


Assignment 4: Recursion to the Rescue!

BASED ON AN ASSIGNMENT BY KEITH SCHWARZ

 **DUE: MONDAY, OCTOBER 29, 11AM**

 **MAY BE DONE IN PAIRS**

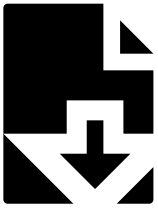
Recursion is an extremely powerful problem-solving tool with tons of practical applications. This assignment consists of four real-world recursive problems, each of which we think is interesting in its own right. By the time you're done with this assignment, we think you'll have a much deeper appreciation both for recursive problem solving and for what sorts of areas you can apply your newfound skills to. You may optionally choose to work on this assignment with a partner.

This assignment, like the preceding one, explores a lot of different concepts in recursion. If you start typing out code without a clear sense of how your recursive solution will work, chances are you'll end up going down the wrong path. Before you begin coding, try thinking about the recursive structure of the code you're writing. Does anything fall into one of the nice categories we've seen before (listing subsets, permutations, combinations, etc.)? Is there a nice decision tree you can draw? If you're using backtracking, what choices can you make at each step? Talking through these questions before you start coding and making sure you have a good conceptual understanding of what it is that you need to do can save you many, many hours of coding and debugging.

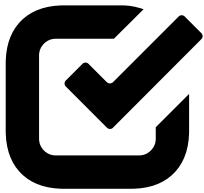
Due Date: October 29 at 11:00am.

Submit: You can submit multiple times. We will grade your latest submission. Submit via Paperless [here](#).

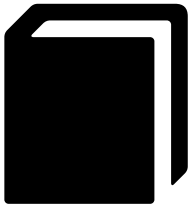
Important Files and Links



Starter Code



Style Guide



Stanford Library Docs

Turn in only the following files:

1. **RecursionToTheRescue.cpp**, the C++ code for the 4 required functions (excluding a main function)
2. **DNADetectiveTests.cpp**, the C++ code for the DNA Detective unit tests
3. **debugging.txt**, a file detailing a bug you encountered in this assignment and how you approached debugging it (provided in the res/ folder)

Recommended Timetable

You have a lot of time to complete this assignment, so make sure that you're not putting it off to the last minute. Here's a rough timetable that we think should be pretty manageable:

- Read over this handout to see what questions are asked as soon as you get it.
- Complete Problem One (Doctors) by Monday, October 22.
- Complete Problem Two (Disaster) by Wednesday, October 24.
- Complete Problem Three (DNA) by Friday, October 26.
- Complete Problem Four (Presidency), implement any extensions, and submit everything by Monday, October 29.

Best of luck on this assignment!

Development Strategy and Hints

- ***Your functions must actually use recursion;*** it defeats the point of the assignment to solve these problems iteratively!
- ***Do not change any of the function prototypes.*** We have given you function prototypes for each of the problems in this assignment, and the functions you write must match those prototypes exactly – the same names, the same arguments, the same return types. You are welcome to add your own helper functions if you would like, unless where noted otherwise.
- ***Test your code thoroughly!*** We'll be running a battery of automated tests on your code, and it would be a shame if you turned in something that almost worked but failed due to a lack of proper testing.
- ***Recursion is a tricky topic,*** so don't be dismayed if you can't immediately sit down and solve these problems. Please feel free ask for advice and guidance if you need it. Once everything clicks, you'll have a much deeper understanding of just how cool a technique this is. We're here to help you get there!
- ***The starter program*** includes ways to test all of the 4 problems on this assignment, and for the provided test cases will print out various messages informing you if your solution seems incorrect. Test cases are stored in the `res/` folder of the starter project.

Style

As in other assignments, you should follow our Style Guide for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1-3 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem:

Recursion: Part of your grade will come from appropriately utilizing recursion to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid "arm's length" recursion, which is where the true base case is not found and unnecessary code/logic is stuck into the recursive

case. Redundancy in recursive code is another major grading focus; avoid repeated logic as much as possible. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While not new to this assignment, we want to stress that you should not make any global or static variables (unless they are constants declared with the `const` keyword). Do not use globals as a way of getting around proper recursion and parameter-passing on this assignment.

Unit Tests: Make sure that your unit tests for problem 3 (DNA Detective) test as small, isolated components of your function as possible. Make sure to give each test a detailed name, and comment above it explaining what it is testing.

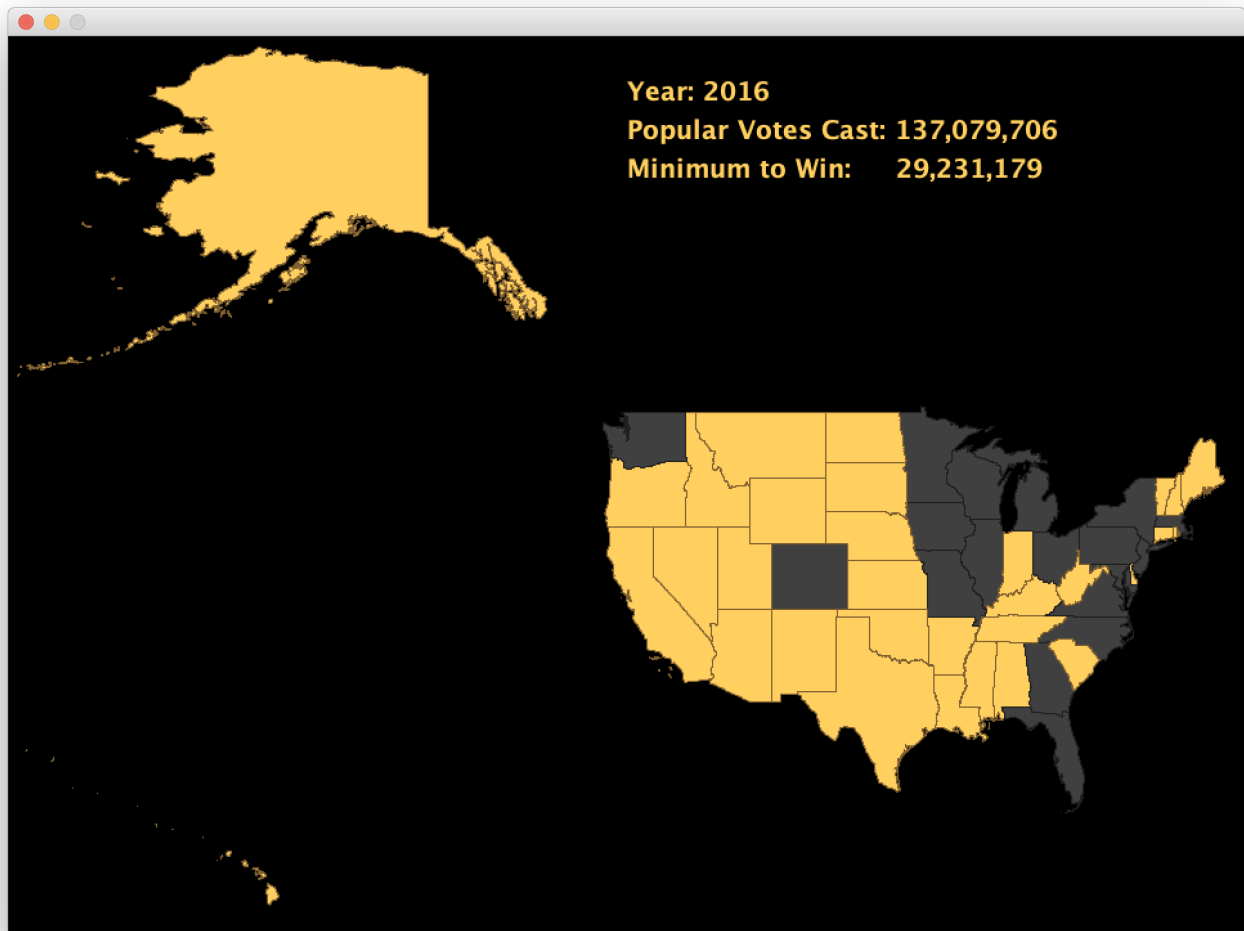
Pair Programming

You may optionally work on this assignment as a pair programming assignment. If you would like to work in a pair on this assignment, you may, under the following conditions:

- Both partners must be in the same section (time and location).
 - At no time should partners be coding alone. To be clear: all coding must be done together, on a single computer. One partner should type, and the other should watch over that partner's shoulder. Each partner should take turns at the keyboard.
 - When you go to the LaIR or office hours for help on your assignment, both students must be present in the pair to receive help. Only one partner should sign up for help in the LaIR Queue at a time.
 - Pair programming is less about sharing the workload and more about sharing ideas and working together. During the IG for the assignment, if a Section Leader does not think both partners contributed equally and worked together, the SL may forbid pair programming on a future assignment.
 - Pairs who "split the assignment" where one partner does part of the assignment alone, and the other partner does another part of the assignment alone, will be in violation of the Honor Code.
-

[Doctors Without Orders](#)[Disaster Preparations](#)[DNA Detective](#)[Winning the Presidency](#)[Extra Features](#)

Problem 4: Winning the Presidency



The President of the United States is not elected by a popular vote, but by a majority vote in the Electoral College. Each state, plus DC, gets some number of electors in the Electoral College, and whoever they vote in becomes the next President. For the purposes of this problem, 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 only 99 people, you'd need 50 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 problem explores the following question: under these assumptions, what's the fewest number of popular votes you can get and still be elected President?

Imagine that we have a list of information about each state, represented by this handy struct:

```
struct State {  
    string name; // The name of the state  
    int electoralVotes; // How many electors it has  
    int popularVotes; // The number of people in that state who voted  
};
```

This struct contains the name of the state, the number of electors (to the Electoral College) the state gets, and its voting population. Your task is to write a function

```
MinInfo minPopularVoteToWin(const Vector<State>& states);
```

that takes as input a list of all the states that participated in the election (plus DC, if appropriate), then returns some information about the minimum number of popular votes you'd need in order to win the election (namely, how many votes you'd need, and which states you'd carry in the process). The **MinInfo** structure is defined in the **RecursionToTheRescue.h** header and is essentially just a pair of a minimum popular vote total plus the list of states you'd carry in the course of reaching that total:

```
struct MinInfo {  
    int popularVotesNeeded; // How many popular votes you'd need.  
    Vector<State> statesUsed; // Which states you'd carry in the course of doing so.  
};
```

To implement this function, you are **required** to implement the following recursive helper function that solves the following problem (this cannot be a wrapper function):

```
/* What are the minimum votes and states needed to get at least "electoralVotesNeeded"  
 * electoral votes, using only states from index "minStateIndex" and greater in the "sta  
MinInfo minPopularVoteToGetAtLeast(int electoralVotesNeeded,  
                                   const Vector<State>& states, int minStateIndex);
```

Notice that if you solve this problem with **electoralVotesNeeded** set to a majority of the total electoral votes and with **minStateIndex** as 0, then you've essentially solved the original problem (do you see why?). You **must** make your original function a wrapper around this helper function that solves this specific problem.

In the course of solving this problem, you might find yourself in the unfortunate situation where, for some specific values of **electoralVotesNeeded** and **minStateIndex**, there's no possible way to get that many votes using only the states from that index onwards. For example, if you're short 75 electoral votes and only have a single state left, there's nothing that you can do to win the election. In that case, you may want to have this helper function indicate that it's not possible to secure that many votes. We recommend using the special value **INT_MAX**, which represents the maximum possible value that you can store in an integer. The advantage of this sentinel value is that you're already planning on finding the strategy that requires the fewest popular votes, so if your sentinel value is greater than any possible legal number of votes, always choosing the option that requires the minimum number of votes will automatically pull you away from the sentinel value. ***Just remember to not add anything to INT_MAX!*** Doing so will cause it to turn into a negative number, which may introduce bugs in your program.

When you first implement this function, we strongly recommend testing it out using the simplified test cases we've provided you from the main menu. These test cases use real election data, but only consider ten states out of a larger election. That should make it easier for you to check whether your solution works on smaller examples.

Without using memoization, it's almost guaranteed that your code won't be fast enough to work with the full elections data. There's just too many subsets of the states to consider. To scale this up to work with actual elections data, ***you'll need to work memoization into your solution.*** The good news is that, if you've followed the strategy we've outlined above, you should find that it's relatively straightforward to introduce memoization into your solution. You may add additional parameters for memoization to the previously-mentioned helper

function to do so. Remember that memoization helps cache previously-computed calculations. There are (in our tests) at most 270 electoral votes and 51 different possible values for **minStateIndex** = $270 \times 51 = 13770$ possible ways to call this function, at least from our tests. If we are blindly considering all possible subsets of 51 states, that's 2^{51} possible subsets! So caching will undoubtedly cut down on the work we are doing. (Hint: a nested ADT, or a SparseGrid, may be helpful in storing the cache for memoizing. Think about how you can quickly look up cached values). Once you've gotten that working, try running your code on full elections data. I think you'll be pleasantly surprised by how fast it runs!

Here are some general notes on this problem:

- The historical election data – and our reduced test cases – do not always include all 50 current US states plus DC, either because those states didn't exist yet, or DC didn't have the vote, or because those states didn't participate in the election, so you shouldn't assume that you'll get them as input.
- The total number of Electoral College votes to win the election depends on the number of electors, which varies over time. Although you currently need 270 electoral votes to become President, you should not assume this in your solution.
- Remember that in all elections you need strictly more than half the votes to win. If there are either 100 or 101 people in a state, you need 51 votes to win its electors. If there are either 538 or 539 total electoral votes, you'd need 270 electoral votes to become president.

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?

(A note: the historical election data here was compiled from a number of sources. In many early elections the state legislatures decided how to choose electors, and so in some cases we extrapolated to estimate the voting population based on the

overall US population at the time and the total fraction of votes cast. This may skew some of the earlier election results. However, to the best of our knowledge, data from 1868 and forward is complete and accurate. Please let us know if you find any errors in our data!)

© Stanford 2018 | Created by Chris Piech and Nick Troccoli.
CS 106X has been developed over time by many talented teachers.