Students:
Section 14.4 is a part of 1 assignment:
Programming Project 04 - Win Presidency With Only 20% Support (Popular Vote Minimizer)

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Includes: zyLab

Due: 10/24/2024, 11:59 PM CDT

Instructor created

14.4 P04 - Win Presidency With Only 20% Support?

Popular Vote Minimizer

(Re)Cursing the Electoral College System for US Presidential Elections

The President of the United States is not elected by a popular vote, but by a majority vote in the Electoral College. Each of the 50 states, plus DC, gets some number of electors in the Electoral College, and whoever they vote in becomes the next President. There are four occurrences where the winner of the Electoral College and eventual President Elect did not win the popular vote (plus the 1824 election where the House of Representatives decided the winner). For the purposes of this project, we're going to make some simplifying assumptions:

- You need to win a majority of the votes in a state to earn its electors, and you get all the state's electors if you win the majority of the votes. For example, in a small state with 999,999 people, you'd need 500,000 votes to win all its electors. Similarly, in a state with 1,000,000 people, you'd need 500,001 votes to win all its electors. These assumptions aren't entirely accurate, both because in most states a plurality suffices and some states split their electoral votes in other ways.
- You need to win a majority of the electoral votes to become president. In the 2008 election, you'd need 270 votes because there were 538 electors. In the 1804 election, you'd need 89 votes because there were only 176 electors. (You can technically win the presidency without winning the Electoral College; we'll ignore this for simplicity.)
- Electors never defect. The electors in the Electoral College are free to vote for

whomever they please, but the expectation is that they'll vote for the candidate that won their home state. As a simplifying assumption, we'll just pretend electors always vote with the majority of their state.

This project explores the following *central question*: under the assumptions laid out above, for past elections ...

What was the fewest number of popular votes you could get and still be elected President of the United States of America?

In order to answer this *central question*, the programming tasks involve command-line arguments, writing a useful makefile, file input, writing a recursive function, developing your own test cases, optimizing recursion using memoization, and file output.

Starter Code & Provided Election Data Files

The workhorse of this program is the function

```
MinInfo minPopVoteToWin(State* states, int szStates)
```

together with its recursive helper function

```
MinInfo minPopVoteAtLeast(State*
states, int szStates, int start, int EVs)
```

that takes as input a list of all the states that participated in the election (plus DC, if applicable), then returns some information about the minimum number of popular votes needed to win the election (namely, how many votes you'd need, and a list of the states you would carry in the process).

Here's a quick overview of the struct types involved here. First, there's the **State** type, defined in the **MinPopVote.h** header file as

```
typedef struct State_struct {
    char name[50];      // the name of the state, e.g. Illinois
    char postalCode[3]; // the postal code of state, e.g. IL
    int electoralVotes; // how many electors the state has
    int popularVotes;   // number of people who voted
} State;
```

The input to **minPopVoteToWin()** is an array of **State**s (and its size **szStates**) containing information about all the states that participated in the election. The **minPopVoteToWin()** function then returns a **MinInfo**, a struct type also defined in the **MinPopVote.h** header file that contains information about the minimum popular votes needed to win the election and

an array of the **State** structs that would need to be carried in order to win:

```
typedef struct MinInfo_struct {
    State someStates[51]; // a subset of states
    int szSomeStates;     // number of states in this subset
    int subsetPVs;     // number of popular votes for subset
    bool sufficientEVs; // true = subset has enough electoral votes to win
} MinInfo;
```

Note that there is a Boolean subitem for **MinInfo**, namely **sufficientEVs**, which flips from **false** to **true** if the accumulated number of electoral votes for this particular combination of states is sufficient to win the election (this is useful during the recursion steps).

The starter code provided in the IDE below includes the following files and folders:

- **MinPopVote.h** - header file for the Popular Vote Minimizer library; nothing needs to be done to this file.
- **MinPopVote.c** - implementation file for the Popular Vote Minimizer library; this includes ALL of the functions developed in the tasks below.
- **app.c** - the main Popular Vote Minimizer application; this is a read-only file so nothing needs to be done to this file; however, it is crucial that you read through the code and read all comments to understand how the main driver works for this program
- **test.c** - the student testing suite for the Popular Vote Minimizer library; develop your own test cases for the functions you write in MinPopVote.c here.
- **makefile** - a partially developed makefile; write additional targets for various command-line argument cases and for the student testing suite.
- **demo.exe** - a sample executable for the primary application. You can run this to check the expected output for app.c with a fully functioning MinPopVote implementation. Make sure to change the permissions to allow execution first. For example:
    - **-> chmod a+x demo.exe**
    - **-> ./demo.exe -f -y 2020**
- **data/** - a folder containing US Presidential election data files, which are titled [*year*].csv, where [*year*] is a year between 1828 and 2020, inclusively, that is a perfect multiple of 4 (since presidential elections occur in 4 year intervals).
    - These files have **c**omma-**s**eparated-**v**alue format, i.e. .csv, which means that information is separated by comma.
    - Each line in the data files contains the state results of the election for that year, with the following structure: [*stateName*],[*postalCode*],[*electoralVotes*],[*popularVotes*]
    - As an example, the first line of 1828.csv is: **Alabama,AL,5,18618** (which means that in the 1828 election, 18,618 votes were cast in Alabama, which has AL for a postal code, and the winner received 5 electoral votes in the Electoral College.)
- **toWin/** - an empty folder where output files for minimum popular vote winning subsets

of states will be saved.

Programming Tasks

Tasks 1-6 are focused on setting up the application program, specifically some basic functions in the **MinPopVote** library, application settings using command-line arguments, a makefile to handle the various settings and a testing suite, and reading the election data into the program to set up the array of **State**s. Tasks 1-6 are a warm-up for Tasks 7-8, which involve a tricky recursive function that is later optimized with memoization. Task 9 is all about properly formatted output written to a file. Finally, Task 10 is a reflection on the output results of the program in answering the *central question*.

1. totalEVs(), totalPVs(), & test.c - write two basic functions & write your own test case(s)

Find the following two function prototypes in the **MinPopVote.c** library implementation file:

```
int totalEVs(State* states, int szStates)
int totalPVs(State* states, int szStates)
```

A **State** struct array **states** of size **szStates** is input to the functions **totalEVs()** and **totalPVs()**, which should calculate and return the total number of electoral votes and popular votes, respectively, in the **states** array. Do not overthink this task. The two functions are VERY similar.

Once you have the function(s) written and are ready to test them, open the **test.c** file, which is where you will develop your own test cases for each function you write in **MinPopVote.c.** A sample test is provided for **totalEVs()**. Note how a small toy States array is built, in such a manner that predicting the expected return value is straightforward. Thus, in order to test the function, we can compare the expected return to the actual return.

Now, write a similar test function, called **test_totalPVs()**, that tests if **totalPVs()** is functioning correctly. The unit test case function is already called from **main()** in the **test.c** file. At this point, it may be helpful to extend the provided **makefile** for the targets **build_test** and **run_test**, which are fully detailed in Task 4. Once these new targets are added to the makefile, the testing suite can be built and run simply using following terminal commands:

```
-> make build_test
-> make run_test
```

As you move on to the following tasks, continue writing, building, and running test cases for the functions you write, which will include **setSettings()**, **inFilename()**, **outFilename()**, **parseLine()**, **readElectionData()**, **minPopVoteAtLeast()**, and **minPopVoteAtLeastFast()**. Your test case functions should check all components of each function.

Continually run **make build_test** and **make run_test** from the console to test the functions as you develop them. Always do this before relying on the autograded test cases.

The testing suite portion of the project is purposely open-ended. In the end, your test case functions will be manually graded for thoroughness and rigor. To help get an idea of the coverage of your test cases, the **Gradescope submission** has a built-in autograder that provides some feedback on test case coverage. Read the details in the **Submission & Grading** section below.

2. setSettings() - settings based on command-line arguments and/or interactive user input

The **app.c** application program should handle the following command-line arguments in ANY order:

```
// command-line argument settings
// [-y yr] = sets the election year for the program
//           valid [yr] values are perfect multiples of 4,
//           between 1828 and 2020, inclusively;
//           if yr is not an election year, then set [year] to 0;
//           default is 0 ([year] then set by user-input later)
// [-q] = quiet mode; if ON, do not print the full State list read-in
//                    from file AND do not print the subset of States
//                    needed to win with minimum popular votes;
//                    default is OFF
// [-f] = fast mode; if ON, use the "fast" version of the functions
//                   that include memoization to find the minimum
//                   number of popular votes to win the election;
//                   default is OFF
// these arguments are optional to run the program;
// if any argument is absent, then use the default value
```

The command-line arguments are processed in the **setSettings()** function in **MinPopVote.c:**

```
bool setSettings(int argc, char** argv, int* year, bool* fastMode, bool* quietMo
```

- The input parameters **argc** and **argv** are identical to the traditional command-line

inputs to main().

- The passed-by-pointer parameter ***fastMode** should be set to **true** if **-f** exists anywhere in the list of command-line flags (except if immediately following **-y**). Otherwise, set ***fastMode** to its default value of **false**.
- The passed-by-pointer parameter ***quietMode** should be set to **true** if **-q** exists anywhere in the list of command-line flags (except if immediately following **-y**). Otherwise, set ***quietMode** to its default value of **false**.
- The passed-by-pointer parameter ***year** should be set to the command-line argument **[yr]** that immediately follows **-y**, only if it is included anywhere in the list of command-line flags, otherwise set it to 0. If **[yr]** is invalid for ANY reason (this includes another valid command-line argument, such as **-f** or **-q**; in which case DO NOT interpret the **-f** or **-q** as an independent command-line argument, but instead the **-f** or **-q** is an attempted year that does not exist), then set ***year** to the default value 0. Note that if ***year** is set to 0 here, then interactive user-input is used in **main()** to set the election year. Examples:
    - if the command-line arguments include **-y 1992**, then year should be set to 1992
    - if the command-line arguments include **-y 1800**, then year should be set to 0 (no data file for the 1800 election)
    - if the command-line arguments include **-y 2023**, then year should be set to 0 (no election in 2023)
    - if the command-line arguments include **-y -f**, then year should be set to 0 (**-f** is NOT an election year); also the **-f** argument should NOT be used to set fast mode ON, but instead the **-f** is simply an attempted year that did not have an election.
    - if the command-line arguments do NOT include **-y**, OR there is no argument after **-y**, then year should be set to 0
- The function should return **true** if all command-line arguments are valid; otherwise, return **false**. Examples:
    - if the command-line arguments are **./app.exe -f -q -y 2020**, then return **true** (all valid)
    - if the command-line arguments are **./app.exe -f -v -y 2020**, then return **false** (**-v** is invalid)
    - if the command-line arguments are **./app.exe -f -q -y**, then return **false** (**-y** has no argument after it for the year)
    - if the command-line arguments are **./app.exe -y 55 -q**, then return **true** (**-y** and **-q** are valid, and year will be set by interactive user-input since **55** is an invalid year)
    - if the command-line arguments are **./app.exe -y -f -q**, then return **true** (**-y** and **-q** are valid, and year will be set by interactive user-input since **-f** is not an election year; note that **-f** is NOT interpreted as fast mode here, just a year

without an election, so quiet mode will be ON but fast mode will be OFF)

Continue developing your testing suite as you develop code for this task. That is, write **test_setSettings()** in **test.c** to fully test the functionality of **setSettings()**. Your test case functions should check all components of each function.

3. inFilename() & outFilename() - generate C-strings for the relative paths to the input and output data files

Once the variable **year** is set in **main()** of **app.c**, either by the command-line arguments in **setSettings()** or interactive user-input in **main()**, the election data file can be located in the **data/** folder, such that the full path and file name for the input file follows the pattern **data/[year].csv**, where **[year]** is replaced by a 4-digit year, e.g. 1828, 1984, 2020, etc. Write the **inFilename()** function in MinPopVote.c:

```
void inFilename(char* filename, int year)
```

The C-string **filename** should be updated with the correct path and name for the input file using the input parameter **year**.

Similarly, the output file that will be written at the end of main() should be stored in the **toWin/** folder, such that the full path and file name for the output file follows the pattern **toWin/[year]_win.csv**, where **[year]** is replaced by a 4-digit year, e.g. 1828, 1984, 2020, etc. Write the **outFilename()** function in MinPopVote.c:

```
void outFilename(char* filename, int year)
```

The C-string **filename** should be updated with the correct path and name for the output file using the input parameter **year**.

Once the command-line arguments are handled and the filenames are generated, **app.c** prints the program settings using the following format:

```
Settings:
  year = 2020
  quiet mode = ON
  fast mode = ON
  input data file = data/2020.csv
  output data file = toWin/2020_win.csv
```

Continue developing your testing suite as you develop code for this task. That is, write **test_inFilename()** and **test_outFilename()** in **test.c** to fully test the functionality of **inFilename()** and **outFilename()**. Your test case functions should check all components of each function.

4. makefile - extend the makefile

The provided **makefile** already has many targets that allows you to run the following commands from the command-line:

- **make build** compiles **app.c** with the functions in **MinPopVote.c** and builds the executable **app.exe**.
- **make run** executes the program **app.exe** using default values for command-line arguments.
- **make run_quiet** executes the program **app.exe** with quiet mode ON and default values for all other command-line arguments.
- **make valgrind** executes the program **app.exe** in fast mode and using default values for other command-line arguments under valgrind.

Extend the makefile for the following targets:

- **run_fast** to execute the program **app.exe** with fast mode ON and default values for all other command-line arguments.
- at least 2 additional run targets for **app.exe**, similar to **run_quiet** but with other meaningful combinations of program settings set by command-line arguments
- **build_test** to compile **test.c** with the functions in **MinPopVote.c** and build the executable **test.exe**
- **run_test** to exectue the testing suite **test.exe**
- any additional targets you find useful

5. parseLine() - parse a single line of data from the election data file

The implementation file **MinPopVote.c** contains the function header

```
bool parseLine(char* line, State* myState)
```

that you are now tasked with writing. If the input string **line** has the correct format of

```
[stateName],[postalCode],[electoralVotes],[popularVotes]
```

then use it to properly set all of the subitems for the passed-by-pointer **State** struct, namely ***myState**, AND return **true**. Otherwise, if the format of the line is not valid (e.g. there are only 2 commas), then return **false**. Note that the **parseLine()** function should accept anything for the four parsed items and should NOT check if each individual item is valid; e.g. the function should accept ANYTHING for the [postalCode], when in actuality, the [postalCode] must be a two-char string with ONLY upper-case letters that represents a real state or DC; instead, just check if the input string contains four items (which could be empty items) separated by commas; return **true** if there are four items, and return **false** if there are more or less than four items. The check for the actual validity of the individual

items would need to be done outside of parseLine after the State struct has been assigned to whatever the input line contains.

Note: your function should handle the input C-string **line** ending with a newline character **'\n'** (e.g. if **fgets()** is used to read a line) or not. For example, both of the following C-strings are valid inputs to the function:

```
"Illinois,IL,20,6033744" and "Illinois,IL,20,6033744\n"
```

Continue developing your testing suite as you develop code for this task. That is, write **test_parseLine()** in **test.c** to fully test the functionality of **parseLine()**. Your test case functions should check all components of each function; e.g. **parseLine()** returns the validity status for the formatting of the input line AND (if the format is valid) sets proper values for **myState;** so, your test case function should test both functionality components.

6. readElectionData() - reading election data from file

The implementation file **MinPopVote.c** contains the function header

```
bool readElectionData(char* filename, State* allStates, int* nStates)
```

that you are now tasked with writing.

Open the election data file **filename** for reading, and read in the data one line at a time. Immediately return **false** if **filename** cannot be found. The string library function **fgets()** can be used to read in an entire line up to ***AND INCLUDING*** the newline character, **'\n'**. Then, use repeated calls to the **parseLine()** function that was developed in the previous task to fill the struct array **allStates**. Since we are dealing with U.S. presidential election data, the number of "states" is at most 51 (i.e. the 50 states plus DC). However, many of the elections involved fewer states than we have now, simply because many modern states did not exist yet. Thus, the array is declared in **main()** of **app.c** as a static array that allows a maximum possible size of 51, but you may not fill all the elements if it is an earlier election year. Thus, ***nStates** is a passed-by-pointer parameter that represents the total number of states. *Make sure to initialize ***nStates** to 0 inside this function, before incrementing. You cannot assume it is set to zero outside of the function.*The string function **feof()** can be used to determine if the end-of-file has been reached. Return **true** if the file reading process was successful.

Once the **State** array is built, and ***only if quiet mode is turned OFF***, **main()** displays the full list of state election data in the following format (this sample output is the first few states for year = 1828):

```
Electoral Data for Alabama (AL):
  Electoral Votes = 5
```

```
  Popular Votes = 18618
Electoral Data for Connecticut (CT):
  Electoral Votes = 8
  Popular Votes = 19378
Electoral Data for Delaware (DE):
  Electoral Votes = 3
  Popular Votes = 13944
Electoral Data for Georgia (GA):
  Electoral Votes = 9
  Popular Votes = 20004
Electoral Data for Illinois (IL):
  Electoral Votes = 3
  Popular Votes = 14222
Electoral Data for Indiana (IN):
  Electoral Votes = 5
  Popular Votes = 39210
...
  ...
  ...
```

Continue developing your testing suite as you develop code for this task. That is, write **test_readElectionData()** in **test.c** to fully test the functionality of **readElectionData()**. Your test case functions should check all components of each function.

7. minPopVoteAtLeast() - slow, brute-force recursion to find minimum-PV, sufficient-EV subset of states

We are now ready to tackle the *central question* head on and implement the brute-force (i.e. *slow*) version of the *Minimize the Popular Vote* algorithm, which involves considering every possible combination of **State**s in the array, checking if the total electoral votes for that subset is enough to win the election, and calculating the total popular votes for the subset. If we do that for ALL possible subsets, the subset we seek is the one with the least number of popular votes but enough electoral votes to win the election.

The function **minPopVoteToWin()** in **MinPopVote.c** should return the **MinInfo** associated with the subset of **states** that has the minimum popular vote total while still having enough electoral votes to win the election. This is VERY similar to the **"Backpack Problem"** that is solved in lab. Refer to the lab exercise for details on the algorithm. However, there are some key differences between the *Backpack Problem* and the *Central Question* here:

- The ***Backpack Problem*** seeks to **maximize** the **value** with an **upper** bound limitation on total **weight**
- The ***Central Question*** seeks to **minimize** total **PVs** with a **lower** bound limitation on total **EVs**

Thus, the algorithm is essentially the same, but with opposite extrema for the value sought and opposite bounding directionality for the constraint.

Note that **minPopVoteToWin()** is a wrapper function that is fully provided in the starter code. All it does is calculate the required number of electoral votes to win the election (with a call to **totalEVs()**) and then call the recursive function **minPopVoteAtLeast()**, which is the function you need to write that implements the *minimize-PVs* algorithm:

```
MinInfo minPopVoteAtLeast(State* states, int szStates, int start, int EVs)
```

This function actually solves a more general version of the *central question* since it returns the minimum popular vote total for State subsets, but only considering the States from index [**start**] to the end of the array. The input State array [**states**] of size [**szStates**] remains unchanged for the entire recursion process, i.e. all calls to **minPopVoteAtLeast()** use the same array [**states**] of size [**szStates**]. The final parameter [**EVs**] varies based on how much additional electoral votes are needed to win the election.

In summary, the recursive approach builds off of generating the power set by considering a single State element in the array and separating subsets based on the inclusion or exclusion of the current State. Then, we recursively do the same thing with the next State element in the array for each subset. In doing so, we generate 2^n subsets, where n is the size of the State array. As subsets are generated, we keep track of the required additional electoral votes needed to win the election. So, the recursive call for the subset that includes the current State should have a reduced [**EVs**] argument, while the recursive call for the subset that excludes the current State should have the same [**EVs**] argument. There are two base cases for exiting recursion: (1) the subset is complete when [**start**] equals [**szStates**], and (2) no additional states should be added to the subset if the required additional electoral votes to win has been met, i.e. [**EVs**] is negative. The total popular votes for each subset and the State subsets themselves are then accumulated as we climb out of the recursion levels. Each recursive step should compare the two subsets: one includes the current State and the other excludes the current State. Return the subset that excludes the current State ONLY IF it achieves a winning electoral vote total AND it has a lower popular vote total compared to the subset that includes the current State. The **MinInfo** struct contains **bool sufficientEVs** that is helpful for making this decision. Otherwise, return the subset that includes the current State.

Note that both minPopVoteAtLeast() and **minPopVoteToWin()** return a **MinInfo** struct that contains not only the total number of popular votes associated with this subset of

states, but it also includes the subset of states itself as a **State** array subitem. You are strongly encouraged to just focus on getting the minimum popular vote working first before attempting to collect the subset of **State**s in the **someStates[]** subitem.

IMPORTANT: Due to the recursive nature of this algorithm, the **State**s that are in the minimum-PV subset are stored in reverse order compared with the full state list; i.e. the full state list is in alphabetical order to match the input data files, BUT the subset is in reverse alphabetical order.

Write the test case function in **test.c** called **test_minPVsSlow()**, that tests if minPopVoteToWin() is functioning correctly. For example, build a small array of States (four States is probably sufficient) where you can easily predict the MinInfo values for the minimum popular vote subset. Check all subitems of the returned MinInfo in your test function. This test function is called from **main()** in **test.c**. Finally, run **make build_test** and **make run_test** from the console to test the functions you wrote. Always do this before relying on the autograded test cases.

The main application in **app.c** includes a call to minPopVoteToWin() if fast mode is OFF. Since this is the *slow* brute-force version, you should expect it to only work for relatively small State arrays. It just so happens that earlier elections had less states. So, when running the application, it is best to stay in the 1800s when in slow mode. The program will likely not run to completion if you try election year 2020 in *slow* mode.

8. minPopVoteAtLeastFast() - fast, optimized recursive helper function using memoization

In order for our program to handle the full 51 election states (50 actual states plus DC), we need to optimize the minPopVoteAtLeast() recursive function. To do so, we apply memoization. This, again, is VERY similar to the approach taken to optimize our solution for the **"Backpack Problem"**. Once again, refer the the lab description for full details on the algorithm optimization technique.

Copy your fully-functioning and fully-tested implementation of minPopVoteAtLeast() and paste it into minPopVoteAtLeastFast():

```
MinInfo minPopVoteAtLeastFast(State* states, int szStates, int start, int EVs, M
```

Because this is a recursive function, make sure to change any minPopVoteAtLeast() calls inside of the pasted code to minPopVoteAtLeastFast(). This "fast" version of the recursive helper function has an added parameter, namely the double **MinInfo** pointer [**memo**]. The new wrapper function minPopVoteToWinFast() allocates memory for the two-dimensional array **memo** with a size of **[number of States + 1] x [required electoral votes to win + 1]** and initializes all **subsetPVs** subitem values to -1.

Make the required memoization additions to your unoptimized code in minPopVoteAtLeastFast():

- After checking for base cases (memoization has no effect on the base cases), check if this particular **[start][EVs]** parameter pair has already been calculated. If the **subsetPVs** subitem value of that **memo** element is no longer -1, then it has already been calculated and you should immediately return that **MinInfo** (no need to continue with recursion). Be careful to first check for the base cases, particularly if **EVs** is negative, as you do not want to access any **memo** elements using negative indices.
- Change any recursive **minPopVoteAtLeast()** calls to **minPopVoteAtLeastFast(),** and add **memo** as the final argument.
- Right before any non-base case return statement, save the **MinInfo** calculated for the specific **[start][EVs]** parameter pair to the **memo** array. That way, if your program ever needs it again during recursion, it will just use this saved **MinInfo** instead of continuing with recursion.

Make sure to free up all of the heap-allocated memory for **memo** at the end of the minPopVoteToWinFast() function.

Lastly, write the test case function in **test.c** called **test_minPVsFast()**, that tests if minPopVoteToWinFast() is functioning correctly. You can reuse the setup of the **test_minPVsSlow()** here, Additionally, in order to really test the optimized algorithm, your test case must also involve a large number (~50) of States. This test function is called from **main()** in **test.c**. Finally, run **make build_test** and **make run_test** from the console to test the functions you wrote. Always do this before relying on the autograded test cases.

You should now compile (**make build**) and run (**make run** OR **make run_fast**) your application program. Check that *slow* mode and *fast* mode produce the same results for the early election years, such as 1828. Then, do the same for a more recent year, say 2020. The program will not run to completion in *slow* mode, but should finish in a flash in *fast* mode.

9. writeSubsetData() - output printed summary and winning strategy data to a file

After the minimum-PV winning subset of States has been determined, the main application will display the State subset in a condensed format:

```
States in the set:
  Alabama (AL): 9 EVs, 1161642 PVs
  Alaska (AK): 3 EVs, 179766 PVs
  Arizona (AZ): 11 EVs, 1693664 PVs
  Arkansas (AR): 6 EVs, 609535 PVs
```

```
        ...
        ...
        ...
```

Note that the PVs shown here are the **minimum number of popular votes in each state to win that state's electoral votes**. Then, a statistical summary is displayed:

```
Statistical Summary:
  Total EVs = 538
  Required EVs = 270
  EVs won = 270
  Total PVs = 158376434
  PVs Won = 34142388
  Minimum Percentage of Popular Vote to Win Election = 21.56%
```

This information is then written to an output file (**toWin/[year]_win.csv**), where the first line format is:

```
[TotalEVs],[TotalPVs],[EVsWon],[PVsWon]
```

After the first line, the individual State details, for the subset of states, are saved, one State per line:

```
[stateName],[postalCode],[electoralVotes],
[popularVotesToWinState]
```

For example, the file **toWin/2020_win.csv** should contain:

```
538,158376434,270,34142388
Alabama,AL,9,1161642
Alaska,AK,3,179766
Arizona,AZ,11,1693664
Arkansas,AR,6,609535
      ...
      ...
      ...
```

The file-writing process should be implemented in the function **writeSubsetData()** in **MinPopVote.c:**

```
bool writeSubsetData(char* filenameW, int totEVs, int totPVs, int wonEVs, MinInf
```

The function should return **false** if the file **filenameW** could not be opened for writing. Otherwise, return **true**.

10. Reflection on Program Results

Right before the 2016 election, **NPR reported** that 23% of the popular vote would be sufficient to win the election, based on the 2012 voting data. They arrived at this number by looking at states with the highest ratio of electoral votes to voting population. This was a correction to their originally-reported number of 27%, which they got by looking at what it would take to win the states with the highest number of electoral votes. But the optimal strategy turns out to be neither of these and instead uses a blend of small and large states. Once you've gotten your program working, try running it on the data from the 2012 election. What percentage of the popular vote does your program say would be necessary to secure the presidency? Then, try a few more years, spread across many decades. Consistently, a candidate could hypothetically be elected President when only 20%-30% of the voters wanted them elected. In our model where there are no third-party candidates, this means that the other candidate received 70%-80% of the votes, but lost. What does this say about the US Presidential Election system? On the non-technical side, are there any interesting patterns/narratives you can glean about the data and the results you are getting? What policy recommendations, if any, could you make from them? Briefly reflect on this topic, referencing actual values output from your program.

Script your reflection in a word-processor and **save it as a .pdf**. Submit this **reflection.pdf** with your Gradescope submission.

Optional Extension Mode (Extra Credit)

There are many variations on how to approach answering the *central question* for this project. Here are some suggestions:

If you look at historical election data, you'll see that it's not all that uncommon for a third-party candidate to win a good number of electoral votes. The Election of 1860, for example, was a four-way race, as was the one in 1912. (The Election of 1836 was an impressive five-way race; the Whig party strategy was really interesting!) The 1992 election was the most recent one in which a major third-party candidate got a good share of the popular vote. Imagine that instead of having to win at least half of the votes in a state to win its electors, you only need to get a third, or a fourth of the votes. How does that change your results?

In the event that there's an Electoral College tie, the choice of who becomes President gets sent to the House of Representatives. Given the historical data, how many different possible outcomes are there that result in an exact Electoral College tie?

After you have completed all required tasks and have built a fully functioning program, take some time to think over possible improved variation on the approach to solving the *central question*. Try implementing your new approach. To do so, copy **app.c** to a file called **ext.c** and implement your approach there. Add new targets to the makefile to build (**make build_ext**) and execute (**make run_ext**) your extension code. If you implement something interesting,

we'd love to see what you've cooked up.

This component is **optional** and only for **extra credit**. Only truly innovative ideas that significantly extend the functionality of the program AND are fully functional will be awarded bonus points. To receive the extra credit you MUST include a file titled **extension.pdf** with your submission, which presents your program extension to the grader with a full explanation of how the program **ext.c** works and how it is an improved approach to tackling the *central question*.

Requirements

- Use the starter code as provided, adding code to complete functions, without making any structural changes: do NOT modify the **State** and **MinInfo** struct definitions (no name change, do not add subitems, do not remove subitems, do not modify subitem definitions, etc.), and do NOT change the function headers (no name changes, do not add parameters, do not remove parameters, do not modify parameter types, etc.). Additional functions are allowed, but likely not necessary. Violations of this requirement will receive a manually graded deduction.
- Solve each task and the program at large as intended. Recursion must be used to find the minimum-PV state subset. Violations of this requirement will receive a manually graded deduction.
- Use the various methods of inputting variables and inputting/outputting data to the program appropriately: command-line arguments and/or interactive user-input set the program settings, file-reading builds state list, and file-writing saves the final results.
- All dynamic heap-allocated memory must be freed to prevent possible memory leaks. This issue is checked by the autograder but may also receive a manually graded deduction.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed using only the concepts and tools introduced or practiced in this course. Use of concepts or tools that are beyond scope of this course are interpreted as academic misconduct and will result in a -100 point manually graded deduction, and may be reported to the Dean of Students office.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-

approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

Tips for Success on this Programming Project

- **Application vs. Testing -** this project asks you to develop functions for a pre-built application. That is, you are not writing code in main() for the primary application. Instead, you are writing many supporting functions. Additionally, a large part of this project is developing your own test cases. The goal is to get better acquainted with the practical reality that autograders will not always be there with a suite of test cases pre-written for you. Whereas the primary application (app.c) is fully written and is somewhat helpful as you develop code, the student-written testing suite (test.c) is where you can harness the power of programming in small chunks, continual testing, and efficient code development.
- **No superfluous print statements -** the functions you write (in MinPopVote.c) should have no print statements. All of the console output is handled in the primary application. Of course, the test cases you write in the student testing suite should have helpful print statements.
- **NOT all Tasks are created equal -** the full Task breakdown is detailed in the *Programming Tasks* section below. The level of scaffolding is designed to lead you through the program development that leaves room for creatively applying course concepts and tools. A natural consequence is that..
    - some Tasks are straightforward and some Tasks are challenging;
    - some Tasks ask you to do one thing specifically and some Tasks are purposefully open-end with multiple components;
    - some Tasks test your ability to follow instructions verbatim and some Tasks require you to practice problem-solving and critical-thinking skills;
    - some Tasks may only take you a few minutes and some Tasks may require an initial attempt followed by a break to do something else only to come back multiple times to tackle the challenge;
- **Trickiest Task is a direct extension of Lab07 -** there is a close connection from the

Backpack Problem solution in Lab07 to the trickiest Task of this project. It is extremely important that you have a deep understanding of the Lab07 coding exercise and a working solution for it.

Submission & Grading (& Test Suite Autograder on Gradescope)

Develop your program in the IDE below. Do NOT use the "Run" button; instead, compile your code using the make utility (e.g. "**make build**") and run the resulting executable (e.g. "**make run**" or "**./app.exe -q**") in the terminal to test your code interactively as you develop your program. Once you have fully-tested the functionality of a given task, use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your program for functionality grading.

The zyBook zyLab IDE autograder tests the **MinPopVote.c** implementation, while feedback on the test suite in test.c is given upon submission to Gradescope. That is, when you submit your **test.c** to Gradescope, you will get a report back on how effective your test cases are at catching faulty implementations of the MinPopVote functions. The Gradescope autograder works by first running a working implementation of MinPopVote through your testing suite to get a baseline functionality; i.e. to ensure a working code passes all of your test cases. This baseline check is called **correct** in the Gradescope autograder. Then, the MinPopVote implementation is *broken* in a variety of distinct ways, which should then result in a failed test case when run through your testing suite. There are 38 total broken cases, labeled **broken01**-**broken38**. Whereas the report does not show how the MinPopVote implementation is broken, since that would detract from the exercise of writing effective and rigorous test cases, the report does provide information on which broken test cases are working or not AND which functions have been broken, which should help you in developing a more thorough set of test cases. The report then gives a suggested test case coverage score, out of 12 total points. Students should interpret this score as follows:

- **12 points**: test cases fully cover all broken implementations, going above and beyond expectations
- **10-11 points**: test cases cover the vast majority of broken implementations, exceeding expectations
- **7-9 points**: test cases cover most broken implementations, but additional coverage is expected

- **4-6 points**: test cases cover about half of broken implementations, which is insufficient coverage
- **1-3 points**: test cases cover a few broken implementations, which is minimal and vastly insufficient coverage
- **0 points**: no test case coverage OR working implementation does not pass ALL test cases

In the end, your test case functions will be manually graded for thoroughness and rigor, but the suggested test case coverage score will be used as a starting point.

When you are ready to formally submit, download the following files from the zyLab IDE file tree: **MinPopVote.c**, **test.c**, and **makefile**, (and **ext.c** if you have an extra credit extension to submit with your work). Also, make sure your **reflection.pdf** for task 10 is ready to upload with the other files. Formally, you must submit your final program files to Gradescope, where your test cases developed in test.c will be manually graded, and your code will be manually inspected for style (details in the syllabus) and project requirements (details listed above). The official code submission is your the version you submit to Gradescope (look for *Project04*). If you fail to submit to Gradescope, you have not formally submitted anything, and the resulting score is a zero for the project.

The standard deadline for the project is **Thursday, October, 24th**. As detailed in the syllabus, early submissions receive +1 extra credit point/day early, up to a maximum of +3 points. Also, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by submitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

The full grading breakdown is as follows, where the zyLab IDE autograder checks functionality and the Gradescope submission autograder checks the student-developed test case suite:

- **86 points for functionality**- autograder points with manual deductions for any style issues (check syllabus for how style is graded) and program requirements not adhered to. Style is checked in both MinPopVote.c and test.c.
  - **Task 1** - [8 points] - **totalEVs( )** and **totalPVs( )**
  - **Task 2** - [8 points] - **setSettings( )**
  - **Task 3** - [8 points] - **inFilename( )** and **outFilename( )**
  - **Task 4** - [8 points] - **makefile** target additions
  - **Task 5** - [8 points] - **parseLine( )**
  - **Task 6** - [8 points] - **readElectionData( )**
  - **Task 7** - [8 points] - **minPopVoteAtLeast( )**
  - **Task 8** - [8 points] - **minPopVoteAtLeastFast( )**

    - **Task 9** - [8 points] - **writeSubsetData( )**
    - **ALL Tasks** - [6 points] - valgrind memory leak and error checks
    - **ALL Tasks** - [8 points] - full program output
- **10 points for test cases** - thoroughness of the student test cases in test.c. There are 9 functions that should be tested: totalEVs(), totalPVs(), setSettings(), inFilename(), outFilename(), parseLine(), readElectionData(), minPopVoteAtLeast(), and minPopVoteAtLeastFast(). Potential for up to +2 extra credit points if submission to Gradescope earns 12 points in test suite check.
- **4 points for reflection** - response for task 10 submitted as a pdf (see above for details)
- **early submission extra credit** - +1 point/day early with a maximum of +3 extra credit points.
- **extension extra credit** - innovative idea to significantly extend the functionality of the program that is fully implemented and functioning. To receive extra credit, up to a maximum of +5 extra credit points, you MUST include a file titled **extension.pdf** with your submission, which presents your program extension to the grader with a full explanation of how the program **ext.c** works and how it is an improved approach to tackling the *central question*.

Citation/Inspiration

The idea for this programming project is greatly inspired by Keith Schwarz at Stanford University, with much of the introduction directly borrowed.

Copyright Statement

LAB ACTIVITY | 14.4.1: Popular Vote Minimizer | 86 / 86

Run | History | Tutorial

MinPopVote.c

```
1   /*-----------------------------------------
2   Program 4: Popular Vote Minimizer Application
3
4   Description: This program allows a user to compile the program with fla
5   For instance, you can use the flag -q while compling to prevent the prc
6   the states. After the user compiles and runs the program, depending on
7   data for each of the 51 states. Then the program will calculate and pri
8   for a given year. A major part of this output is the Minimum Percentage
9   the central question. After printing, the program will write the subset
10
11  Course: CS 211, Fall 2024, UIC
12
13  Author: Shreya Ganguly
14  ----------------------------------------- */
15
16  #include <stdio.h>
17  #include <stdlib.h>
18  #include <string.h>
19  #include <stdbool.h>
20
```

DESKTOP | CONSOLE

Submit for grading

Coding trail of your work What is this?

```
10/11 F 4 ,0 ,8 ,0 ,18 ,0 S 0 ,18 ,12 ,12 ,12 ,16 ,18 ,18 ,16 ,16 U 16
,16 ,20 ,16 ,16 ,16 ,24 ,28 ,28 ,32 ,0 ,32 ,32 ,32 ,32 ,32 ,32 ,40
,36 ,40 ,36 ,40 ,0 ,48 ,48 ,48 M 48 T 0 ,48 ,0 ,48 ,0 ,0 ,48 ,48 ,48
,48 ,0 ,0 ,48 ,48 ,48 ,48 ,48 ,48 ,48 ,48 ,48 ,0 ,48 ,48 ,60 ,0 ,0 ,0 ,0
,78 ,0 ,78 ,78 ,78 ,0 ,0 ,78 ,78 ,86 R 82 ,86 ,86 min:463
```

Latest submission - 10:50 PM CDT on 10/24/24

Submission passed all tests ✓

Total score: 86 / 86

☑ Only show failing tests *(26 tests hidden)*

Open submission's code

5 previous submissions

1:41 AM on 10/24/24	86 / 86	**View** ⌄
1:34 AM on 10/24/24	82 / 86	**View** ⌄
10:49 PM on 10/22/24	86 / 86	**View** ⌄
10:45 PM on 10/22/24	78 / 86	**View** ⌄
10:44 PM on 10/22/24	78 / 86	**View** ⌄

Trouble with lab?

Feedback?

Activity summary for assignment: Programming Project 04 - Win Presidency With Only 20% Support (Popular Vote Minimizer)

86 / 86 points

Due: 10/24/2024, 11:59 PM CDT

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Completion details