PR-STAR: Proportional Representation with Score Then Automatic Runoff Voting

A method for allocating votes and choosing winners in a multi-member election with proportional partisan representation

Trevin Beattie

November 22, 2018

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

The full legal text of this Creative Commons license can be found at
`https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode`

1 Overview

This method is adapted from Score Then Automatic Runoff (STAR) voting and attempts to preserve its strengths in single-seal elections when used in multi-seat elections, specifically:

- Easy for voters to understand and use
- Can be tabulated at the precinct level and aggregated upwards without requiring the original ballots
- Encourages honest voting

In addition, this method strives to ensure that the party preferences of the voters are represented as closely as possible by those who are elected.

2 How Votes Are Counted

2.1 Seat Apportionment

The first phase of this method entails apportioning the available seats among the competing parties. Every candidate will have a party affiliation; for independent candidates, they will be grouped in an "independent" party. For non-partisan races, this first phase can be skipped; all candidates will be considered as members of the same "party".

The highest scoring candidate within each party on a voter's ballot will be weighted by the maximum overall score divided by the sum of the maximum score given in each party, and used as that voter's party score.

For example, if a voter scored 5 points for a candidate in party A and 3 points for a candidate in party B (and no other candidates,) his party score would be $5 \times \frac{5}{8}$ or 3.125 for party A and $3 \times \frac{5}{8}$ or 1.875 for party B.

It is important that party scores be based on candidate scores so that voters cannot influence the proportions of seats without also influencing the choice of candidates, and vice-versa. This helps keep the voters honest and limits strategic voting against an opposing party.

Party scores are tallied across all voters, and the relative strength of these scores is used to determine the number of seats each party receives. Fractional seats will be awarded to parties in order from highest remaining fraction to lowest, until all seats are exhausted.

For example, say we have 7 seats to divide among 4 parties. The parties' cumulative scores are:

Party	Score	Percentage	Seats
A	5,434	5.4%	0.35
B	45,678	45.7%	3.20
C	43,210	43.2%	3.02
D	5,678	5.7%	0.40

The method awards 3 whole seats to each of parties B & C, and the last seat goes to party D.

2.2 Candidate Selections (per party)

Once the number of seats for each party has been determined, then candidates for each party's seats are selected. The score for each candidate is weighted according to the relative score given to the candidate's party versus the others. Following the example given in the previous section, where the voter's raw score was 5 for party A and 3 for party B, the scores for candidates in party A will be weighted by $\frac{5}{8}$ and those in party B will be weighted by $\frac{3}{8}$.

If a party has a single seat, then candidates are paired individually as with STAR voting. If there are multiple seats available to the party, then instead of pairing off individuals, we list and pair off groups of candidates.

Consider for example that party B has 5 candidates — v, w, x, y, and z — to fill 3 seats. There are 10 possible combinations: (v,w,x), (v,w,y), (v,w,z), (v,x,y), (v,x,z), (v,y,z), (w,x,y), (w,x,z), (w,y,z), and (x,y,z). Each combination is given a score of the voter's total score for those candidates, and compared with the scores of other combinations to determine relative rankings. The highest score for each pair of groups is given a weighted vote according to the party weights calculated previously. This weighting ensures that each voter has a total of one vote across all parties.

Note that the number of seats available to each party will not be known in advance of collecting the party scores from all precincts. Thus it may be necessary to determine votes for all possible combinations for any number of seats. For the previous example, this would be 5 individuals, 10 pairs, 10 triplets, and 5 quadruplets. More generally, there are $2^N - 2$ combinations of N candidates, not counting the cases where all or none of the candidates are selected.

Let's look at a complete example of a single voter's ballot. There are 4 parties. Party A has candidates p, q, and r. Party B has candidates w, x, y, and z. The voter will not score any candidate in parties C or D, so we can ignore those candidates. The candidates are scored as follows:

	Candidate	Score
Party A	p	3
	q	0
	r	1

	Candidate	Score
Party B	w	2
	x	5
	y	1
	z	3

His top scores are 3 for party A and 5 for party B. Weighing these, he contributes 1.875 points to party A and 3.125 points to party B.

Within party A, the candidates are scored and weighted:

Candidate	Raw Score	Weighted Score
p	3	1.125
q	0	0
r	1	0.375

The groups are then ordered and given weighted votes of $\frac{3}{8}$:

↓ beats →	(p)	(q)	(r)
(p)		$\frac{3}{8}$	$\frac{3}{8}$
(q)			
(r)		$\frac{3}{8}$	

↓ beats →	(p,q)	(p,r)	(q,r)
(p,q)			$\frac{3}{8}$
(p,r)	$\frac{3}{8}$		$\frac{3}{8}$
(q,r)			

Within party B, the combinations of candidates are weigthed and scored in the same manner, only the weighting is by $\frac{5}{8}$ and he gets $\frac{5}{8}$ of a vote for this party.

Just like in STAR voting, the scores for all candidates and votes for all combinations of candidates in each party can be tallied at the precinct level, then the aggregates collected. Within each party, the two groups with the highest scores are selected, then their total votes against each other are counted to determine the winners for that party.

3 How PR-STAR Compares to STAR Voting

STAR voting is equivalent to a special case of PR-STAR where there is only one party (for the purpose of grouping candidates) and only one seat. Thus when considering the seats apportioned to any given party, this method should most of the same advantages and disadvantages that STAR voting does for single-seat elections. PR-STAR adds the general advantage that proportional representation has over single-seat districts, in that the diversity of constituents is represented more evenly. Its disadvantage over STAR is increased complexity in the form of weighting scores and votes, and of tracking votes for groups of candidates.

4 Known Issues

This method does not address how to handle write-in candidates. Write-ins would need to include their party affiliation in order to properly apportion seats, or else be counted as independents. Each write-in doubles the number of possible combinations of groups in a party.

The vote weighting as currently described penalizes voters whose preferred party does not manage to obtain a seat in the apportionment phase, because it does not take into account how many seats were won. This could cause voters who would prefer a small party that does not have enough critical mass to instead give all their scores to candidates in larger parties.

5 Example Code

The following code example illustrates how votes can be counted using the PR-STAR method. It works in four phases:

1. Declare candidates. This phase establishes which candidates are presented on the ballot and which parties they belong to.

2. Read ballots. This phase reads ballots one at a time consisting of candidate names and associated scores. All ballots read during a single run have their scores added together.

3. Aggregate. This phase adds subtotals from multiple ballot readings to come up with new totals. It may be repeated as often as needed until all precincts have been reported.

4. Score. This phase takes the aggregated totals and number of available seats in the election, and determines how many seats to allocate to each party and which candidates win those seats.

This code should not be used for actual elections — it is for demonstration and evaluation purposes only. Actual election code needs to include secure signature generation to verify the source and authenticity of results, as well as protections against code tampering.

```
#!/usr/bin/perl -w

# Copyright (C) 2018 Trevin Beattie
#
# This program is free software: you can redistribute it and/or modify
```

```
# it under the terms of the GNU General Public License version 3 as
# published by the Free Software Foundation.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# The GNU General Public License can be found at:
# https://www.gnu.org/licenses/gpl-3.0.en.html

=pod

=head1 PR-STAR Vote Counter

Example vote tabulation code
by Trevin Beattie

Four phases of operation:

=over

=item 1

Declare candidates

=item 2

Read ballots

=item 3

Aggregate scores

=item 4

Calculate winners

=back

=cut

use strict;
use File::Slurp;
use Getopt::Long;
use JSON;
```

```
use List::Util qw( max min sum );
use Text::CSV_XS qw( csv );

# Operating modes
use constant {
    HELP    => 'Help',
    DECLARE => 'Declare',
    BALLOTS => 'Ballots',
    AGGREGATE  => 'Aggregate',
    SCORE   => 'Score',
};

my $help;
my $mode = HELP;
my $verbosity = 1;
my $candidate_filename = undef;
my $max_score = 10;
my $output_filename = undef;
my $seat_count = 1;

sub usage {
    print STDERR "Usage: $0 --mode [mode] [options]\n\n";
    print STDERR "'Declare' mode: Reads the names of candidates for a seat.\n";
    print STDERR "\tNames are read in CSV file format. Each line consists\n";
    print STDERR "\tof the candidate name followed by the party name.\n\n";
    print STDERR "'Ballots' mode: Reads ballot entries from the given files.\n";
    print STDERR "\tBallots are read in CSV file format. The first line MUST\n";
    print STDERR "\tconsist of a unique ballot ID which will be used to verify\n";
    print STDERR "\twhich ballots were counted, and guard against duplicates.\n";
    print STDERR "\tThe remaining lines consist of a candidate name (which
        must\n";
    print STDERR "\thave been declared previously) followed by a score.\n\n";
    print STDERR "'Aggregate' mode: Reads files output from the 'Ballot'
        and/or\n";
    print STDERR "\tanother 'Aggregate' step. Verifies the uniqueness of all\n";
    print STDERR "\tballot ID's and adds the scores and votes together\n\n";
    print STDERR "'Score' mode: Reads files output from the 'Aggregate' step.\n";
    print STDERR "\tDetermines the number of seats to give to each party and\n";
    print STDERR "\twhich candidates in each party have won those seats.\n\n";
    print STDERR "Options:\n";
    print STDERR "\t--candidates [file]\n";
    print STDERR "\t\tGive the name of the candidate file generated in the\n";
    print STDERR "\t\tDeclare' step. Used in all modes except 'Declare'\n";
    print STDERR "\t--output-file [file]\n";
    print STDERR "\t\tGive the name of the file in which to write the results.\n";
    print STDERR "\t\tIf not provided, results will be written to stdout.\n";
```

```
    print STDERR "\t--max-score [n]\n";
    print STDERR "\t\tThe maximum possible score on a ballot.\n";
    print STDERR "\t\tShould be provided in 'Declare' mode.\n";
    print STDERR "\t--seats [n]\n";
    print STDERR "\t\tThe total number of seats available in this race.\n";
    print STDERR "\t\tOnly used in the 'Score' step.\n";
}

GetOptions(
    'help' => \$help,
    'mode=s' => \$mode,
    'verbose:+' => \$verbosity,
    'candidates=s' => \$candidate_filename,
    'max-score=i' => \$max_score,
    'output-file=s' => \$output_filename,
    'seats=i' => \$seat_count,
    ) or do {
    usage();
    exit(1);
};

if ( $help || ( $mode eq HELP ) ) {
    usage();
    exit(0);
}

# Write a JSON file
sub write_json($;$;$) {

    my $description = shift;
    my $data = shift;
    my $filename = shift;
    my $outfile = *STDOUT;

    if ( defined( $filename ) ) {
    open( $outfile, '>', $filename ) or do {
        warn "$filename: $!";
        print STDERR "Writing $description to stdout\n";
        $outfile = *STDOUT;
        $filename = undef;
    };
    }

    my $json_data = JSON->new->latin1( 0 )->pretty->encode( $data );

    print $outfile $json_data;
```

```
    if ( $filename ) {
    close( $outfile );
    }

}

# Read a JSON file
sub read_json($;$) {

    my $description = shift;
    my $filename = shift;
    die "No $description file given" unless ( defined( $filename ) );

    my $json_data = read_file( $filename, binmode => ':utf8' )
    or die "$filename: $!";

    my $data = decode_json( $json_data );
    return $data;

}

# Read a candidate file and verify its contents
sub read_candidates() {

    my $candidates = read_json( 'candidate', $candidate_filename );

    die "No parties found in $candidate_filename"
    unless ( exists( $candidates->{'parties'} ) );
    die "No names found in $candidate_filename"
    unless ( exists( $candidates->{'names'} ) );

    if ( $candidates->{'max_score'} ) {
    $max_score = $candidates->{'max_score'};
    }

    return $candidates;

}

# Count the number of bits set in a number.
# We use a lookup table to cache the results.
my @bit_counts = ( 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4 );
sub bit_count($) {

    my $x = shift;
```

```
    while ( $x >= scalar( @bit_counts ) ) {
    my $h = scalar( @bit_counts );
    for ( my $i = 0; $i < $h; $i++ ) {
        $bit_counts[$h+$i] = $bit_counts[$i] + 1;
    }
    }

    return $bit_counts[$x];

}

# Read candidate scores from a ballot file.
# Returns an 'aggregate' of a single ballot. Much of the
# scoring and weighting logic happens in this subroutine.
sub read_ballot(*;$) {

    my $infile = shift;
    my $candidates = shift;
    my $csv = Text::CSV_XS->new({ binary => 1 });

    # First row MUST be the ballot ID
    my $row = $csv->getline( $infile );
    unless ( $row->[0] =~ /^[0-9a-f]{8}(-[0-9a-f]{4}){3}-[0-9a-f]{12}$/i ) {
    warn "Invalid ID: " . join( ',', @$row );
    return;
    }
    my $ballot_id = lc( $row->[0] );
    my %party_weights = ();
    my %party_scores = ();
    my %candidate_scores = ();
    my %group_scores = ();
    my %votes = ();
    my $local_max = 1;

    while ( $row = $csv->getline( $infile ) ) {

    my $person = $row->[0];
    chomp( $person );
    my $score = $row->[1];

    unless( exists( $candidates->{'names'}{$person} ) ) {

        # Check for the same name with a different case
        foreach my $real_name ( keys %{$candidates->{'names'}} ) {
        if ( CORE::fc( $real_name ) eq CORE::fc( $person ) ) {
```

```
        $person = $real_name;
        last;
    }
    }

    unless ( exists( $candidates->{'names'}{$person} ) ) {
    warn "No candidate found by name '$person'";
    return;
    }

}

$candidate_scores{$person} = $score;

$local_max = max( $local_max, $score );

my $party = $candidates->{'names'}{$person}{'party'};

# Set the party score to the highest score of candidates in the party
$party_scores{$party} ||= 0;
$party_scores{$party} = max( $party_scores{$party}, $score );

# Add this candidate's score to all candidate
# groups of which this candidate is a member
my $groups = $candidates->{'parties'}{$party}{'groups'};
my $bit = $candidates->{'names'}{$person}{'bit'};
for ( my $n = 1; $n < scalar( @$groups ); $n++ ) {
    next unless ( $n & $bit );
    $group_scores{$party}[$n] ||= 0;
    $group_scores{$party}[$n] += $score;
}

}

# Total the scores for each party then weight
# them so their sum is equal to the maximum score
my $parties_total = sum( values %party_scores );
foreach my $party ( keys %party_scores ) {
$party_weights{$party} = $party_scores{$party} / $parties_total;
$party_scores{$party} = min( $max_score, $local_max )
    * $party_weights{$party};


# Determine the vote given for each pairing of groups in each party
for ( my $m = 1; $m < scalar( @{$group_scores{$party}} ); $m++ ) {
    for ( my $n = 1; $n < scalar( @{$group_scores{$party}} ); $n++ ) {
```

```
        next if ( $m == $n );
        # Only compare groups of the same size -- that means counting bits
        next unless ( bit_count( $m ) == bit_count( $n ) );
        my $m_score = $group_scores{$party}[$m] || 0;
        my $n_score = $group_scores{$party}[$n] || 0;
        # Ignore ties
        next if ( $m_score == $n_score );
        if ( $m_score > $n_score ) {
            $votes{$party}[$m][$n] = $party_weights{$party};
        } else {
            $votes{$party}[$n][$m] = $party_weights{$party};
        }
        }
    }

    }

    # Weight the candidate scores by their parties' weights
    foreach my $person ( keys %candidate_scores ) {

    my $party = $candidates->{'names'}{$person}{'party'};
    $candidate_scores{$person} = min( $max_score, $candidate_scores{$person} )
        * $party_weights{$party};

    }

    return {
    'ballots' => { $ballot_id => 1 },
    'parties' => \%party_scores,
    'candidates' => \%candidate_scores,
    'votes' => \%votes,
    };

}

# Add two aggregate ballots together.
# Aborts if it finds duplicate ballots.
sub add_aggregates($;$) {

    my $agg1 = shift;
    my $agg2 = shift;

    foreach my $ballot_id ( keys %{$agg2->{'ballots'}} ) {
    die "Duplicate ballot detected: $ballot_id"
        if ( exists( $agg1->{'ballots'}{$ballot_id} ) );
```

```
    $agg1->{'ballots'}{$ballot_id} = $agg2->{'ballots'}{$ballot_id};
    }

    foreach my $party ( keys %{$agg2->{'parties'}} ) {
    $agg1->{'parties'}{$party} ||= 0;
    $agg1->{'parties'}{$party} += $agg2->{'parties'}{$party};
    }

    foreach my $person ( keys %{$agg2->{'candidates'}} ) {
    $agg1->{'candidates'}{$person} ||= 0;
    $agg1->{'candidates'}{$person} += $agg2->{'candidates'}{$person};
    }

    foreach my $party ( keys $agg2->{'votes'} ) {
    for ( my $i = 1; $i < scalar( @{$agg2->{'votes'}{$party}} ); $i++ ) {
        next unless ( defined( $agg2->{'votes'}{$party}[$i] ) );
        for ( my $j = 1; $j < scalar( @{$agg2->{'votes'}{$party}[$i]} ); $j++ ) {
        next unless ( defined( $agg2->{'votes'}{$party}[$i][$j] ) );
        $agg1->{'votes'}{$party}[$i][$j] ||= 0;
        $agg1->{'votes'}{$party}[$i][$j] += $agg2->{'votes'}{$party}[$i][$j];
        }
    }
    }

    return $agg1;

}

=pod

=head2 Declare Candidates

Reads a file of candidates for a multi-member office and their party
affiliations. Defines the data structure that will be used to tally
votes for these candidates.

=cut

if ( $mode eq DECLARE ) {

    my $infile;
    if ( $ARGV[0] ) {
    open( $infile, '<', $ARGV[0] )
        or die "$ARGV[0]: $!";
    } else {
    $infile = *STDIN;
```

```
}

my $csv = Text::CSV_XS->new({ binary => 1 });
my %candidates = (
'names' => {},
'parties' => {},
);

while ( my $row = $csv->getline( $infile ) ) {

my $person = $row->[0];
chomp( $person );
my $party = $row->[1];
chomp( $party );

next unless ( $person );
unless ( $party ) {
    warn "No party declared for $person; assigning to 'Independent'";
    $party = 'Independent';
}

$candidates{'names'}{$person} = {
    'party' => $party,
};
$candidates{'parties'}{$party}{'members'}{$person} = {};

}

if ( $ARGV[0] ) {
close( $infile );
}

# Remove duplicates and assign numeric values to candidates
foreach my $party ( keys %{$candidates{'parties'}} ) {

my @members = sort { CORE::fc($a) cmp CORE::fc($b) }
keys %{$candidates{'parties'}{$party}{'members'}};

for ( my $i = 0; $i < scalar( @members ); $i++ ) {

    # Remove duplicates
    while ( ( $i + 1 < scalar( @members ) ) &&
        ( CORE::fc( $members[$i] ) eq CORE::fc( $members[$i+1] ) ) ) {
    delete( $candidates{'names'}{$members[$i+1]} );
    splice( @members, $i+1, 1 );
    }
```

```
        # Assign numeric values to candidates
        $candidates{'names'}{$members[$i]}{'bit'} = 2 ** $i;

    }

    $candidates{'parties'}{$party}{'members'} = \@members;

    # Determine groupings
    my $last_group = 2 ** scalar( @members ) - 1;
    for ( my $n = 1; $n < $last_group; $n++ ) {

        my @group = ();
        for ( my $i = 0; $i < scalar( @members ); $i++ ) {
        if ( $n & ( 2 ** $i ) ) {
            push @group, $members[$i];
        }
        }

        $candidates{'parties'}{$party}{'groups'}[$n] = \@group;

    }

    }

    $candidates{'max_score'} = $max_score;

    write_json( 'candidate file', \%candidates, $output_filename );

}

=pod

=head2 Read Ballots

Reads an arbitrary number of files containing individual votes for
candidates. Each file must contain a single ballot. The scores for
the ballot are weighted and added to the data structure defined by the
candidate declaration. A single file is written out containing the
subtotal of scores from all ballots read in this pass.

=cut

elsif ( $mode eq BALLOTS ) {

    my $infile;
```

```
    my $candidates = read_candidates();
    my %subtotals = (
    'ballots' => {},
    'parties' => {},
    'candidates' => {},
    'votes' => {},
    );

    if ( $ARGV[0] ) {

    foreach my $filename ( @ARGV ) {
        open( $infile, '<', $filename )
        or die "$filename: $!";

        my $ballot_data = read_ballot( $infile, $candidates );

        close( $infile );

        if ( $ballot_data ) {
        add_aggregates( \%subtotals, $ballot_data );
        } else {
        warn "$filename: Not counted";
        }
    }

    }

    else {
    my $ballot_data = read_ballot( *STDIN, $candidates );

    die unless ( $ballot_data );

    %subtotals = %$ballot_data;
    }

    write_json( 'aggregated ballots', \%subtotals, $output_filename );

}

=pod

=head2 Aggregate Scores

Reads an arbitrary number of files containing subtotal scores
generated from reading ballots and adds them together. Writes out a
single file containing their total, in the same format.
```

```
=cut

elsif ( $mode eq AGGREGATE ) {

    my $candidates = read_candidates();
    my %subtotals = (
    'ballots' => {},
    'parties' => {},
    'candidates' => {},
    'votes' => {},
    );

    foreach my $filename ( @ARGV ) {
    open( my $infile, '<', $filename )
        or die "$filename: $!";

    my $aggr_data = read_json( 'aggregated ballots', $filename );

    if ( $aggr_data ) {
        die "No ballot ID's found in $filename"
        unless ( exists( $aggr_data->{'ballots'} ) );
        # All other sections depend on there being a vote.
        # Voters are not required to score any candidates.

        add_aggregates( \%subtotals, $aggr_data );
    } else {
        warn "$filename: Not counted";
    }
    }

    write_json( 'aggregated ballots', \%subtotals, $output_filename );

}

=pod

=head2 Calculate Winners

Takes the total number of available seats as a parameter. Reads a
score file generated by reading the ballots or aggregates scores.
Calculates the number of seats to allocate to each party, then for
each party which has seats determines the winning candidate(s) by
taking the top two scores and then choosing the one with the most
votes.
```

```
=cut

elsif ( $mode eq SCORE ) {

    my $candidates = read_candidates();
    die "Exactly one aggregated ballots file must be provided"
    unless ( scalar( @ARGV ) == 1 );

    my $aggr_data = read_json( 'aggregated ballots', $ARGV[0] );
    die unless ( $aggr_data );

    die "No ballot ID's found in $ARGV[0]"
    unless ( exists( $aggr_data->{'ballots'} ) );

    # Determine the number of seats to allocate to each party
    my $total_score = sum( values %{$aggr_data->{'parties'}} );
    die "No confidence in any party" unless ( $total_score > 0 );

    my %party_seats = ();
    my %party_fractions = ();
    my $remaining_seats = $seat_count;
    foreach my $party ( keys %{$aggr_data->{'parties'}} ) {
    $party_fractions{$party} = $seat_count *
        $aggr_data->{'parties'}{$party} / $total_score;
    $party_seats{$party} = int( $party_fractions{$party} );
    $party_fractions{$party} -= $party_seats{$party};
    $remaining_seats -= $party_seats{$party};
    delete $party_fractions{$party} if ( $party_fractions{$party} == 0 );
    }

    while ( ( $remaining_seats > 0 ) && scalar( keys %party_fractions ) ) {
    my @ordered_fractions = sort {
        $party_fractions{$b} <=> $party_fractions{$a}
    } keys %party_fractions;
    $party_seats{$ordered_fractions[0]}++;
    delete $party_fractions{$ordered_fractions[0]};
    $remaining_seats--;
    }

    print "Allocated " . ( $seat_count - $remaining_seats ) . "/$seat_count seats
        as follows:\n";
    foreach my $party ( sort { CORE::fc($a) cmp CORE::fc($b) }
           keys %party_seats ) {
    print "\t$party \t$party_seats{$party}\n";
    }
```

```
# For each party, determine the winning candidates
foreach my $party ( sort keys %party_seats ) {
print "\n$party Party winners: ";

if ( $party_seats{$party} == 0 ) {
    print "NO SEATS (Only got " .
    ( int( $aggr_data->{'parties'}{$party} * 1000 / $total_score ) / 10 )
    . "% of the total score)\n";
    next;
}

my @members = sort { CORE::fc($a) cmp CORE::fc($b) }
@{$candidates->{'parties'}{$party}{'members'}};
if ( $party_seats{$party} >= scalar( @members ) ) {
    print join( ', ', @members );
    if ( $party_seats{$party} > scalar( @members ) ) {
    print " (" . ( $party_seats{$party} - scalar( @members ) )
        . " unfilled seats)";
    }
    print "\n";
    next;
}

my %scores = ();
my $last_group = 2 ** scalar( @members ) - 1;
for ( my $n = 1 ; $n < $last_group; $n++ ) {
    next unless ( bit_count( $n ) == $party_seats{$party} );
    foreach my $name ( @members ) {
    if ( $n & $candidates->{'names'}{$name}{'bit'} ) {
        $scores{$n} ||= 0;
        $scores{$n} += $aggr_data->{'candidates'}{$name};
    }
    }
}

my @top_groups = sort {
    $scores{$b} <=> $scores{$a}
} keys %scores;

if ( scalar( @top_groups ) < 2 ) {
    warn "ERROR! Unable to determine top scoring candidates";
    next;
}

my $group_A = $top_groups[0];
my $group_B = $top_groups[1];
```

```
    my $winning_group;
    my $winning_score;
    my $winning_vote;
    my $losing_score;
    my $losing_vote;
    if ( $aggr_data->{'votes'}{$party}[$group_A][$group_B]
         > $aggr_data->{'votes'}{$party}[$group_B][$group_A] ) {
        $winning_group = $group_A;
        $winning_score = $scores{$group_A};
        $losing_score = $scores{$group_B};
        $winning_vote = $aggr_data->{'votes'}{$party}[$group_A][$group_B];
        $losing_vote = $aggr_data->{'votes'}{$party}[$group_B][$group_A];
    } else {
        $winning_group = $group_B;
        $winning_score = $scores{$group_B};
        $losing_score = $scores{$group_A};
        $winning_vote = $aggr_data->{'votes'}{$party}[$group_B][$group_A];
        $losing_vote = $aggr_data->{'votes'}{$party}[$group_A][$group_B];
    }

    my @winners = ();
    foreach my $name ( @members ) {
        if ( $winning_group & $candidates->{'names'}{$name}{'bit'} ) {
        push @winners, $name;
        }
    }

    print join( ', ', @winners )
        . " (score: " . ( int( $winning_score * 100 ) / 100 )
        . "; votes: " . ( int( $winning_vote * 10 ) / 10 )
        . " to " . ( int( $losing_vote * 10 ) / 10 ) . ")\n";

    }

}

else {
    usage();
    exit(1);
}
```