Instant Runoff Voting

Kartik Mohanram, University of Pittsburgh

In this project, you will implement one solution to the instant runoff voting[1] method (abbreviated IRV henceforth) for preferential voting that has seen widespread adoption in multiple contexts/scenarios throughout the world. I can think of no better way to get you up-to-speed than to point you to the Wikipedia entry on IRV.

In particular, once you grasp the generalities of IRV and its variants after a screen or two of reading, please proceed to the "Five voters, three candidates" example that will form the basis for introducing your class project. This example demonstrates full preferential voting, i.e., every voter assigns every candidate a unique rank/preference.[2] This class project requires that you implement IRV in its full preferential voting form assuming that the data is well-formed, i.e., you may assume that there are no spoilt/invalid votes. Once you understand and maybe even formulate a simple process for implementing the algorithm outlined there, please return to this document to understand the input/output format and the general programming framework that has been released to the class. Simultaneously, I suggest that you also open and read the file `main.asm` that will serve as the single source file that integrates a simple test harness to get you going. **All your code, with embedded documentation, will go into this single file, which you will submit by the project deadline.**

[1] **in·stant run·off vot·ing** *noun* : an electoral system whereby voters rank candidates in order of preference. In the event that one candidate fails to achieve a sufficient majority, the candidate with the fewest number of first-preference rankings is eliminated and these votes redistributed, the process being repeated until one candidate achieves the required majority.

[2] Wikipedia: In a full preferential voting method, voters are required to mark a preference for every candidate standing. Ballots that do not contain a complete ordering of all candidates are in some jurisdictions considered spoilt or invalid, even if there are only two candidates standing.

1 *Input+output format*

The test data (input as well as output) is located in the files `test1.asm`, `test2.asm`, etc. and a single test file will be explicitly referenced from `main.asm` on program execution. The rest of this section uses the Wikipedia example (reproduced below) to orient you to the specifics of the test file format.

	Round 1						Round 2					
Candidate	a	b	c	d	e	Votes	a	b	c	d	e	Votes
Bob	1	2	3	1	2	2	1	2	2	1	2	*2*
Sue	3	1	2	3	1	2	2	1	1	2	1	**3**
Bill	2	3	1	2	3	*1*						

Figure 1: Wikipedia IRV example

For the example outlined on Wikipedia, your `test1.asm` file uses the following format and relies on three variables: `votes`, `results`, and `golden_results`.

```
.data
votes: .word 5,3,1,3,2,2,1,3,3,2,1,1,3,2,2,1,3
results: .word 0,0,0,0
golden_results: .word 1,3,2,1
```

The first word at the `votes` variable is the number of voters. The second word at the `votes` variable is the number of candidates. The preferential votes of all the voters are then setup as a flattened two-dimensional array, and the rest of the entries in the `votes` array define the votes (preferences) of each voter in turn/sequence. For example, voter a's preferences read `1,3,2` following the entries `5,3`. You should interpret this to mean that voter a rank's candidate 1 first, candidate 2 third, and candidate 3 second, consistent with the Wikipedia example.[3]

You are expected to record the winner and loser of each round till the algorithm terminates (output) using pre-allocated space at the `results` variable. Note that for n candidates, you will have a maximum of $n-1$ rounds, each with a winner and a loser, and hence there will be $2(n-1)$ zeroes against this variable. If the algorithm terminates in less than $n-1$ rounds, then the extra entries are to be left as-is. For your benefit, the expected (i.e., golden) results are also provided in the `golden_results` location. For the Wikipedia example, the winner of the first round is candidate 1 while candidate 3 is eliminated. On round two, candidate 2 is the winner while candidate 1 is eliminated. It is your responsibility to craft additional test cases beyond the four tests released in this package[4]. Each of the remaining three test cases described/discussed in the rest of this section illustrate some salient

[3] A common trap is to interpret `1,3,2` to mean that voter a prefers candidates 1, 3, and 2 in that order; while this input format and interpretation is definitely possible (with consistent/equivalent outcomes as in this particular case), it is not compatible with a general IRV setup in practice, since names are committed to the ballot with boxes in which ranks are to be recorded. Thus, `3,1,2` means the voter ranks the candidates in the order 2, 3, and then 1, and not in the order 3, 1, and 2.

[4] Please note that your documentation/report should explicitly clarify what additional tests, however simple, were crafted to test your implementation.

points regarding the specifics of this IRV algorithm (how to resolve ties, when to declare a winner, etc.).

2 Tests 2–4

2.1 Test 2: When to declare a winner, and how to resolve ties

```
.data
votes: .word 4,4,1,2,3,4,1,2,3,4,2,1,3,4,2,1,3,4
results: .word 0,0,0,0,0,0
golden_results: 1,3,0,0,0,0
```

Declaring a winner: If one or more candidates secure 50% of the total votes cast at the end of the round, the first candidate to do so in order of the candidate listing on the ballots is declared the winner. In this test case, both candidates 1 and 2 secure 2 votes each at the end of the first round of IRV. Simply declare candidate 1 the winner since she/he has secured 50% (2-out-of-4) of the votes cast in the election.

Eliminating a candidate: On every round, the candidate who secures the least number of votes is eliminated. If multiple candidates are in contention for elimination (i.e., they are tied in bottom place at the end of a round), simply eliminate the first candidate tied for bottom place in order of the candidate listing on the ballots.[5] For this particular test case, candidates 3 and 4 both secure zero first place votes at the end of the first round, and candidate 3 is eliminated. This is a strange test case since the election terminates with ties all around, but it is there to capture these scenarios succinctly.

Deadlock: The above elimination strategy guarantees that no test case will result in deadlock. For example, in a simple scenario where two/three voters cast ballots for two/three candidates, respectively, and each candidate receives 1 top vote, the first candidate is the winner of round 1 while the second candidate is eliminated (in both scenarios). Draconian, sure, but it keeps the ball rolling and does away with the need to resolve these cases using other more complex rules in assembly code. Build two such test cases and use them as a reference from the word go ...

[5] By enforcing a listing rule that says candidate names appear on the ballot in the order of the time of official filing of their respective candidacies, it would make more sense to eliminate the first candidate tied for bottom place in reverse order of the candidate listing, i.e., the last candidate tied for bottom place in order of the candidate listing, but let's just keep things simple here.

2.2 Test 3: Redistribution test

```
.data
votes: .word 11,4,1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,
                  4,1,2,3,4,1,2,3,4,2,3,1,4,2,3,1,4,2,3,1,
                  4,2,3,1
results: .word 0,0,0,0,0,0
golden_results: .word 1,3,1,2,4,1
```

This test will stress your code in terms of how well you are redistributing the votes of eliminated candidates. In particular, at the end of each round, you have to redistribute the votes of the eliminated candidates. However, when you start this process, you may find that the new top choice of a voter may have already been eliminated. You have to ensure that in such cases, you setup your reassignment process such that you ensure that an uneliminated candidate emerges as the voter's top preference. This may be easy to track between successive rounds, but it is harder to track when the distance between rounds is greater than 1. And hence this test to help you ...

2.3 *Test 4: Combination of above scenarios with 5 candidates*

```
.data
votes: .word 11,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,2,1,3,4,5,2,1,5,4,3,
                  5,2,3,4,1,5,2,3,1,4,4,5,1,2,3,4,5,2,1,3,2,3,1,4,5,
                  4,5,1,3,2
results: .word 0,0,0,0,0,0,0,0
golden_results: .word 1,5,1,4,2,1,2,3
```

Adding more candidates and combining the above scenarios, though I must warn you that our final evaluation test cases (six more in number) will expand to up to 8 candidates and 64 voters. In other words, please test your solutions aggressively.

3 *Documentation*

You are expected to document your approach to solving the problem. Did you implement a high-level (C/C++/Java/Python/Go/New-fancy-language-on-the-block) solution? How did you develop your MIPS solution? How many nested loops? Complexity analysis? And Kartik's pet peeve/question: How did you test and validate your solution? If you choose to embed your documentation in text in your `main.asm` at the very bottom, fine. Else you may submit a separate document outlining your answers to these questions and add other information that you believe is pertinent to this project. We look for about 1-2 pages of text besides any supplementary material like tests, high-level code, etc.

4 *Collaboration policy*

To keep the process simple, I request that there be NO collaboration in any form on this project. If you have a question, please approach the TAs or the instructor by email (Cc all of us to keep things moving). Anyone in violation of this policy will be reported to the Dean's Office.

5 *Advice*

Implement a high-level solution, or at least some robust pseudo-code. Get a head start. If you try to finish this in the week of the deadline, it will more than likely not end well. Test well—no test is too simple or too silly to pass up, and no test can be that hard to craft either.

Please backup your work regularly, and please be prepared to share this upon request.

Finally, I believe that not every solution (including my own) is perfect. To this end, you (or we) may uncover a corner case or two in the next few weeks, and I will do my best to broadcast these by way of an explicit test. Further, not every submission will be perfect and I usually offer a regrade window of about a week. Submissions that failed one or more tests because they mishandled a corner case, or had some such relatively small error will be given an opportunity to earn credit back at our discretion. However, radical rewrites of code will not be eligible for such a regrade, so please make every effort to get to "code complete" by the deadline. Good luck!