CS106B	Handout 17S
Winter 2020	February 5, 2020

Practice CS106B Midterm I Solutions

Here's one possible set of solutions for the practice midterm questions. Before reading over these solutions, please, please, please work through the problems under semi-realistic conditions (spend three hours, have a notes sheet, etc.) so that you can get a better sense of what taking a coding exam is like.

This solutions set contains some possible solutions to each of the problems in the practice exam. It's not meant to serve as "the" set of solutions to the problems – there are lots of ways you can go about solving each problem here. In other words, if your solution doesn't match ours, don't take that as a sign that you did something wrong. Feel free to stop by the CLaIR, to ask questions on Piazza, or to ask your section leader for their input!

The solutions we've included here are a lot more polished than what we'd expect most people to turn in on an exam. You'll three hours to complete this whole exam. We have a lot of time to write up these solutions, clean them up, and try to get them into a state where they'd be presentable as references. Don't worry if what you wrote isn't as clean as what we have here, but do try to see if there's anything we did here that would help you improve your own coding habits.

Problem One: Containers I (8 Points)

Here's one possible solution:

```
/* Given a district and a party, how many votes did that party get in that
 * district?
 */
int votesIn(const Vector<string>& district, const string& party) {
    int result = 0;
    for (string vote: district) {
        if (vote == party) {
            result++;
        }
    }
    return result;
}

double gerrymanderingRatio(const Vector<Vector<string>>& districts, string party) {
    int numVotes = 0;         // Total votes cast
    int numPartyVotes = 0;    // Votes for this party
    int numDistrictsWon = 0; // Districts won by this party

    for (Vector<string> district: districts) {
        numVotes += district.size();

        int districtVotes = votesIn(district, party);
        numPartyVotes += districtVotes;

        /* We need 50% or more to win. Be careful with the math! Integer division
         * rounds down.
         */
        if (districtVotes >= district.size() / 2) {
            numDistrictsWon++;
        }
    }

    /* What fraction of districts did we win? Note the cast to double so that we
     * don't have integer division round us down to zero.
     */
    double districtFraction = double(numDistrictsWon) / districts.size();

    /* What fraction of the votes did we earn? */
    double voteFraction = double(numPartyVotes) / numVotes;

    return districtFraction / voteFraction;
}
```

Why we asked this question: This question was designed to let you show us what you'd learned about basic C++ programming (task decomposition, variable types, integer division, strings, and `Vectors`). Plus, we figured that this application was an interesting way of exploring how simple data analysis can give you insight into contemporary politics!

Problem Two: Containers II (8 Points)

One way to solve this problem is to count the number of cities that are at the centers of stars – this ensures that each star gets counted once and exactly once. We can check whether a city is a star center by checking whether it has at least three neighbors and that each of those neighbors only has a single neighbor. Here's one way to do this :

```
bool isStarCenter(const string& city,
                  const HashMap<string, HashSet<string>>& network) {
    /* Stars need at least four cities, including the center. */
    if (network[city].size() < 3) {
        return false;
    }

    /* All cities aside from the center must just link to the center. */
    for (string neighbor: network[city]) {
        if (network[neighbor].size() != 1) {
            return false;
        }
    }
    return true;
}

int countStarsIn(const HashMap<string, HashSet<string>>& network) {
    int result = 0;
    for (string city: network) {
        if (isStarCenter(city, network)) {
            result++;
        }
    }
    return result;
}
```

Why we asked this question: This question was designed to get you playing around with container types (here, `HashMap` and `HashSet`) and nested containers while also touching on an interesting algorithmic problem. Specifically, we hoped you'd think about the problem abstractly ("how do I determine whether a cluster of items is a star when I'm working one element at a time?"), figure out a strategy that works, and then translate your solution into C++.

Problem Three: Recursion I (8 Points)

There are many ways to approach this problem. Our solution works by keeping track of two groups – the group of people who haven't proofread anything and the group of people who haven't had their sections proofread – and at each point chooses a person and gives them a section to read. We try out all options except for those that directly assign a person their own section.

Here's what this looks like in code:

```
void listAssignmentsRec(const HashSet<string>& unassignedReaders,
                        const HashSet<string>& unassignedSections,
                        const HashMap<string, string>& assignments) {
    /* Base Case: If everyone is assigned to read something, print out the
     * reading assignments.
     */
    if (unassignedReaders.isEmpty()) {
        cout << assignments << endl;
    }
    /* Recursive Case: Choose a reader and look at all ways to give them
     * something to read.
     */
    else {
        string reader = unassignedReaders.first();

        /* Loop over all sections that haven't yet been assigned. */
        for (string section: unassignedSections) {
            /* Don't assign someone to read their own section. */
            if (section != reader) {
                /* Explore all options that include giving this person this
                 * section to read.
                 */
                auto newAssignments = assignments;
                newAssignments[reader] = section;

                listAssignmentsRec(unassignedReaders  - reader,
                                   unassignedSections - section,
                                   newAssignments);
            }
        }
    }
}

void listCheckingArrangements(const HashSet<string>& people) {
    listAssignmentsRec(people, people, {});
}
```

Why we asked this question: We chose this particular problem for a number of reasons. First, we included it because it's closely related to the classic problem of listing permutations, which by this point we hoped you'd have a good handle on. Second, we liked how this problem forced you to think back to the intuition behind how to generate permutations (choose a position and think of all the things that could go into that position), since the problem structure was sufficiently different from the traditional permutations problem as a consequence of having the items stored in an inherently unordered structure. Third, we thought this problem would allow you to demonstrate what you'd learned about using extra arguments to recursive functions in order to keep track of the decisions made so far and decisions left to make; our solution requires two extra arguments to remember unassigned sections and unassigned people separately. Finally, we wanted to include this problem because it has nice mathematical properties; the assignments you're asked to list here are called ***derangements*** and show up frequently in CS theory.

Common mistakes: We saw a number of mistakes on this problem that were focused on small details of the problem rather than the overall structure. A number of solutions correctly attempted to keep track of which sections weren't read and who hadn't read anything, but accidentally looped over the wrong one when trying to determine which section to assign or which person to assign something to. Some solutions did correctly list all the options, but did so by generating all permutations and then filtering out invalid ones at the last second (which was specifically prohibited by the problem statement). Other solutions modified a map or set while iterating over it, which leads to runtime errors.

If you had trouble with this problem: Our advice on how to proceed from here varies a bit based on what specific difficulty you were having.

If you looked at this problem and didn't recognize that you were looking at a permutations-type problem, we would recommend trying to get more practice looking at these sorts of problems and broadly categorizing them into subtypes like "permutation problems," "subset problems," etc. Section Handout 3 and Section Handout 4 have a number of good problems to work through that might be good spots to look for questions like those.

If you had trouble avoiding repeated assignments in the course of solving this problem, we'd recommend taking some time to look at the different recursive functions we've written in the past that enumerate options to see how their design prevents duplicates. Why, for example, does the "list all subsets" function we wrote when we first talked about recursion not list any subsets twice? Why does the "list all permutations" function list each option once? See if you can connect this back to recursion trees.

If you weren't sure how to keep track of which people were assigned sections and which sections had been assigned readers, it might be worth practicing working with the decision tree model of recursive problem-solving. At each point in time, your goal is to make some sort of choice about what the next option will be. What information would you need to have access to in order to make that choice? What would the possible choices look like? A little extra practice with this skill can go a long way.

Problem Four: Recursion II (8 Points)

One way to solve this problem is to focus on a single person. If we choose this person, we'd want to pick, from the people who aren't friends with that person, the greatest number of people that we can. If we exclude this person, we'd like to pick from the remaining people the greatest number of people that we can. One of those two options must be optimal, and it's just a question of seeing which one's better.

```
/**
 * Given a network and a group of permitted people to pick, returns the largest
 * group you can pick subject to the constraint that you only pick people that
 * are in the permitted group.
 *
 * @param network The social network.
 * @param permissible Which people you can pick.
 * @return The largest unbiased group that can be formed from those people.
 */
Set<string> largestGroupInRec(const HashMap<string, HashSet<string>>& network,
                              const HashSet<string>& permissible) {
    /* Base case: If you aren't allowed to pick anyone, the biggest group you
     * can choose has no people in it.
     */
    if (permissible.isEmpty()) return {};

    /* Recursive case: Pick a person and consider what happens if we do or do not
     * include them.
     */
    string next = permissible.first();

    /* If we exclude this person, pick the biggest group we can from the set of
     * everyone except that person.
     */
    auto without = largestGroupInRec(network, permissible - next);

    /* If we include this person, pick the biggest group we can from the set of
     * people who aren't them and aren't one of their friends, then add the
     * initial person back in.
     */
    auto with = largestGroupInRec(network,
                                  permissible - next - network[next]) + next;

    /* Return the better of the two options. This uses the ternary conditional
     * operator ?:. The expression expr? x : y means "if expr is true, then
     * evaluate to x, and otherwise evaluate to y."
     */
    return with.size() > without.size()? with : without;
}

HashSet<string>
largestUnbiasedGroupIn(const HashMap<string, HashSet<string>>& network) {
    HashSet<string> everyone;
    for (string person: network) {
        everyone += person;
    }
    return largestGroupInRec(network, everyone);
}
```

Why we asked this question: We included this problem for several different reasons. First, we wanted to include at least one "subset-type" recursion on this exam and figured that this particular problem would be a nice way to do this. Second, we liked how this problem connected back to the assignments: it's along the lines of the Shift Scheduling problem from Assignment 3, in that you need to find a group of items that don't conflict with one another. Third, we thought that the recursive optimization here was particularly elegant to code up once you had the right insights. You can solve this either by keeping track of what options are permissible (as we did here) or by which options are impermissible (track which items can't be chosen again), and can code the solution up in a number of different ways. Finally, this problem is related to a famous problem called the ***maximum independent set problem***, which has a rich history in computer science and shows up in all sorts of interesting practical and theoretical contexts.

Common mistakes: We saw a number of solutions that recognized that picking someone precluded any future call from choosing any of that person's friends, but which forgot to *also* mark that the chosen person couldn't be included in future recursive calls (oops!), which is fairly easy to correct but an easy mistake to make.

If you had trouble with this problem: As with Problem Three, our advice on what the next steps should be if you had trouble with this particular problem depend on which aspect of the problem you were having trouble with.

If, fundamentally, you didn't recognize that this problem was a subset-type problem, then your first step should probably be to just get a little bit more practice working with recursion problems and trying to recognize their structure. You might benefit from revisiting some of the older section problems (Section Handout 3 and Section Handout 4 have a bunch of great recursion problems on them. Without looking at the solutions, try to see if you can predict which of them use a subsets-type recursion, which use a permutations-type recursion, and which use something altogether different. Or go back to Assignment 3 and make sure you have a good understanding about which problems fall into which types.

If you had some trouble keeping track of the decisions you'd made so far and how they interacted with the future (that is, making sure you didn't pick the same thing twice), you may want to get a bit more practice converting between the different container types. This problem is a lot easier to solve if you have the right data structure to remember what's already been seen and what hasn't yet been considered. As a general rule, try to find a way to solve the problems you're working on that involves the minimal amount of "fighting with the code," even if that means doing some sort of conversion step beforehand.