Project 3 Report

1. Notable obstacles I overcame:

One of the main obstacles I had was facing tracking positions through the poll data string. Because there were predictions that could have either one or two digits for the vote count, I had to be very careful about how I advanced the position counter after reading digits. I solved for this by first checking for one digit and then checking if a second digit existed.

Another obstacle I had was converting digit characters to numbers. I had to remember how to do this, and after I did, I subtracted the character `0` from a digit character, giving its numeric value, and for two digits, multiplying the first digit by 10 and adding the second.

2. Design of program:

The design of the program centers around the two main functions, hasRightSyntax and computeVotes. In hasRightSyntax, it determines if the string follows the required format for poll data. It does this by first handling the special case of an empty string, then processing the string character by character. It examines each part of the prediction—party code, vote count, and state code. It also has a position counter that returns false if any component is invalid and advances if it is valid.

Pseudocode for hasRightSyntax:  
  
if string is empty

return true

for each prediction in the string:

verify next character is a letter (party)

if not, return false

verify next 1-2 characters are digits (votes)

if not, return false

verify next 2 characters form valid state code

if not, return false

advance to next prediction

return true (whole string was valid)

The function computeVotes builds on hasRightSyntax by performing the validation checks in a specific order—the party must be a letter, string must have valid syntax, and no prediction can have zero votes. This allows the function to save the original voteCount to restore it if needed. When the string is valid, it accumulates the votes when it finds predictions for the specific party. The empty string returns 0 votes for a valid party.

Pseudocode for computeVotes:

save original voteCount

if party parameter is not a letter

return 3

if string format is invalid (using hasRightSyntax)

restore voteCount

return 1

if string is empty

set voteCount to 0

return 0

initialize voteCount to 0

for each prediction in string:

get party and vote count

if votes is zero

restore original voteCount

return 2

if current party matches input party (ignoring case)

add votes to running total

return 0 with final vote count

3. Test data:

Empty String Tests:

assert(hasRightSyntax("")); // Empty string must be valid

votes = 666;

assert(computeVotes("", 'D', votes) == 0 && votes == 0); // Empty string with valid party: must set votes to 0

votes = 666;

assert(computeVotes("", '@', votes) == 3 && votes == 666); // Empty string with invalid party: must preserve votes

votes = 666;

assert(computeVotes("", '0', votes) == 3 && votes == 666); // Empty string with digit as party

votes = 666;

assert(computeVotes("", ' ', votes) == 3 && votes == 666); // Empty string with space as party

Basic Format Tests:

assert(hasRightSyntax("D5CA")); // Basic valid format: letter-digit-state

assert(hasRightSyntax("D15CA")); // Two digits in vote count

assert(hasRightSyntax("D05CA")); // Leading zero in vote count is valid

assert(hasRightSyntax("D99CA")); // Maximum two-digit vote count

assert(!hasRightSyntax("DCA")); // Missing vote count

assert(!hasRightSyntax("5CA")); // Missing party code

assert(!hasRightSyntax("D5C")); // Incomplete state code

assert(!hasRightSyntax("DD5CA")); // Two letters in a row invalid

assert(!hasRightSyntax("D5CAD")); // Incomplete prediction after valid one

Case Sensitivity Tests:

assert(hasRightSyntax("d5CA")); // Lowercase party is valid

assert(hasRightSyntax("D5ca")); // Lowercase state is valid

votes = -999;

assert(computeVotes("d5CA", 'D', votes) == 0 && votes == 5); // Party code is case insensitive

votes = -999;

assert(computeVotes("D5CA", 'd', votes) == 0 && votes == 5); // Party parameter is case insensitive

Multiple Predictions:

assert(hasRightSyntax("D5CAR4NY")); // Two predictions

assert(hasRightSyntax("D5CAR4NYL3CT")); // Three predictions

assert(!hasRightSyntax("D5CAD")); // Incomplete second prediction

assert(!hasRightSyntax("D5CARD4TX")); // Missing separation between predictions

Error Priority Tests

votes = -999;

assert(computeVotes("D5CA", '@', votes) == 3 && votes == -999); // Must return 3 for invalid party

votes = -999;

assert(computeVotes("@5CA", 'D', votes) == 1 && votes == -999); // Must return 1 for invalid syntax

votes = -999;

assert(computeVotes("D0CA", 'D', votes) == 2 && votes == -999); // Must return 2 for zero votes

Invalid Party Parameter Tests:

votes = -999;

assert(computeVotes("D5CA", '0', votes) == 3 && votes == -999); // Digit as party invalid

votes = -999;

assert(computeVotes("D5CA", '!', votes) == 3 && votes == -999); // Symbol as party invalid

votes = -999;

assert(computeVotes("D5CA", ' ', votes) == 3 && votes == -999); // Space as party invalid

Vote Counting Tests:

votes = -999;

assert(computeVotes("R99TXD99CA", 'D', votes) == 0 && votes == 99); // Maximum two-digit vote count

votes = -999;

assert(computeVotes("R40TXD54CAr6MS", 'D', votes) == 0 && votes == 54); // Sum only matching party

votes = -999;

assert(computeVotes("R40TXD54CAr6MS", 'R', votes) == 0 && votes == 46); // Case insensitive matching

votes = -999;

assert(computeVotes("D5CAD4NY", 'D', votes) == 0 && votes == 9); // Sum multiple predictions

votes = -999;

assert(computeVotes("R40TXD54CAr6MS", 'L', votes) == 0 && votes == 0); // Party with no predictions

Multiple Error Tests:

votes = -999;

assert(computeVotes("@5CA", '@', votes) == 1 ||

computeVotes("@5CA", '@', votes) == 3); // Can return either 1 or 3 when both errors present

votes = -999;

assert(computeVotes("D0CX", 'D', votes) == 1 ||

computeVotes("D0CX", 'D', votes) == 2); // Can return either 1 or 2 when both errors present

State Code Tests:

assert(hasRightSyntax("D5DC")); // DC is valid

assert(hasRightSyntax("D5NY")); // NY is valid

assert(hasRightSyntax("D5Ca")); // Mixed case state valid

assert(!hasRightSyntax("D5XX")); // Invalid state letters

assert(!hasRightSyntax("D5C.")); // Punctuation in state invalid

assert(!hasRightSyntax("D5C9")); // Number in state invalid

All test cases are handled correctly.