NumCandidates <= NumSeats THEN writeln('All candidates elected') ELSE
    BEGIN
    faulty := false;
    Preliminaries;
    IF NumCandidates <= NumSeats + NumExcluded THEN
        writeln('All non-withdrawn candidates elected') ELSE
        BEGIN

        {The Preliminaries procedure will have reset faulty to true if
            the data contain errors}

        IF NOT faulty THEN
            BEGIN
                REPEAT

                    {Count votes and elect candidates, transferring
                        surpluses until no more can be done  or all
                        seats are filled}

                    REPEAT
                    compute
                    UNTIL NOT SomeoneElected OR (NumElected >= NumSeats);

                {Unless the election is finished, someone must
                    now be excluded}

                IF NumElected < NumSeats THEN
                    BEGIN
                    PrintOut;
                    exclude(LowestCandidate);
                    IF NumCandidates - NumExcluded = NumSeats
                        THEN complete ELSE PrintOut
                    END
                UNTIL NumElected = NumSeats;

            {Now that all seats are filled, exclude any candidates not
                already elected, and print out the final table}

            RandomUsed := false;
            FOR candidate := 1 TO NumCandidates DO
                IF status[candidate] = Hopeful THEN exclude(candidate);
            PrintOut
            END
        END
    END.
```

in a real election, the existence of a feasible vector is assured, since the solution vector at each stage of the counting yields a feasible vector for the next stage.

We shall use the following algorithm which, starting with a feasible vector, will construct a solution vector. (This is the algorithm described in Section 2.9.)

Algorithm: Let $\boldsymbol{w}^0 = (w_1^0, \ldots, w_m^0)$ be a feasible vector. Given $\boldsymbol{w}^i$, define $\boldsymbol{w}^{i+1}$ by the rule

$$w_j^{i+1} := w_j^i Q(\boldsymbol{w}^i)/V_j(\boldsymbol{w}^i) \tag{6}$$

for each $j (j = 1, \ldots, m)$.

Theorem 1. *This Algorithm constructs a sequence of feasible vectors that converges to a solution vector.*

Proof. Suppose that $\boldsymbol{w}^i$ is a feasible vector, so that $V_j(\boldsymbol{w}^i) \geqslant Q(\boldsymbol{w}^i) > 0$ and so $w_j^i > 0$ for each j. Then to convert $\boldsymbol{w}^i$ into $\boldsymbol{w}^{i+1}$ we must (weakly) *decrease* each of its components. Fix j, and let $\boldsymbol{w}'$ be the vector obtained from $\boldsymbol{w}^i$ by replacing the one component w_j^i by w_j^{i+1}. By (2), (1), (6) and (5),

$$V_j(\boldsymbol{w}^{i+1}) \geqslant V_j(\boldsymbol{w}') = V_j(\boldsymbol{w}^i) w_j^{i+1}/w_j^i = Q(\boldsymbol{w}^i) \geqslant Q(\boldsymbol{w}^{i+1}). \tag{7}$$

This holds for each j, and so $\boldsymbol{w}^{i+1}$ is a feasible vector. Since $\boldsymbol{w}^0$ is feasible by hypothesis, it follows by induction that $\boldsymbol{w}^i$ is feasible for all i.

It follows from this that, for each fixed j, the sequence

$$w_j^0, w_j^1, w_j^2, \ldots$$

is a monotonic decreasing sequence that is bounded below (by 0), and so converges. Thus there is a limit vector $\boldsymbol{w}^\infty = (w_1^\infty, \ldots, w_m^\infty)$. We must prove that $\boldsymbol{w}^\infty$ is a solution vector. By the feasibility of $\boldsymbol{w}^i$ and (7),

$$\begin{aligned} 0 \leqslant V_j(\boldsymbol{w}^i) - Q(\boldsymbol{w}^i) &= V_j(\boldsymbol{w}^i) - V_j(\boldsymbol{w}'), \\ &\leqslant V \cdot (w_j^i - w_j^{i+1}) \end{aligned}$$

since decreasing w_j by δ cannot decrease $V_j(\boldsymbol{w})$ by more than $V\delta$ (V being the total number of ballot papers). But, as $i \to \infty$, $w_j^i - w_j^{i+1} \to 0$, and since $V_j(\boldsymbol{w})$ and $Q(\boldsymbol{w})$ are continuous functions of $\boldsymbol{w}$ it follows that $V_j(\boldsymbol{w}^\infty) = Q(\boldsymbol{w}^\infty)$. This holds for each j, and so $\boldsymbol{w}^\infty$ is a solution vector, as required. □

Theorem 2. *The solution vector, whose existence was proved in Theorem 1, is unique.*

Proof. Let $\boldsymbol{w} = (w_1, \ldots, w_m)$ and $\boldsymbol{w}^* = (w_1^*, \ldots, w_m^*)$ be two solution vectors, and define $\boldsymbol{w}^0 = (w_1^0, \ldots, w_m^0)$ by

$$w_j^0 := \min(w_j, w_j^*)$$

for each j. For a fixed j, suppose without loss of generality that $w_j^0 = w_j$, and note that, by (2) and (5)

$$V_j(\boldsymbol{w}^0) \geqslant V_j(\boldsymbol{w}) = Q(\boldsymbol{w}) \geqslant Q(\boldsymbol{w}^0).$$

This holds for each j, and so $\boldsymbol{w}^0$ is a feasible vector. By Theorem 1 we can apply the Algorithm to $\boldsymbol{w}^0$ to construct a solution vector $\boldsymbol{w}^\infty = (w_1^\infty, \ldots, w_m^\infty)$ such that

$$0 < w_j^\infty \leqslant w_j^0 \leqslant w_j$$

for each j. We shall prove that $\boldsymbol{w}^\infty = \boldsymbol{w}$, from which it will immediately follow that $\boldsymbol{w} = \boldsymbol{w}^0 = \boldsymbol{w}^*$, as required.

We prove first that $Q(\boldsymbol{w}^\infty) = Q(\boldsymbol{w})$. By (5), $Q(\boldsymbol{w}^\infty) \leqslant Q(\boldsymbol{w})$. By the same argument that is used to derive (5) from (3),

$$\begin{aligned} (s+1)(Q(\boldsymbol{w}) - Q(\boldsymbol{w}^\infty)) = E(\boldsymbol{w}^\infty) - E(\boldsymbol{w}) &\leqslant \sum_{j=1}^{m} (V_j(\boldsymbol{w}) - V_j(\boldsymbol{w}^\infty)) \\ &= m(Q(\boldsymbol{w}) - Q(\boldsymbol{w}^\infty)) \end{aligned}$$

since $\boldsymbol{w}$ and $\boldsymbol{w}^\infty$ are both solution vectors. Since $m \leqslant s$, $Q(\boldsymbol{w}) - Q(\boldsymbol{w}^\infty) \leqslant 0$. Thus $Q(\boldsymbol{w}^\infty) = Q(\boldsymbol{w})$, and

$$V_j(\boldsymbol{w}^\infty) = V_j(\boldsymbol{w}) \tag{8}$$

for each j.

Finally, let S denote the set of candidates C_j (if any) for whom $w_j^\infty < w_j$, and suppose that S is non-empty. Since $w_j^\infty < 1$ for each such j, and $V_j(\boldsymbol{w}^\infty) = Q(\boldsymbol{w}^\infty) > 0$, it is not difficult to see (by considering each ballot paper) that the sum of the votes for all the candidates in S is a *strictly* increasing function of the weight assigned to each such candidate, and so must strictly increase when the vector $\boldsymbol{w}^\infty$ is replaced by $\boldsymbol{w}$. But this violates (8). So S must be empty and $\boldsymbol{w}^\infty = \boldsymbol{w}$. This completes the proof that there can be at most one solution vector $\boldsymbol{w}$. □.

REFERENCES

1. B. L. Meek. Une nouvelle approche du scrutin transférable. *Mathématiques et sciences humaines* **25**, 13–23 (1969).
2. B. L. Meek. Une nouvelle approche du scrutin transférable (fin). *Mathématiques et sciences humaines* **29**, 33–39 (1970).
3. B. A. Wichmann and I. D. Hill. Algorithm AS 183 – an efficient and portable pseudo-random number generator. *Applied Statistics* **31**, 188–190 (1983).
4. D. R. Woodall. Computer counting in STV elections. *Representation* **90**, 4–6 (1982).