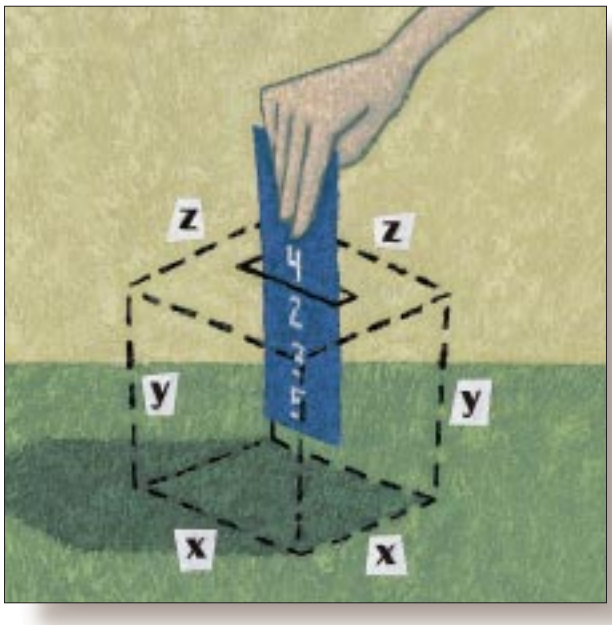
by Jeffreys Copeland and Haemer



Refinement for Efficiency's Sake

Jeffrey Copeland *(copeland@alumni.caltech.edu) lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats, and being a thorn in the side of his local school board.*

Jeffrey S. Haemer *(jsh@usenix.org) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work.html`.

Even though we never do anything but read and write code, we're surrounded by people who have other interests. Copeland's wife, for example, has been a long-standing member of a group of fellow southerners who share an interest in science fiction. It's an amusingly diverse group, including lawyers, government bureaucrats, technology people, editors and writers. This past year, it was her turn to take the post of the group's secretary, and among her responsibilities is to conduct a poll that determines the group's mostly ceremonial president. Because nothing in such an offbeat group is simple, the poll has a complicated ballot, and she turned to us to write some software to tally the results.

This isn't the most complicated ballot we've ever seen in terms of just determining the winner of an election. (For that, we turn to the preferential ballots used for selecting winners of the Hugo Awards, and seats on the Cambridge, MA, city council.) The interesting feature of this ballot is that it gives everyone a chance to rank everyone else's performance over the past year in a number of categories suggested by the secretary and even some she hadn't thought of. For example, this year's ballot included a half-dozen categories–for argument, let's call them congeniality, talent, swimsuit, sex, drugs and rock 'n' roll–and then the kicker, a large blank space called "roll your own," in which the voter gets to suggest a category (usually amusing, and at the expense of someone's dignity) undreamt of by the secretary. Each voter assigns points to each member in each category, including the invented ones, according to some limits. (The limits aren't important, but you can think of the obvious schemes: distribute a total of 200 points across all members in all categories; give no more than 20 points to an individual; rank each category for each

Example 1. Sample Ballot File

```
Brown    5  5  5  5  5  5  A most professional amateur   5
Hlavaty  4  5  4  2  5  4  Cannot be second-guessed      5
weber    4  2  2  1  4  4  Irregular but persistent      4
Wells    5  5  4  1  5  5  Old but large cheerleader     5
```

Listing 1. Our First Shell Script

```
1   #! /bin/sh
2   # $Id: tally,v 1.1 1997/12/19 15:56:01 jeff Exp $

3   # first get a list of the victims from the ballots
4   ls ballots/* | xargs cut -d " " -f 1 | sort -u >/tmp/$$.names

5   # now for each category, total 'em up
6   # (categories start in field 2 on the ballot)
7   fld=2
8   for category in Congeniality Talent Swimsuit Sex Drugs Rock-n-Roll
9   do
10     echo "\\\\category{$category}"
11     cat /tmp/$$.names | while read who
12     do
13      cat ballots/* | egrep $who |
14        cut -d " " -f 1,$fld |
15        awk ' { name=$1; total=total+$2 }
16        END { printf "\\nm %s: %d.\n", name, total }'
17     done | sort +2rbn
18     echo "\\\\endcat"; echo
19     fld=$(expr $fld \+ 1)
20  done

21  #     roll your own is a special case
22  #        (fld now points at the stuff trailing off the end)
23  echo "\\\\rollyourown"
24  cat /tmp/$$.names | while read who
25  do
26     cat ballots/* | egrep $who |
27      cut -d " " -f $fld- |
28      awk -v who=$who '
29        BEGIN { total = 0; }
30        { for(i=1; i<=NF; i++) total+=$i; }
31        END { print who, total }'
32  done | sort +1brn |
33  while read who total
34  do
35     echo "\\\\roll $who: $total."
36     cat ballots/* | egrep $who |
37      cut -d " " -f $fld- |
38      sort -fd | grep . |
39      sed -e 's/[0-9]$/(&);/' -e '$s/;$/./'
40     [ $total -eq 0 ] && echo "\\\\sorry"
41  done
42  echo "\\\\endcat"; echo

43  #   grand totals
44  echo "\\\\category{Grand Totals}"
45  cat /tmp/$$.names | while read who
46  do
47     cat ballots/* | egrep $who |
48      awk ' { for(i=2; i<=NF; i++) total=total+$i; name=$1; }
49      END { printf "\\nm %s: %d.\n", name, total }'
50  done | sort +2rbn
51  echo "\\\\endcat"; echo
52  echo "\\\\bye"

53  rm -f /tmp/$$.names

54  exit
```

member from zero to five.) In other words, every member is both a candidate and a potential voter.

From our data-munging point of view, the important thing is that you begin with a stack of ballots with some points assigned to each member in a number of categories. In practice, we need to report the results of each category separately, and especially to individually report each made-up category in order to puncture inflated egos and ladle abuse back on the practical jokers. What we want to show you today is the process we used to develop code for the main task of tallying all the votes.

Why are we looking at this particular problem? Because we got it wrong the first couple of tries. Once we had correct code, we turned it into elegant code, and once we had done that, it turned out to be pretty efficient, too.

First Try

First, we need to talk about the structure of the input. For convenience, think of the ballots as a three-dimensional array. The categories run along the x-axis, the members (as candidates) run along the y-axis and, finally, the members (as voters) run along the z-axis. We enter each ballot into a new file in a subdirectory, `ballots`. Each file consists of a line for each member, with the vote for each category, followed by the made-up categories, with their totals. A ballot file might contain lines such as those shown in Example 1. The whole of our first version–a shell script–is shown in Listing 1.

We begin with the usual shebang and IDs in lines 1 and 2. Normally, we'd follow these with a short description of what the program does, but we've removed it to save space here. On line 4, we gather a list of the names on the ballots. This list will be used later to drive the totals by category.

The next block of code (lines 8 through 20) is a `for` loop that runs across all the categories, totaling that column of the ballots by name. Inside the loop, we start by printing

the category name–note that we're producing output interspersed with TeX macros, pushing the formatting off onto a program that's geared for that task. Line 11 starts us looping over the names, cutting the given column and totaling it with awk. We finish the inner loop by sorting the results top-to-bottom, and inserting a macro to end the category. Our output consists mostly of lines using a macro for printing pairs of names and points, thus:

```
\nm Lillian: 39.
```

We begin the complicated roll-your-own processing on line 23. By the time we get here, we've counted up the number of simple columns in the variable fld. Again, we loop over the names (lines 24 to 32), totaling the points for each of the invented category names for each person. We sort those totals top-to-bottom on line 32.

Now that we have the totals, we want to print them out, but also to print out the invented category names. So, we read the categories again in a second loop, which prints not only the totals, but the invented category names themselves. For grins, we add parentheses around the one-digit number of points assigned on line 39, and replace the terminal semicolon with a period. We also have a way of handling the case where someone ends up with no invented categories, on line 40. (Notice that we provide some randomizing while printing out the categories: We sort them on line 38, so that they are not in the order of the ballot files, and the identity of the person who called Brown "a most professional amateur" is somewhat obscured.) This double loop–one feeding into the next–is inefficient, but necessary, because we need to collect the totals and sort them before we can print the names of the invented categories themselves. Perhaps there's another way we can collect the totals before printing? We'll explore this in a later version.

We finish this version by collecting the grand totals. Lines 44 through 50 are fairly straightforward. Again, we loop through the names in a while loop, adding together all numeric fields using awk.

Note that we've introduced a bug in the way we total the roll-your-own votes. The bug exists on lines 30 and 48. In the case of a line such as

```
Weisskopf 3 4 4 2 4 5 Back to the Motherland 3
```

the code will work, but given the line

```
Simon 5 3 4 3 3 5 50 ways to leave your lover 4
```

Simon would find himself with an extra 50 points.

Abortive Second Attempt

Occasionally, we try to make something smoother and fail miserably. Our first thought is to clean up the loop for totaling the categories. If we use bc instead of awk to total the ballots, the code may be simpler. We replace lines 13 to 16 of our original with the following:

```
total=$(cat ballots/* | egrep $who |
   cut -d " " -f $fld | fmt |
   sed "s/ / + /g" | bc)
echo "\\\\nm $who: $total."
```

We take a vertical slice through ballots, and end up composing a line like

```
4 + 3 + 5 + 2 + 4 + 5
```

which is then fed to bc.

Dutifully, we test this modification with time, and discover that the new version is 20% *slower* than the previous one. We remind ourselves that we can't be right all the time and toss this attempt overboard.

Third Try

We have two bottlenecks in the previous versions. The first involves reading each ballot from each ballot file to total each of the known categories–that's voters *times* ballots *times* categories reads. The other, of course, is the double loop to total and then enumerate the made-up roll-your-own categories–notice that it reads all the ballots in both loops, for voters *times* ballots *times* 2 reads.

We can fix both problems at once by doing a bit of preprocessing. If we make a single pass through all the ballots, totaling each column for each member, we can cut out all of the repetitive data shuffling. We will only need to read each ballot a single time for the main categories, plus another time to collect the names of the roll-your-own categories. To do this, we extract the ballot data into a file named totals with the following code fragment:

```
# total up everything in the
#  ballot files in a big loop
sort ballots/* |
sed "s/  */ /g" |
awk -v categories=6 '
 function showtotals()
  { # print out this guy's totals
  printf "%s", lastname;
  sum = 0;
  for(j=1; j<=(nc+1); j++)
  {
    printf " %d", total[j];
    sum += total[j];
    total[j]=0;
  }
  printf " %d\n", sum;
  }
 BEGIN {lastname = "zzz"; nc = categories; }
  { split($0, input);
  if( lastname != "zzz"
    &&  input[1] != lastname )
      showtotals();
  for(j=1; j<=nc; j++)
    total[j] += input[j+1];
```

```
  #  now cut off the fixed categories,
  #  and split the rest
  roll = substr($0, length(input[1])+1+nc*2);
  n = split(roll, inputb, ";");
  for(j in inputb) {
   nn = split(inputb[j], words);
   total[nc+1] += (words[nn]+0);
  }
  lastname = input[1];
  }
 END {showtotals(); }
' >totals
```

Notice that we don't hardwire the number of categories into the script, but provide it as a variable in a command-line argument. We use a function `showtotals()` to print out the total when the name on the ballot changes. This function also conveniently prints a grand total of points.

The body of the `awk` program is a fairly straightforward loop through the ballots. Once we have totaled the fixed columns, we use `substr` to collect the text of the roll-your-own votes at the end of each line. We `split` on semicolon so that multiple votes like

```
Best of the Year: Chall 3; Missing Turkey 2
```

are put into separate elements of the array `inputb[]` array. We then carefully extract the last item–the points–from each element and add it to the total array. (This solves the bug we introduced in our original version, in which we tallied an extra 50 points for Mr. Simon.)

Given a `totals` file, containing a summary of all the ballots, the rest of the task is simple. We simply `cut` and `sort` for each fixed category. Using a pipe such as

```
cut -d" " -f1,$fld totals |
 sort +1bnr |
 sed 's/\(.*\) \(.*\)/\\nm \1: \2./'
```

We eschew our roll-your-own double loop for a similar `grep`, `cut` and `sort`, except that this one is driven by the sorted totals for the invented categories. To end up with the grand totals, we need only sort the output from our earlier `awk` program: The totals are already done for us!

Is it faster? Much: `time` shows us that this version executes about six times faster than the first version.

But There's More

A speed increase of six times is nice, but in this case, it's at the cost of a pretty complicated supplementary program in `awk`. We can perhaps do a little better by recoding the whole exercise in Perl. We won't develop the whole tally program here but will just show the code for the hard part: extracting and totaling the points, and printing the sorted grand totals (see Listing 2).

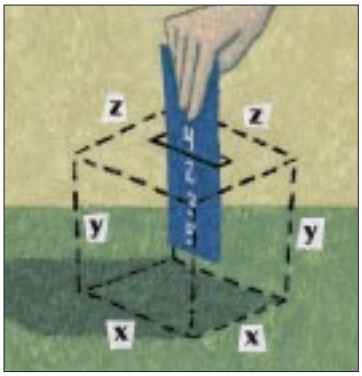
We begin our task by naming the categories in the `@category` array. While reading each ballot line in the loop, beginning on line 4, we split up

Listing 2. Extract from Our Perl Program

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

2  @category = ( 'congeniality', 'talent', 'swimsuit',
3     'sex', 'drugs', 'rock&roll', 'misc' );

4  while (<>) {
5     $voter = $ARGV;
6     ($candidate, @score) = split /\s+/, $_, @category+1;

7     # initialize the total the first time we see this candidate
8     if( ! grep /$candidate/, keys %total ) {
9       $total{$candidate} = 0;
10    }

11    {
12      local @category = @category;
13      while (@score) {
14        $category = shift @category;
15        $score = shift @score;
16        %rankings = (%rankings, $category => $score);
17        }
18    }

19    $v->{$voter}{$candidate} = {%rankings};
20    $y = $v->{$voter}{$candidate}{misc};
21    @misclist = split /;/, $y;
22    $miscsum = 0;
23    while( @misclist ) {
24        $item = shift @misclist;
25        $item =~ s/.*\s+(\d)/$1/;
26        $miscsum += $item;
27    }

28   foreach $cat (@category) {
29     $total{$candidate} += $v->{$voter}{$candidate}{$cat}
30       if( $cat ne 'misc' );
31    }
32    $total{$candidate} += $miscsum;
33    }

34    foreach $cand (sort { $total{$b} <=> $total{$a} } keys %total) {
35      print "TOTAL: $cand $total{$cand}\n";
36    }
```

the input lines, as we did in the previous versions. Here, we also make sure to initialize the associative array of totals by candidate (see line 9). This is partly why we use the `-w` flag to Perl: It allows Perl to warn us that we are adding to an uninitialized variable. We loop through the point scores for each category, putting them in associative array `rankings` in the `while` loop on line 13.

We insert these rankings into a massive three-dimensional associative array, `v` (line 19), representing the three-dimensional data structure we discussed earlier. In lines 21 through 27, we carefully extract the points for the roll-your-own categories and total them. Given the data in this array, we can gather a running total of points for each candidate, as shown in lines 28 through 32.



Once we've fallen out of the main reading loop at line 33, we can sort and print the totals. We loop over the sorted keys to the associative array, `totals`, using a technique explained on the `perlfunc` man page.

By this point, you know what we're going to ask. How much faster? If we strip out everything from the last shell script except gathering and printing the grand totals, the new Perl version runs at about the same time as the shell version. By now, the time for our code to execute is swamped by the time to get the ballot data off the disk and write out the results. So, at this point, it's a judgment call whether you want to use the Perl code or the `awk` version. Two considerations that may come into play in that decision are the relative sizes of the code (the shell version is a bit longer, but it could be made a little shorter) and the debugging facilities (Perl's are better).

Conclusions

It's possible to write tricky code that doesn't work, or worse, is less efficient than the simpler version. Our code, like our theories, should follow Occam's razor.

We ran across a program fragment by an eminent coder the other day. In it, he'd done a time conversion in a nonobvious way, which took us half-an-hour to untangle. In his defense, he had intended the code to be a throw-away, but we should remember that clear code needs to be one of our goals, too.

It is possible to write code that's elegant, correct, clear and efficient. Sometimes, we can achieve another of these goals by reworking our algorithms, or by recoding in a different language. Like any good worker, we need to know what we have in our toolbox, and which tool is appropriate for which job.

Until next time, happy trails.