# Practice CS106B Midterm II 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)

One way to solve this problem is to keep track of the most-recently-seen word and to use that to help fill in the predecessor map. Here's a solution that uses this strategy:

```
HashMap<string, Lexicon> predecessorMap(istream& input) {
    HashMap<string, Lexicon> result;
    string lastWord; // Most-recently-read string, initially empty as a sentinel.

    /* The canonical "loop over the lines of a file" loop. */
    for (string line; getline(input, line); ) {
        for (string token: tokenize(line)) {
            /* For consistency, we convert everything to lower-case. */
            token = toLowerCase(token);

            /* If this isn't a word, we don't care about it. */
            if (!isalpha(token[0])) {
                continue;
            }

            /* Update the predecessor map, unless this is the first word. */
            if (lastWord != "") {
                result[token].add(lastWord);
            }
            lastWord = token;
        }
    }
    return result;
}
```

***Why we asked this question:*** This question was designed to explore basic C++ constructs (nested loops, iterating over the contents of a file, etc.), container types (specifically, `HashMap`s, `Vector`s, and `Lexicons`), and general problem-solving (how do you track the information needed to build the result up?). This problem is somewhat related to the Plotter from Assignment 1 – you have to loop over the contents of a file while keeping track of some state that persists across loop iterations.

## Problem Two: Container Classes (8 Points)

Here's one possible way to solve this problem. This one works by repeatedly finding a person without at least *k* friends left in the core, then removing them from the core.

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

HashSet<string> kCoreOf(const HashMap<string, HashSet<string>>& network, int k) {
    /* Initially, assume everyone is in the core. */
    auto core = allPeopleIn(network);

    while (true) {
        string lonelyPerson;
        bool isLonelyPerson = false;

        for (string person: core) {
            /* See how many people they're friends with who haven't yet been
             * filtered out.
             */
            if ((network[person] * core).size() < k) {
                lonelyPerson = person;
                isLonelyPerson = true;
                break;
            }
        }

        if (!isLonelyPerson) break;
        core -= lonelyPerson;
    }

    return core;
}
```

Here's another solution. This one follows the same basic principle as the previous solution, except that instead of maintaining a set of people in the core and leaving the social network unchanged, this one actively modifies the social network whenever someone is removed.

```
HashSet<string> kCoreOf(const HashMap<string, HashSet<string>>& network, int k) {
    auto coreNet = network;

    while (true) {
        string lonelyPerson;
        bool isLonelyPerson = false;

        for (string person: coreNet) {
            /* See how many people they're friends with who haven't yet been
             * filtered out.
             */
            if (coreNet[person].size() < k) {
                lonelyPerson = person;
                isLonelyPerson = true;
                break;
            }
        }

        if (!isLonelyPerson) break;

        /* Remove this person from the network, and make sure the remaining
         * people don't consider them a friend.
         */
        for (string acquaintance: coreNet[lonelyPerson]) {
            coreNet[acquaintance] -= lonelyPerson;
        }
        coreNet.remove(lonelyPerson);
    }

    /* Build a set of all the remaining people. */
    HashSet<string> result;
    for (string member: coreNet) {
        result += member;
    }

    return result;
}
```

Here's another route that works recursively. It's essentially the first solution, but written recursively rather than iteratively.

```
HashSet<string> kCoreOfRec(const HashMap<string, HashSet<string>>& network, int k,
                           const HashSet<string>& remainingFolks) {
    string lonelyPerson;
    bool isLonelyPerson = false;

    for (string person: remainingFolks) {
        /* See how many people they're friends with who haven't yet been
         * filtered out.
         */
        if ((network[person] * remainingFolks).size() < k) {
            lonelyPerson = person;
            isLonelyPerson = true;
            break;
        }
    }

    /* Base case: If everyone has at least k friends, the remaining people form
     * the core.
     */
    if (!isLonelyPerson) return remainingFolks;

    /* Recursive case: Otherwise, the lonely person isn't in the core, and we
     * want the core of what's left if we remove them.
     */
    return kCoreOfRec(network, k, remainingFolks - lonelyPerson);
}

HashSet<string> kCoreOf(const HashMap<string, HashSet<string>>& network, int k) {
    HashSet<string> everyone;
    for (string person: network) {
        everyone += person;
    }
    return kCoreOfRec(network, k, everyone);
}
```

This next solution uses a different perspective on the problem. For starters, note that no one in the network with fewer than *k* friends can possibly be in the *k*-core. So we could begin by making a candidate *k*-core in the following way: build up a smaller network of people who purely have *k* or more friends in the original network. When doing this, we might find that some of those people no longer have *k* or more friends, since some of their friends might not have been copied into the network. We therefore repeat this process – copying over the people in the new network with *k* or more friends – until we converge on the *k*-core. We could do this either iteratively or recursively, and just for fun I've written it recursively here:

```
HashSet<string> kCoreOf(const HashMap<string, HashSet<string>>& network, int k) {
    /* Build up a new social network consisting of everyone with k or more
     * friends. For simplicity later on, we're going to make both a new Map
     * representing the network and a new Set of the people in that network.
     */
    HashMap<string, HashSet<string>> coreNetwork;
    HashSet<string> core;

    /* Copy over the people with at least k friends. */
    for (string person: network) {
        if (network[person].size() >= k) {
            coreNetwork[person] = network[person];
            core += person;
        }
    }

    /* If we copied everyone over, great! The set of everyone in the network is
     * the k-core.
     */
    if (core.size() == network.size()) return core;

    /* Otherwise, someone didn't make it. We need to therefore take the new
     * social network and filter down the friend lists purely to the people in
     * the new network.
     */
    for (string person: coreNetwork) {
       coreNetwork[person] *= core; // Intersect friends with people in the core
    }

    return kCoreOf(coreNetwork, k);
}
```

***Why we asked this question:*** We included this question as practice working with the fundamental container types that we've used over the quarter (here, `HashMap`s and `HashSet`s). You've used these types in a number of contexts, and we figured this would be a great place for you to demonstrate what you'd learned along the way. We also included this problem to give you practice breaking a larger task down into smaller, more manageable pieces and translating high-level intuitions into code.

Plus, we thought that this concept from social network analysis was sufficiently interesting that it was worth sharing with you!

***Common mistakes:*** By far the most common mistake on this problem was to only consider each person once when deciding whether to remove them. To illustrate why this is an issue, suppose you find a person in the network who has fewer than *k* friends and therefore needs to be removed from the network. That in turn might mean that someone who previously had *k* or more friends no longer does, and therefore needs to get removed as well. If you make a single pass over the network and have already checked that person, you'll miss that they now need to be removed. In other words, it's not enough to just find everyone with fewer than *k* friends and remove them; you have to then look at the resulting network and repeat this process until it stabilizes on the *k*-core.

Another issue we saw on this problem had to do with maintenance of the social network after removing a person. Suppose you remove a person from the social network. This requires two steps to execute properly: first, you have to remove them as a key from the network; second, you have to remove them from each of their friends' associated `HashSet`s. If you forget to do this – or don't do something equivalent – then you can end up in a situation where there's a person in the network who believes they're friends with more people than they actually are. Think of it this way – people leave social networks because their friends leave; if you have a friend who leaves a network and you don't realize they've left, their departure is unlikely to influence you.

Another issue we saw, which was somewhat subtle, had to do with modifying collections when iterating over them. If you are iterating over a `HashMap` or `HashSet` using a range-based `for` loop and you remove an element from the `HashMap` or `HashSet`, you will trigger an error and cause the loop to stop operating properly. We thought we'd point this out because modifying collections this way causes problems in most programming languages.

***If you had trouble with this problem:*** This problem is mostly about syncing your intuition for what a piece of code should do with the code itself. If you missed out on the nuances above – either by forgetting to make multiple passes over the network or by forgetting to update the network – there are a couple strategies you can use to improve. The first would be to ***draw lots of pictures***, one of the major themes from this quarter. Once you have a draft of the code, pick a sample network and see what happens when you try your code out on it. Since `HashMap`s and `HashSet`s are unordered containers, you might ask what happens if you visit the people in the network in different orders. Do you get back the answer you intended to get? Or does something else happen?

The second would be to just get more practice and reps with your coding skills. Work through some of the older section problems from Section Handout 2 or the chapter exercises from the textbook's sections on collections. There are some great exercises in there, and blocking out the time to work on them can really make a difference.

## Problem Three: Recursion I (8 Points)

Here's one possibility. This solution works by trying all ways of providing the first person something they like, removing from the list of people everyone who now has something they like, then trying to cover the remaining people as efficiently as possible.

```
HashSet<string> smallestCarePackageFor(const Vector<HashSet<string>>& preferences) {
    /* Base case: If there are no people, we need no treats. */
    if (preferences.isEmpty()) {
        return {};
    }

    /* Track the best option so far. */
    HashSet<string> result; // Empty set is a sentinel.

    /* Otherwise, pick a person, then try all ways of sending them something. */
    for (string option: preferences[0]) {
        /* Try sending this option. That will cover everyone who also likes this
         * option.
         */
        Vector<HashSet<string>> remaining;
        for (auto person: preferences) {
            if (!person.contains(option)) {
                remaining += person;
            }
        }

        /* See what this would require us to send. That's what's needed to cover
         * everyone else, plus this option.
         */
        auto bestWithThis = smallestCarePackageFor(remaining) + option;

        /* See if this is better than the best option so far. */
        if (result.isEmpty() || result.size() > bestWithThis.size()) {
            result = bestWithThis;
        }
    }

    return result;
}
```

Here's another option. This works by explicitly keeping track of what treats we've sent up. We then look at the first person and either (1) do nothing, because they're already covered, or (2) send something for them, because they aren't.

```
/* Given we've already picked set of treats chosen in soFar, what is the smallest
 * care package we can send that makes everyone from position nextPerson and
 * onward happy?
 */
HashSet<string> smallestCarePackageRec(const Vector<HashSet<string>>& preferences,
                                       int nextPerson,
                                       const HashSet<string>& soFar) {
    /* Base case: If we've satisfied everyone, return our decisions so far. */
    if (nextPerson == preferences.size()) return soFar;

    /* Base case: If this person is already happy, we don't need to do anything
     * for them.
     */
    for (string treat: soFar) {
        if (preferences[nextPerson].contains(treat)) {
            return smallestCarePackageRec(preferences, nextPerson + 1, soFar);
        }
    }

    /* Recursive case: They're not happy. Try all ways of making them happy and
     * take the best one.
     */
    HashSet<string> best; // Initially empty; this is a sentinel.

    for (string treat: preferences[nextPerson]) {
        auto bestWithThis = smallestCarePackageRec(preferences, nextPerson + 1,
                                                   soFar + treat);
        if (best.isEmpty() || best.size() > bestWithThis.size()) {
            best = bestWithThis;
        }
    }

    return best;
}
HashSet<string> smallestCarePackageFor(const Vector<HashSet<string>>& preferences) {
    return smallestCarePackageRec(preferences, 0, {});
}
```

***Why we asked this question:*** We included this question for a number of reasons. First, we wanted to give you a chance to show us what you'd learned about recursive optimization. Although you haven't seen this exact recursive approach on the assignments, you have seen an optimization problem with a similar setup (Shift Scheduling, where the goal was to find the best set of some type), and we hoped this would be a venue for you to demonstrate what you'd learned in the course of solving that problem.

Second, we wanted to give you an opportunity to demonstrate your skill in translating an abstract description of a recursive solution – here, "pick an unsatisfied person and try all ways of sending them treats" – into an actual recursive tree exploration.

***Common mistakes:*** We saw three general categories of mistakes on this problem. First, there were issues associated with the recursion tree. For example, many solutions worked by picking a person and then trying every way of sending something up for them, but didn't account for the fact that the person might already be covered. This doesn't produce wrong answers, but it does dramatically increase the amount of work that needs to be done. Specifically, the decision tree gets much bigger, both because there are more decisions to make (we have to consider how to cover people who were already covered) and because the branching factor is higher (we explore many options that might not have been explored had we recognized that the person was covered).

Similarly, we saw some solutions that did a double-for loop inside the recursive step, once over people and once over treats, with the idea of trying all ways of picking a person and then all ways of sending something for them. This approach, again, isn't incorrect, but it's highly inefficient. Think of the shape of the recursion tree in this case – this says that, at each point in the tree, you have to pick both a person and a treat, so the branching factor gets much, much higher, dramatically increasing the amount of work that needs to be done.

We also saw some solutions that basically followed the outline of the code above, but which added an extra branch inside the for loop by trying out two options – both giving the person the specific treat and not giving the person that treat. This can potentially introduce some issues downstream. For example, what happens if you always choose not to give treats? This risks not covering everyone and requires extra logic for validation at the end of the recursion. But, more importantly, it introduces a large number of unnecessary branches into the recursion tree. Think of it this way – the question is not "do I give this treat to this person?," but rather "*which* treat do I give this person?" By looking at things the first way, you dramatically increase the number of cases to check, since it turns what would normally be a choice-based recursion ("which one do I pick?") into a subsets-based recursion ("which combination of these do I pick?")

The next category of errors we saw were issues with implementing recursive optimization. Many solutions included a return statement inside of the loop over all treats, along the lines of what's shown here:

```
for (string treat: preferences[person]) {
    /* … mirth and whimsy … */

    return someSolution;
}
```

The problem here is that this return statement prevents the for loop from running multiple iterations. It'll stop as soon as the first one finishes, returning whatever solution was produced there. This is a common error in recursive problem-solving, and it's important one to keep an eye out for down the line.

Many of you realized, correctly, that you need to have some logic to keep track of what the best care package is. Our solution uses the empty set as a sentinel, but other strategies included having a separate variable for the size of the set and initially giving it some large value (say, `INT_MAX`). While many of these approaches work, some solutions attempted to address this issue in a way that didn't work correctly. Some solutions left the set uninitialized and forgot that this would make the set appear to be extremely small. Other solutions tried to give the `HashSet` an illegal value (say, `-1`), which wouldn't compile.

The last major class of error we saw was, unfortunately, not using the strategy required by the problem. Many solutions approached this problem as a pure subsets problem, which we'd said in the problem statement was too slow to be an effective solution. It was difficult for us to award partial credit to those solutions because by going down that route, the solution missed out on several of the specific details we were looking for (finding an uncovered person, tracking the best set across all iterations of the loop, etc.).

***If you had trouble with this problem:*** The steps to take to improve if this problem tripped you up will depend on what specific areas were giving you difficulty.

If you had trouble figuring out the shape of the recursion tree, start off by identifying how you approach these sorts of recursion problems. If you're primarily approaching these problems by asking "is this subsets, combinations, or permutations?," then you've made quite a bit of progress from where we've started (great!), but may need a bit more practice to handle questions that generalize beyond these patterns. Start off by looking back over the decision trees for those patterns. What do those trees look like? How do we explore those trees in code? Can you see how the specific code patterns you're used to for those patterns follow from the tree? Once you've done that, take a second stab at this problem. Draw the recursion tree – or at least, a small piece of the tree. If you're able to do that, great! If not, come talk to us in office hours or in the CLaIR. Once you have the tree, try writing the code for this one a second time.

If you had an issue with returning too early, or you accidentally mishandled the sentinel value that occurs inside the loop, we recommend the following. As you're writing out a recursive function, there's a good deal of work to do to figure out what the recursive calls should be simply to enumerate all the options. But, fundamentally, these optimization-type problems boil down to "call some function lots of times and return the best one." Pretend, for a minute, that you're not writing a recursive function and that the call you're making is to some totally unrelated function. Then ask: if I wanted to capture the best value returned by this function, what would I do? Decoupling the recursion bit from the optimization bit might make things a lot easier to solve.

## Problem Four: Recursion II (8 Points)

This solution is based on the idea that each shift either

- doesn't get assigned, or
- does get assigned, and gets assigned to one of the workers.

We therefore go one shift at a time, trying each option and taking whichever gives us the best results.

```
/* Given a partial set of shift assignments, what's the maximum value we can
 * produce by extending that partial set of assignments?
 */
HashMap<string, HashSet<Shift>>
bestScheduleForRec(const HashSet<Shift>& remaining,
                   const HashMap<string, int>& hoursFree,
                   const HashMap<string, HashSet<Shift>>& soFar) {
    /* Base case: If all shifts are assigned, we're committed to what we have. */
    if (remaining.isEmpty()) {
        return soFar;
    }

    /* Recursive case: process some unchosen shift. */
    auto curr = remaining.first();

    /* One option is to not use this shift at all. */
    auto best = bestScheduleForRec(remaining - curr, hoursFree, soFar);

    /* Another is to give this to someone. */
    for (string person: hoursFree) {
        if (hoursFree[person] >= lengthOf(curr) &&
            isCompatibleWith(soFar[person], curr)) {

            /* Adjust the remaining hours and assigned shifts to include this. */
            auto hoursWith  = hoursFree;
            auto shiftsWith = soFar;

            hoursWith[person]  -= lengthOf(curr);
            shiftsWith[person] += curr;

            /* See if this is better than what we have so far. */
            auto bestWith = bestScheduleForRec(remaining - curr,
                                               hoursWith, shiftsWith);
            if (valueOf(bestWith) > valueOf(best)) {
                best = bestWith;
            }
        }
    }

    return best;
}
```

/* Continued on the next page… */

```
/* Given a set of used shifts, can we add this new shift in? */
bool isCompatibleWith(const HashSet<Shift>& used, const Shift& shift) {
    for (Shift s: used) {
        if (overlapsWith(s, shift)) {
            return false;
        }
    }
    return true;
}


/* How much total value is produced by this set of shifts? */
int valueOf(const HashMap<string, HashSet<Shift>>& shifts) {
    int result = 0;
    for (string person: shifts) {
        for (Shift shift: shifts[person]) {
            result += valueOf(shift);
        }
    }
    return result;
}

HashMap<string, HashSet<Shift>>
bestScheduleFor(const HashSet<Shift>& shifts,
                const HashMap<string, int>& hoursFree) {
    return bestScheduleForRec(shifts, hoursFree, {});
}
```

***Why we asked this question:*** We included this problem for a number of reasons. First, the particular recursive pattern here – that each shift either (1) gets discarded entirely or (2) goes to exactly one person – is something that's related to what you've seen on the programming assignments but which isn't an exact match for anything you've seen. We thought this would be a great way for you to show us what you've learned about recursive problem-solving in the course of working through the coding assignments from this quarter. We also thought that this problem would be a great way to let you show us what you've learned about recursive optimization.

***Common mistakes:*** There were several classes of errors that we saw on this problem. Let's address each one in turn.

The first class of errors we saw on this problem were errors involving the recursion tree. For example, many approaches had this general shape:

```
Shift toAssign = /* … pick a shift … */
for (string person: people) {
    /* try giving this person the shift with one recursive call. */
    /* try not giving this person the shift with one recursive call. */
}
```

This approach will indeed try out all possible combinations of shifts, but it does so extremely inefficiently. For example, notice that the recursive call for "don't give the first person the shift" is exactly the same as the recursive call for "don't give the second person the shift," which is the exact same call as "don't give the third person the shift," etc. In each case, the shift isn't assigned to anyone. This makes the recursion tree *significantly* bigger than it needs to be, which would make this implementation prohibitively slow.

Another error we saw was writing code that always tried to give a shift to someone, regardless of whether it was worthwhile to assign. That is, the code never considered the possibility that it might be optimal to not assign the shift to anyone. This will cause the program to sometimes return the wrong schedule. For example, consider a case where the first shift in the set of possibilities is very long and has a low value. Not giving anyone that shift is likely a better option than trying to have someone work through it.

The next class of errors we saw had to do with the implementation of the recursive optimization. Many pieces of code contained errors like these, which we assume were due to trying to modify the code from the regular Shift Scheduling problem. For example, we saw many solutions that looked like this:

```
Shift toAssign = /* … pick a shift … */
for (string person: people) {
    /* try giving this person the shift with one recursive call. */
    auto with = /* … */

    /* try not giving this person the shift with one recursive call. */
    auto without = /* … */


    /* This is the "ternary conditional operator" ?:. The expression
     * expr? x : y means "if expr is true, then produce x, and otherwise
     * produce y.
     */
    return valueOf(with) > valueOf(without)? with : without;
}
```

Notice that this code will never execute more than one iteration of the for loop, since it always returns a value at the end of the first iteration and therefore can't move on to the next.

At a more detail-oriented level, we saw many solutions that would make recursive calls like these ones:

```
auto result = bestScheduleRec(/* … */, schedule[person] += shift);
```

Here, the code passes in `schedule[person] += shift` rather than `schedule[person] + shift` into the function. The use of the += operator here is legal, and means "modify `schedule[person]` by adding shift into it, then return the resulting `HashSet`," and makes a permanent change to `schedule[person]`. This can sometimes cause problems as each loop iteration accidentally picks up changes that were intended for other loop iterations. (Also, while this wasn't always the case, this was often a type error, since the intent was to pass down another `HashMap<string, HashSet<string>>` to the recursive call, but the value of the expression `schedule[person] += shift` is a `HashSet<string>`.)

By far the most common mistake on this problem was to take the code from the Shift Scheduling problem from the Recursion assignment and to try to tweak it to get it to fit this problem. Although there are indeed many similarities between that problem and this one, the core recursive insight is different and, accordingly, the resulting code bears only a slight resemblance to the source material. Here are a few of the differences between the problems:

- Shift Scheduling follows a combinations/subsets-style recursion: each shift is either chosen or excluded. This problem has more of an assignment-type recursion: we need to decide who to give a particular shift to.
- Because Shift Scheduling only has two recursive branches, the logic to find the best option can be as simple as an if statement or a return statement with a `?:` conditional. This problem requires looping over lots of options, plus an option that is different from all of them, and therefore requires different code to find the maximum value.

It is indeed possible to solve this problem by starting with that code and making changes to it, but it requires a good deal of attention to detail to make sure that you don't introduce the sorts of errors described above.

***If you had trouble with this problem:*** As with most recursion problems, if you found this one tricky, it's worth taking stock of where the issue was.

The first step in solving problems like these, typically, is to think about the recursion tree. You'll want to ponder questions like "what sorts of decisions do I need to make here?" and "in what order do I need to make those decisions?" It often helps to do what we were doing in lecture when we talked about subsets, permutations, combinations, etc., where we wrote out a list of objects and then went one at a time, asking questions like "do we want to pick this one?" or "which one should we pick next?" That can help you settle on a strategy.

Once you've done that, the next question is to think about what information you need to keep track of in order to implement the strategy you've described. Do you need to remember all the decisions that you've made so far? Do you need to remember what your next decision is? Does order matter? Does order not matter? These decisions will help you choose your data structures and the arguments to your functions.

From there, it's time to start exploring the whole space. Write out code to generate every possible solution – not necessarily to return the best solution, just code that prints out all the options. Doing that might expose that your strategy doesn't work, or that you need some extra arguments, etc. But that's just you being an engineer! It's part trial-and-error, part informed guesses, and part drawing on your experiences.

Only after you have those steps working should you start thinking about how to do the optimization bit. And to do that, you'll want to look at the code you have. You might have a loop over a bunch of options, in which case your optimization logic will likely be just keeping track of the best option returned by any of those calls, potentially with some sort of sentinel value. This is just like what you did in CS106A – it's just like finding the maximum value in an array! Or it might be that there really are only a fixed number of recursive calls, in which case you can just name all the values and compare them at the end.

This is something you'll get with practice. The more times you work through these sorts of problems, the more comfortable you'll be tackling new ones. So look back at Section Handout 3, Section Handout 4, and the textbook chapters on recursion and recursive backtracking. Find some problems that look interesting and work through them. Try using the workflow above when doing so – does that make things easier?

And, of course, feel free to stop by the LaIR, the CLaIR, or office hours with questions!