This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 3

What to Do

1. Submit Plurality
2. Submit one of:
   - Runoff if feeling less comfortable
   - Tideman if feeling more comfortable

If you submit both Runoff and Tideman, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 3's sandboxes.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `plurality`, and

  ```
  make plurality
  ```

  is yielding errors, try running

  ```
  help50 make plurality
  ```

  instead!

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Plurality

Implement a program that runs a plurality election, per the below.

```
$ ./plurality Alice Bob Charlie
Number of voters: 4
Vote: Alice
Vote: Bob
Vote: Charlie
Vote: Alice
Alice
```

Background

Elections come in all shapes and sizes. In the UK, the Prime Minister (https://www.parliament.uk/education/about-your-parliament/general-elections/) is officially appointed by the monarch, who generally chooses the leader of the political party that wins the most seats in the House of Commons. The United States uses a multi-step Electoral College (https://www.archives.gov/federal-register/electoral-college/about.html) process where citizens vote on how each state should allocate Electors who then elect the President.

Perhaps the simplest way to hold an election, though, is via a method commonly known as the “plurality vote” (also known as “first-past-the-post” or “winner take all”). In the plurality vote, every voter gets to vote for one candidate. At the end of the election, whichever candidate has the greatest number of votes is declared the winner of the election.

Getting Started

Here’s how to download this problem’s “distribution code” (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you’re in `~/` (i.e., your home directory).
- Execute `mkdir pset3` to make (i.e., create) a directory called `pset3` in your home directory.
- Execute `cd pset3` to change into (i.e., open) that directory.
- Execute `mkdir plurality` to make (i.e., create) a directory called `plurality` in your `pset3` directory.
- Execute `cd plurality` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/3/plurality/plurality.c` to download this problem’s distribution code.
- Execute `ls` . You should see this problem’s distribution code, in a file called `plurality.c` .

Understanding

Let’s now take a look at `plurality.c` and read through the distribution code that’s been provided to you.

The line `#define MAX 9` is some syntax used here to mean that `MAX` is a constant (equal to `9`) that can be used throughout the program. Here, it represents the maximum number of candidates an election can have.

The file then defines a `struct` called a `candidate` . Each `candidate` has two fields: a `string` called `name` representing the candidate’s name, and an `int` called `votes` representing the number of votes the candidate has. Next, the file defines a global array of `candidates` , where each element is itself a `candidate` .

Now, take a look at the `main` function itself. See if you can find where the program sets a global variable `candidate_count` representing the

Now, take a look at the `main` function itself. See if you can find where the program sets a global variable `candidate_count` representing the number of candidates in the election, copies command-line arguments into the array `candidates`, and asks the user to type in the number of voters. Then, the program lets every voter type in a vote (see how?), calling the `vote` function on each candidate voted for. Finally, `main` makes a call to the `print_winner` function to print out the winner (or winners) of the election.

If you look further down in the file, though, you'll notice that the `vote` and `print_winner` functions have been left blank. This part is up to you to complete!

Specification

Complete the implementation of `plurality.c` in such a way that the program simulates a plurality vote election.

- Complete the `vote` function.
  - `vote` takes a single argument, a `string` called `name`, representing the name of the candidate who was voted for.
  - If `name` matches one of the names of the candidates in the election, then update that candidate's vote total to account for the new vote. The `vote` function in this case should return `true` to indicate a successful ballot.
  - If `name` does not match the name of any of the candidates in the election, no vote totals should change, and the `vote` function should return `false` to indicate an invalid ballot.
  - You may assume that no two candidates will have the same name.
- Complete the `print_winner` function.
  - The function should print out the name of the candidate who received the most votes in the election, and then print a newline.
  - It is possible that the election could end in a tie if multiple candidates each have the maximum number of votes. In that case, you should output the names of each of the winning candidates, each on a separate line.

You should not modify anything else in `plurality.c` other than the implementations of the `vote` and `print_winner` functions (and the inclusion of additional header files, if you'd like).

Usage

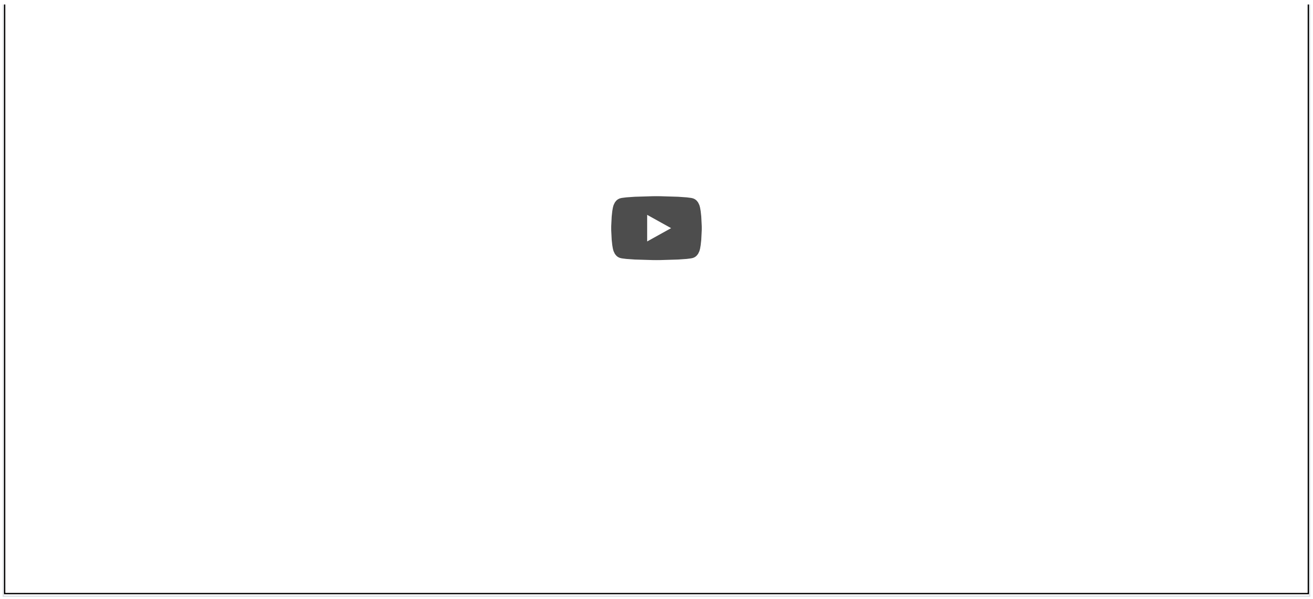
Your program should behave per the examples below.

```
$ ./plurality Alice Bob
Number of voters: 3
Vote: Alice
Vote: Bob
Vote: Alice
Alice
```

```
$ ./plurality Alice Bob
Number of voters: 3
Vote: Alice
Vote: Charlie
Invalid vote.
Vote: Alice
Alice
```

```
$ ./plurality Alice Bob Charlie
Number of voters: 5
Vote: Alice
Vote: Charlie
Vote: Bob
Vote: Bob
Vote: Alice
Alice
Bob
```

Walkthrough



Testing

Be sure to test your code to make sure it handles…

- An election with any number of candidate (up to the `MAX` of `9`)
- Voting for a candidate by name
- Invalid votes for candidates who are not on the ballot
- Printing the winner of the election if there is only one
- Printing the winner of the election if there are multiple winners

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/plurality
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 plurality.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/plurality
```

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Runoff

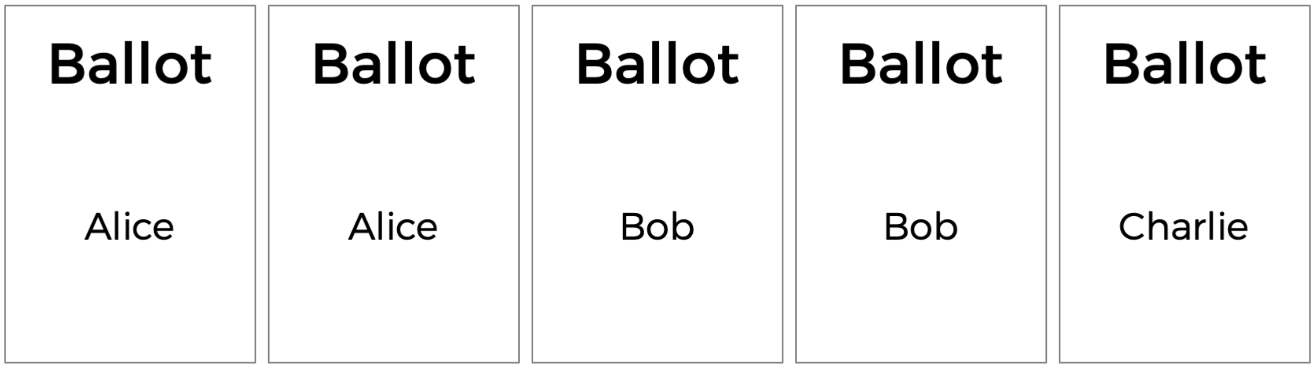
Implement a program that runs a runoff election, per the below.

```
./runoff Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Bob
Rank 3: Charlie

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Alice
Rank 3: Charlie

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Alice
```

Background

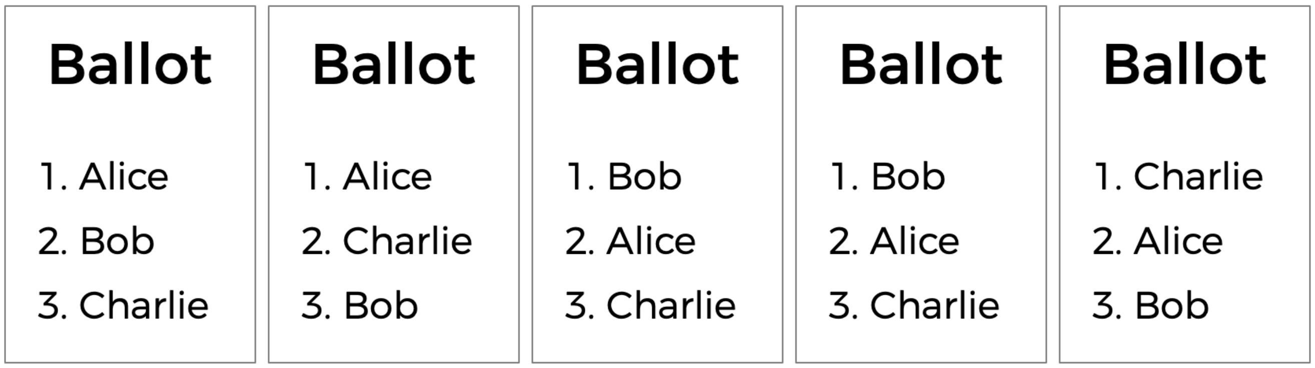
You already know about plurality elections, which follow a very simple algorithm for determining the winner of an election: every voter gets one vote, and the candidate with the most votes wins.

But the plurality vote does have some disadvantages. What happens, for instance, in an election with three candidates, and the ballots below are cast?



A plurality vote would here declare a tie between Alice and Bob, since each has two votes. But is that the right outcome?

There's another kind of voting system known as a ranked-choice voting system. In a ranked-choice system, voters can vote for more than one candidate. Instead of just voting for their top choice, they can rank the candidates in order of preference. The resulting ballots might therefore look like the below look like the below.



Ballot
1. Alice
2. Bob
3. Charlie

Ballot
1. Alice
2. Charlie
3. Bob

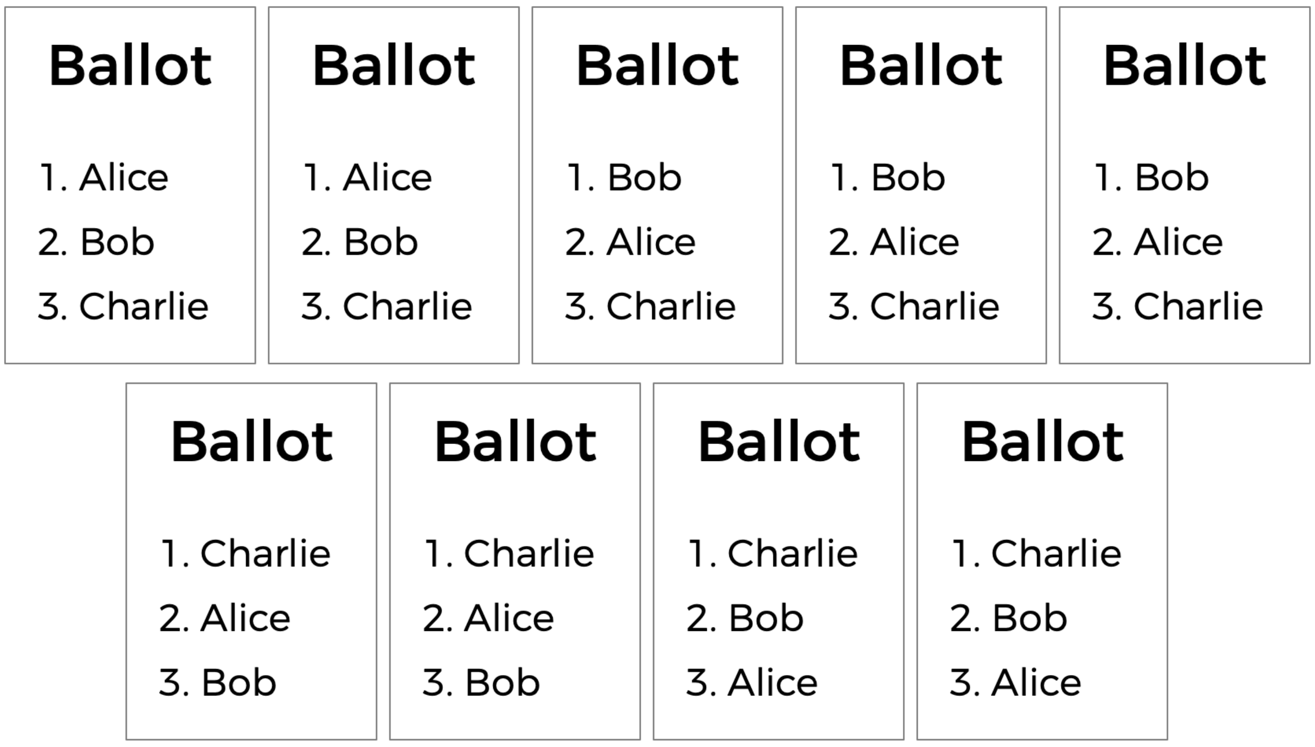
Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Charlie
2. Alice
3. Bob

Here, each voter, in addition to specifying their first preference candidate, has also indicated their second and third choices. And now, what was previously a tied election could now have a winner. The race was originally tied between Alice and Bob, so Charlie was out of the running. But the voter who chose Charlie preferred Alice over Bob, so Alice could here be declared the winner.

Ranked choice voting can also solve yet another potential drawback of plurality voting. Take a look at the following ballots.



Ballot
1. Alice
2. Bob
3. Charlie

Ballot
1. Alice
2. Bob
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Charlie
2. Alice
3. Bob

Ballot
1. Charlie
2. Alice
3. Bob

Ballot
1. Charlie
2. Bob
3. Alice

Ballot
1. Charlie
2. Bob
3. Alice

Who should win this election? In a plurality vote where each voter chooses their first preference only, Charlie wins this election with four votes compared to only three for Bob and two for Alice. But a majority of the voters (5 out of the 9) would be happier with either Alice or Bob instead of Charlie. By considering ranked preferences, a voting system may be able to choose a winner that better reflects the preferences of the voters.

One such ranked choice voting system is the instant runoff system. In an instant runoff election, voters can rank as many candidates as they wish. If any candidate has a majority (more than 50%) of the first preference votes, that candidate is declared the winner of the election.

If no candidate has more than 50% of the vote, then an “instant runoff” occurrs. The candidate who received the fewest number of votes is eliminated from the election, and anyone who originally chose that candidate as their first preference now has their second preference considered. Why do it this way? Effectively, this simulates what would have happened if the least popular candidate had not been in the election to begin with.

The process repeats: if no candidate has a majority of the votes, the last place candidate is eliminated, and anyone who voted for them will instead vote for their next preference (who hasn't themselves already been eliminated). Once a candidate has a majority, that candidate is declared the winner.

Let's consider the nine ballots above and examine how a runoff election would take place.

Alice has two votes, Bob has three votes, and Charlie has four votes. To win an election with nine people, a majority (five votes) is required. Since nobody has a majority, a runoff needs to be held. Alice has the fewest number of votes (with only two), so Alice is eliminated. The voters who originally voted for Alice listed Bob as second preference, so Bob gets the extra two vote. Bob now has five votes, and Charlie still has four votes. Bob now has a majority, and Bob is declared the winner.

What corner cases do we need to consider here?

One possibility is that there's a tie for who should get eliminated. We can handle that scenario by saying all candidates who are tied for last place will be eliminated. If every remaining candidate has the exact same number of votes, though, eliminating the tied last place candidates means eliminating everyone! So in that case, we'll have to be careful not to eliminate everyone, and just declare the election a tie between all remaining candidates.

Some instant runoff elections don't require voters to rank all of their preferences — so there might be five candidates in an election, but a voter might only choose two. For this problem's purposes, though, we'll ignore that particular corner case, and assume that all voters will rank all of the candidates in their preferred order.

Sounds a bit more complicated than a plurality vote, doesn't it? But it arguably has the benefit of being an election system where the winner of the election more accurately represents the preferences of the voters.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `cd pset3` to change into (i.e., open) your `pset3` directory that should already exist.
- Execute `mkdir runoff` to make (i.e., create) a directory called `runoff` in your `pset3` directory.
- Execute `cd runoff` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/3/runoff/runoff.c` to download this problem's distribution code.
- Execute `ls` . You should see this problem's distribution code, in a file called `runoff.c` .

Understanding

Let's open up `runoff.c` to take a look at what's already there. We're defining two constants: `MAX_CANDIDATES` for the maximum number of candidates in the election, and `MAX_VOTERS` for the maximum number of voters in the election.

Next up is a two-dimensional array `preferences` . The array `preferences[i]` will represent all of the preferences for voter number `i` , and the integer `preferences[i][j]` here will store the index of the candidate who is the `j` th preference for voter `i` .

Next up is a `struct` called `candidate` . Every `candidate` has a `string` field for their `name` , and `int` representing the number of `votes` they currently have, and a `bool` value called `eliminated` that indicates whether the candidate has been eliminated from the election. The array `candidates` will keep track of all of the candidates in the election.

The program also has two global variables: `voter_count` and `candidate_count` .

Now onto `main` . Notice that after determining the number of candidates and the number of voters, the main voting loop begins, giving every voter a chance to vote. As the voter enters their preferences, the `vote` function is called to keep track of all of the preferences. If at any point, the ballot is deemed to be invalid, the program exits.

Once all of the votes are in, another loop begins: this one's going to keep looping through the runoff process of checking for a winner and eliminating the last place candidate until there is a winner.

The first call here is to a function called `tabulate` , which should look at all of the voters' preferences and compute the current vote totals, by looking at each voter's top choice candidate who hasn't yet been eliminated. Next, the `print_winner` function should print out the winner if there is one; if there is, the program is over. But otherwise, the program needs to determine the fewest number of votes anyone still in the election received (via a call to `find_min`). If it turns out that everyone in the election is tied with the same number of votes (as determined by the `is_tie` function), the election is declared a tie; otherwise, the last-place candidate (or candidates) is eliminated from the election via a call to the `eliminate` function.

to the `eliminate` function.

If you look a bit further down in the file, you'll see that these functions — `vote` , `tabulate` , `print_winner` , `find_min` , `is_tie` , and `eliminate` — are all left up to you to complete!

Specification

Complete the implementation of `runoff.c` in such a way that it simulates a runoff election. You should complete the implementations of the `vote` , `tabulate` , `print_winner` , `find_min` , `is_tie` , and `eliminate` functions, and you should not modify anything else in `runoff.c` (except you may include additional header files, if you'd like).

`vote`

Complete the `vote` function.

- The function takes arguments `voter` , `rank` , and `name` . If `name` is a match for the name of a valid candidate, then you should update the global preferences array to indicate that the voter `voter` has that candidate as their `rank` preference (where `0` is the first preference, `1` is the second preference, etc.).
- If the preference is successfully recorded, the function should return `true` ; the function should return `false` otherwise (if, for instance, `name` is not the name of one of the candidates).
- You may assume that no two candidates will have the same name.

▶ **Hints**

`tabulate`

Complete the `tabulate` function.

- The function should update the number of `votes` each candidate has at this stage in the runoff.
- Recall that at each stage in the runoff, every voter effectively votes for their top-preferred candidate who has not already been eliminated.

▶ **Hints**

`print_winner`

Complete the `print_winner` function.

- If any candidate has more than half of the vote, their name should be printed to `stdout` and the function should return `true` .
- If nobody has won the election yet, the function should return `false` .

▶ **Hints**

`find_min`

Complete the `find_min` function.

- The function should return the minimum vote total for any candidate who is still in the election.

▶ **Hints**

`is_tie`

Complete the `is_tie` function.

- The function takes an argument `min` , which will be the minimum number of votes that anyone in the election currently has.
- The function should return `true` if every candidate remaining in the election has the same number of votes, and should return `false` otherwise.

▶ **Hints**

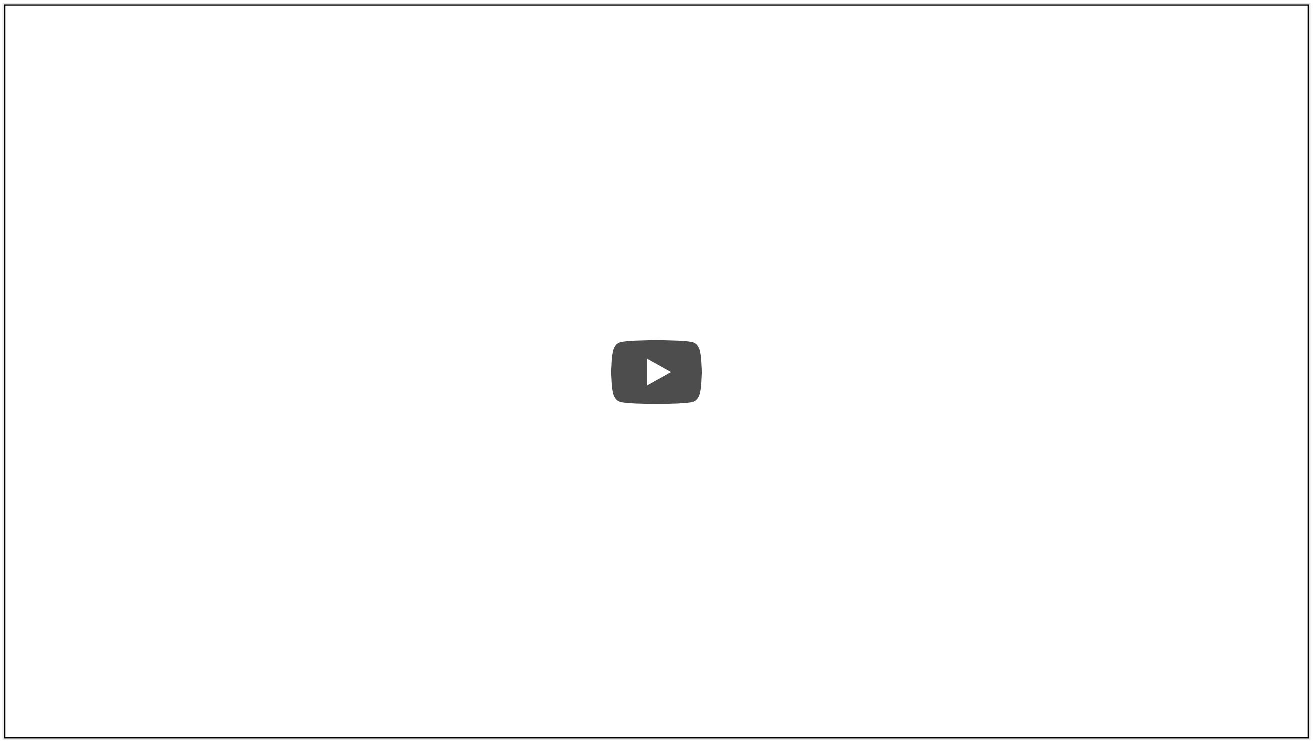
`eliminate`

Complete the `eliminate` function.

- The function takes an argument `min` , which will be the minimum number of votes that anyone in the election currently has.

- The function takes an argument `min`, which will be the minimum number of votes that anyone in the election currently has.
- The function should eliminate the candidate (or candidates) who have `min` number of votes.

Walkthrough



Usage

Your program should behave per the example below:

```
./runoff Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Alice
```

Testing

Be sure to test your code to make sure it handles…

- An election with any number of candidate (up to the `MAX` of `9`)
- Voting for a candidate by name
- Invalid votes for candidates who are not on the ballot
- Printing the winner of the election if there is only one

- Printing the winner of the election if there is only one
- Not eliminating anyone in the case of a tie between all remaining candidates

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/runoff
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 runoff.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/runoff
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Tideman

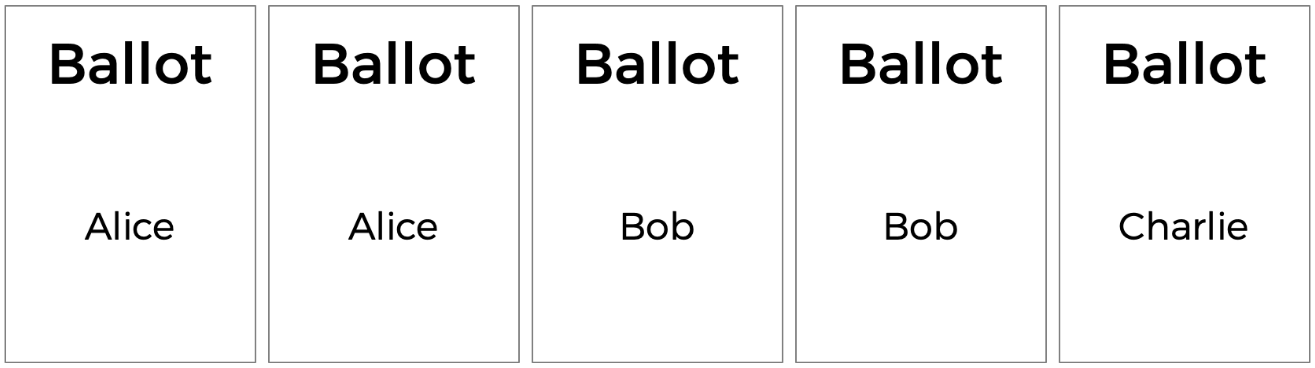
Implement a program that runs a Tideman election, per the below.

```
./tideman Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Charlie
```

Background

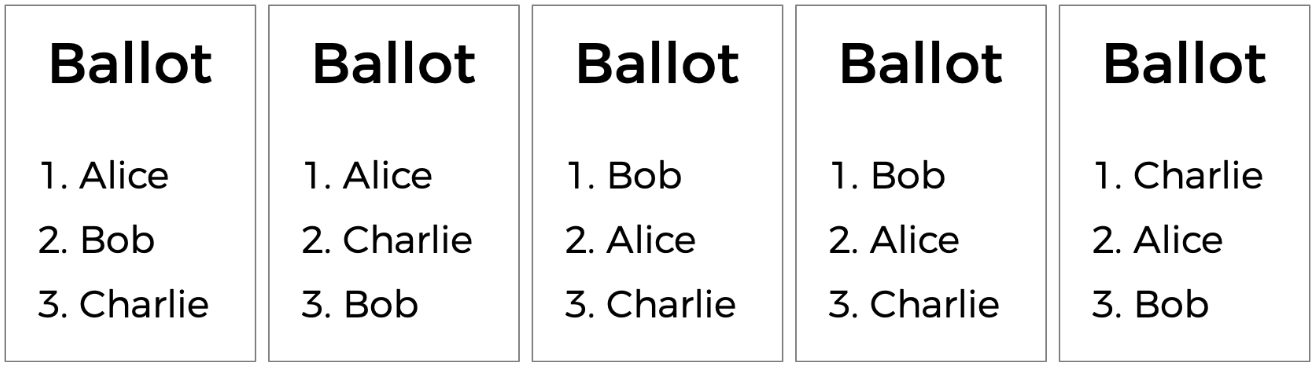
You already know about plurality elections, which follow a very simple algorithm for determining the winner of an election: every voter gets one vote, and the candidate with the most votes wins.

But the plurality vote does have some disadvantages. What happens, for instance, in an election with three candidates, and the ballots below are cast?



A plurality vote would here declare a tie between Alice and Bob, since each has two votes. But is that the right outcome?

There's another kind of voting system known as a ranked-choice voting system. In a ranked-choice system, voters can vote for more than one candidate. Instead of just voting for their top choice, they can rank the candidates in order of preference. The resulting ballots might therefore look like the below look like the below.



Ballot
1. Alice
2. Bob
3. Charlie

Ballot
1. Alice
2. Charlie
3. Bob

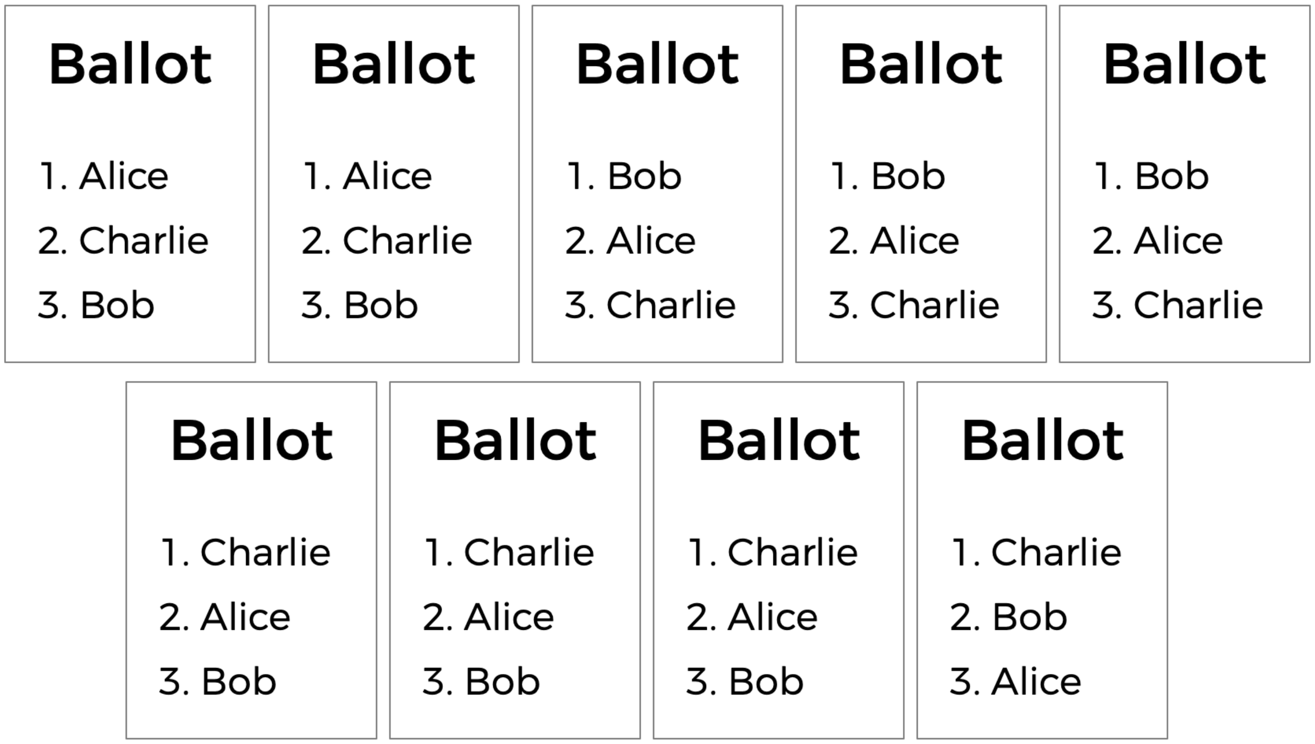
Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Charlie
2. Alice
3. Bob

Here, each voter, in addition to specifying their first preference candidate, has also indicated their second and third choices. And now, what was previously a tied election could now have a winner. The race was originally tied between Alice and Bob. But the voter who chose Charlie preferred Alice over Bob, so Alice could here be declared the winner.

Ranked choice voting can also solve yet another potential drawback of plurality voting. Take a look at the following ballots.



Ballot
1. Alice
2. Charlie
3. Bob

Ballot
1. Alice
2. Charlie
3. Bob

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Bob
2. Alice
3. Charlie

Ballot
1. Charlie
2. Alice
3. Bob

Ballot
1. Charlie
2. Alice
3. Bob

Ballot
1. Charlie
2. Alice
3. Bob

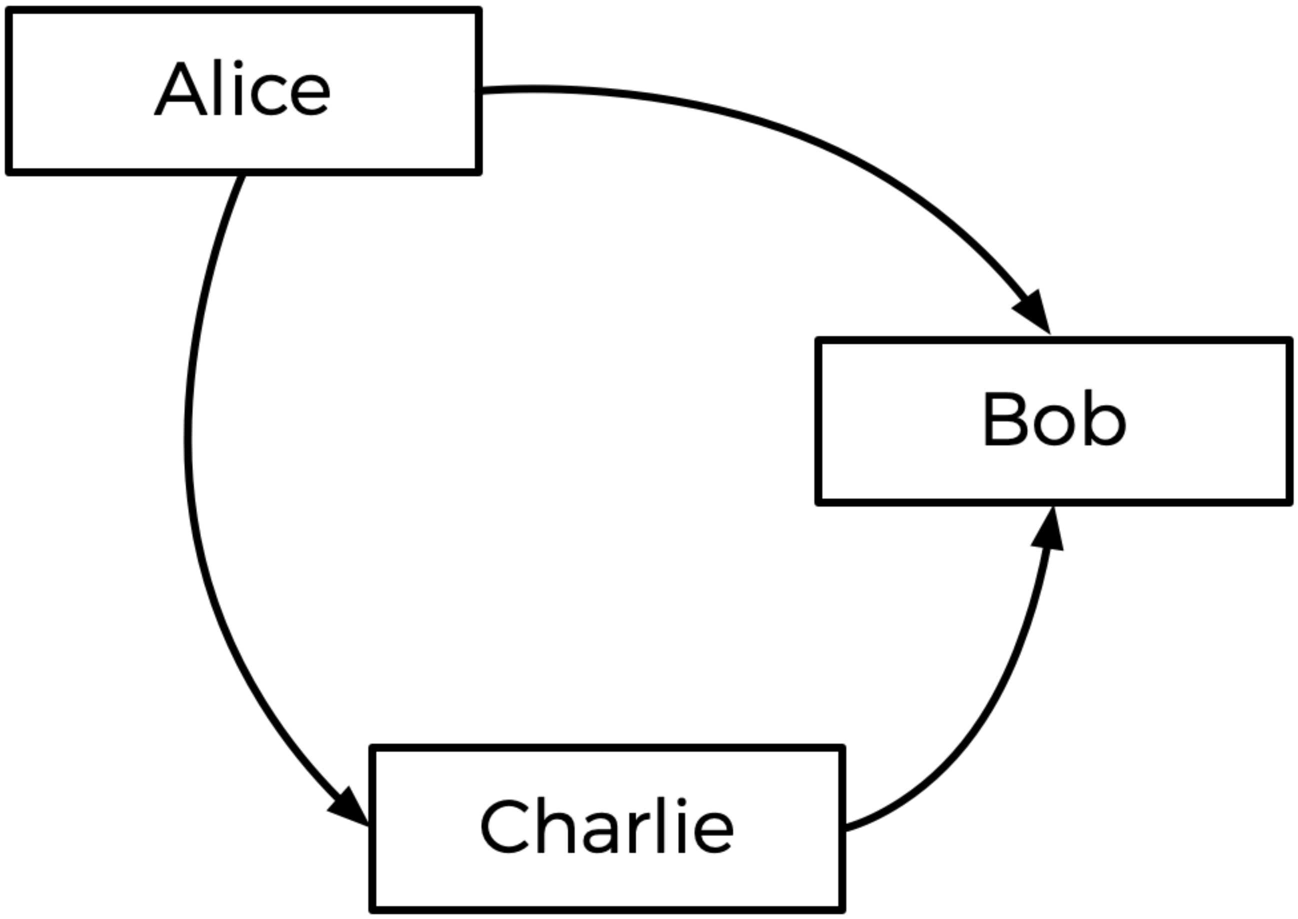
Ballot
1. Charlie
2. Bob
3. Alice

Who should win this election? In a plurality vote where each voter chooses their first preference only, Charlie wins this election with four votes compared to only three for Bob and two for Alice. (Note that, if you're familiar with the instant runoff voting system, Charlie wins here under that system as well). Alice, however, might reasonably make the argument that she should be the winner of the election instead of Charlie: after all, of the nine voters, a majority (five of them) preferred Alice over Charlie, so most people would be happier with Alice as the winner instead of Charlie.

Alice is, in this election, the so-called "Condorcet winner" of the election: the person who would have won any head-to-head matchup against another candidate. If the election had been just Alice and Bob, or just Alice and Charlie, Alice would have won.

The Tideman voting method (also known as "ranked pairs") is a ranked-choice voting method that's guaranteed to produce the Condorcet winner of the election if one exists.

Generally speaking, the Tideman method works by constructing a "graph" of candidates, where an arrow (i.e. edge) from candidate A to candidate B indicates that candidate A wins against candidate B in a head-to-head matchup. The graph for the above election, then, would look like the below.



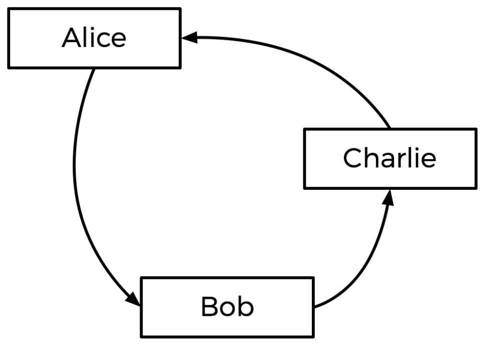
The arrow from Alice to Bob means that more voters prefer Alice to Bob (5 prefer Alice, 4 prefer Bob). Likewise, the other arrows mean that more voters prefer Alice to Charlie, and more voters prefer Charlie to Bob.

Looking at this graph, the Tideman method says the winner of the election should be the "source" of the graph (i.e. the candidate that has no arrow pointing at them). In this case, the source is Alice — Alice is the only one who has no arrow pointing at her, which means nobody is preferred head-to-head over Alice. Alice is thus declared the winner of the election.

It's possible, however, that when the arrows are drawn, there is no Condorcet winner. Consider the below ballots.

Ballot	**Ballot**	**Ballot**	**Ballot**	**Ballot**
1. Alice	1. Alice	1. Alice	1. Bob	1. Bob
2. Bob	2. Bob	2. Bob	2. Charlie	2. Charlie
3. Charlie	3. Charlie	3. Charlie	3. Alice	3. Alice

Ballot	**Ballot**	**Ballot**	**Ballot**
1. Charlie	1. Charlie	1. Charlie	1. Charlie
2. Alice	2. Alice	2. Alice	2. Alice
3. Bob	3. Bob	3. Bob	3. Bob



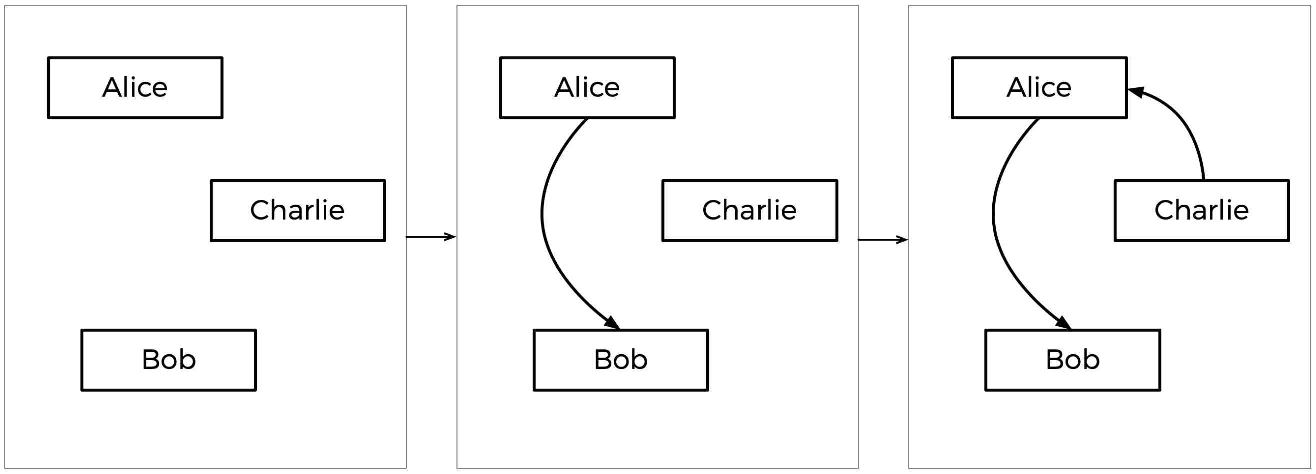
Between Alice and Bob, Alice is preferred over Bob by a 7-2 margin. Between Bob and Charlie, Bob is preferred over Charlie by a 5-4 margin. But between Charlie and Alice, Charlie is preferred over Alice by a 6-3 margin. If we draw out the graph, there is no source! We have a cycle of candidates, where Alice beats Bob who beats Charlie who beats Alice (much like a game of rock-paper-scissors). In this case, it looks like there's no way to pick a winner.

To handle this, the Tideman algorithm must be careful to avoid creating cycles in the candidate graph. How does it do this? The algorithm locks in the strongest edges first, since those are arguably the most significant. In particular, the Tideman algorithm specifies that matchup edges should be "locked in" to the graph one at a time, based on the "strength" of the victory (the more people who prefer a candidate over their opponent, the stronger the victory). So long as the edge can be locked into the graph without creating a cycle, the edge is added; otherwise, the edge is ignored.

How would this work in the case of the votes above? Well, the biggest margin of victory for a pair is Alice beating Bob, since 7 voters prefer Alice over Bob (no other head-to-head matchup has a winner preferred by more than 7 voters). So the Alice-Bob arrow is locked into the graph first. The next biggest margin of victory is Charlie's 6-3 victory over Alice, so that arrow is locked in next.

Next up is Bob's 5-4 victory over Charlie. But notice: if we were to add an arrow from Bob to Charlie now, we would create a cycle! Since the graph can't allow cycles, we should skip this edge, and not add it to the graph at all. If there were more arrows to consider, we would look to those next, but that was the last arrow, so the graph is complete.

This step-by-step process is shown below, with the final graph at right.



Based on the resulting graph, Charlie is the source (there's no arrow pointing towards Charlie), so Charlie is declared the winner of this election.

Put more formally, the Tideman voting method consists of three parts:

- **Tally**: Once all of the voters have indicated all of their preferences, determine, for each pair of candidates, who the preferred candidate is and by what margin they are preferred.
- **Sort**: Sort the pairs of candidates in decreasing order of strength of victory, where strength of victory is defined to be the number of voters who prefer the preferred candidate.
- **Lock**: Starting with the strongest pair, go through the pairs of candidates in order and "lock in" each pair to the candidate graph, so long as locking in that pair does not create a cycle in the graph.

Once the graph is complete, the source of the graph (the one with no edges pointing towards it) is the winner!

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `cd pset3` to change into (i.e., open) your `pset3` directory that should already exist.
- Execute `mkdir tideman` to make (i.e., create) a directory called `tideman` in your `pset3` directory.
- Execute `cd tideman` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/3/tideman/tideman.c` to download this problem's distribution code.
- Execute `ls`. You should see this problem's distribution code, in a file called `tideman.c`.

Understanding

Let's open up `tideman.c` to take a look at what's already there.

First, notice the two-dimensional array `preferences` . The integer `preferences[i][j]` will represent the number of voters who prefer candidate `i` over candidate `j` .

The file also defines another two-dimensional array, called `locked` , which will represent the candidate graph. `locked` is a boolean array, so `locked[i][j]` being `true` represents the existence of an edge pointing from candidate `i` to candidate `j` ; `false` means there is no edge. (If curious, this representation of a graph is known as an "adjacency matrix").

Next up is a `struct` called `pair` , used to represent a pair of candidates: each pair includes the `winner` 's candidate index and the `loser` 's candidate index.

The candidates themselves are stored in the array `candidates` , which is an array of `string` s representing the names of each of the candidates. There's also an array of `pairs` , which will represent all of the pairs of candidates (for which one is preferred over the other) in the election.

The program also has two global variables: `pair_count` and `candidate_count` , representing the number of pairs and number of candidates in the arrays `pairs` and `candidates` , respectively.

Now onto `main` . Notice that after determining the number of candidates, the program loops through the `locked` graph and initially sets all of the values to `false` , which means our initial graph will have no edges in it.

Next, the program loops over all of the voters and collects their preferences in an array called `ranks` (via a call to `vote`), where `ranks[i]` is the index of the candidate who is the `i` th preference for the voter. These ranks are passed into the `record_preference` function, whose job it is to take those ranks and update the global `preferences` variable.

Once all of the votes are in, the pairs of candidates are added to the `pairs` array via a called to `add_pairs` , sorted via a call to `sort_pairs` , and locked into the graph via a call to `lock_pairs` . Finally, `print_winner` is called to print out the name of the election's winner!

Further down in the file, you'll see that the functions `vote` , `record_preference` , `add_pairs` , `sort_pairs` , `lock_pairs` , and `print_winner` are left blank. That's up to you!

Specification

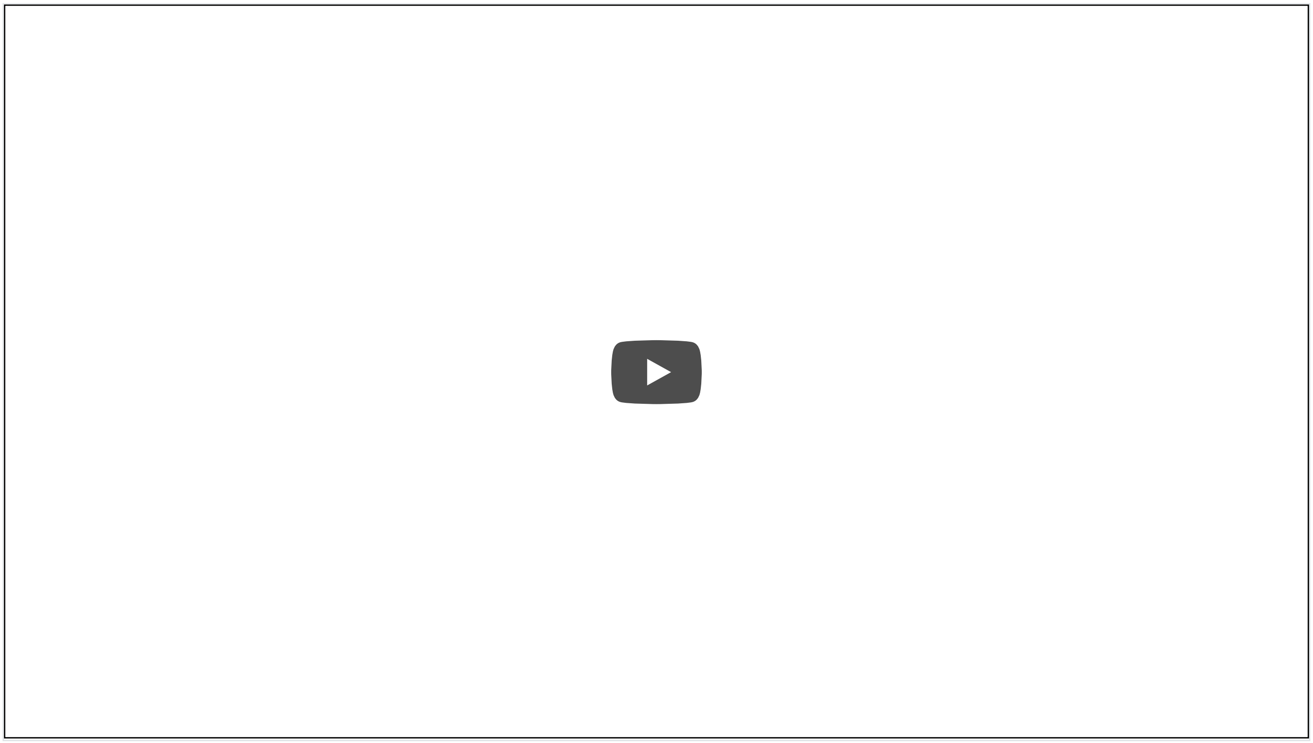
Complete the implementation of `tideman.c` in such a way that it simulates a Tideman election.

- Complete the `vote` function.
  - The function takes arguments `rank` , `name` , and `ranks` . If `name` is a match for the name of a valid candidate, then you should update the `ranks` array to indicate that the voter has the candidate as their `rank` preference (where `0` is the first preference, `1` is the second preference, etc.)
  - Recall that `ranks[i]` here represents the user's `i` th preference.
  - The function should return `true` if the rank was successfully recorded, and `false` otherwise (if, for instance, `name` is not the name of one of the candidates).
  - You may assume that no two candidates will have the same name.
- Complete the `record_preferences` function.
  - The function is called once for each voter, and takes as argument the `ranks` array, (recall that `ranks[i]` is the voter's `i` th preference, where `ranks[0]` is the first preference).
  - The function should update the global `preferences` array to add the current voter's preferences. Recall that `preferences[i][j]` should represent the number of voters who prefer candidate `i` over candidate `j` .
  - You may assume that every voter will rank each of the candidates.
- Complete the `add_pairs` function.
  - The function should add all pairs of candidates where one candidate is preferred to the `pairs` array. A pair of candidates who are tied (one is not preferred over the other) should not be added to the array.
  - The function should update the global variable `pair_count` to be the number of pairs of candidates. (The pairs should thus all be stored between `pairs[0]` and `pairs[pair_count - 1]` , inclusive).
- Complete the `sort_pairs` function.
  - The function should sort the `pairs` array in decreasing order of strength of victory, where strength of victory is defined to be the number of voters who prefer the preferred candidate. If multiple pairs have the same strength of victory, you may assume that the order does not matter.
- Complete the `lock_pairs` function.
  - The function should create the `locked` graph, adding all edges in decreasing order of victory strength so long as the edge would not create a cycle.

- Complete the `print_winner` function.
  - The function should print out the name of the candidate who is the source of the graph. You may assume there will not be more than one source.

You should not modify anything else in `tideman.c` other than the implementations of the `vote`, `record_preferences`, `add_pairs`, `sort_pairs`, `lock_pairs`, and `print_winner` functions (and the inclusion of additional header files, if you'd like). You are permitted to add additional functions to `tideman.c`, so long as you do not change the declarations of any of the existing functions.

Walkthrough



Usage

Your program should behave per the example below:

```
./tideman Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Charlie
```

Testing

Be sure to test your code to make sure it handles

Be sure to test your code to make sure it handles…

- An election with any number of candidate (up to the `MAX` of `9`)
- Voting for a candidate by name
- Invalid votes for candidates who are not on the ballot
- Printing the winner of the election

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/tideman
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 tideman.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/tideman
```