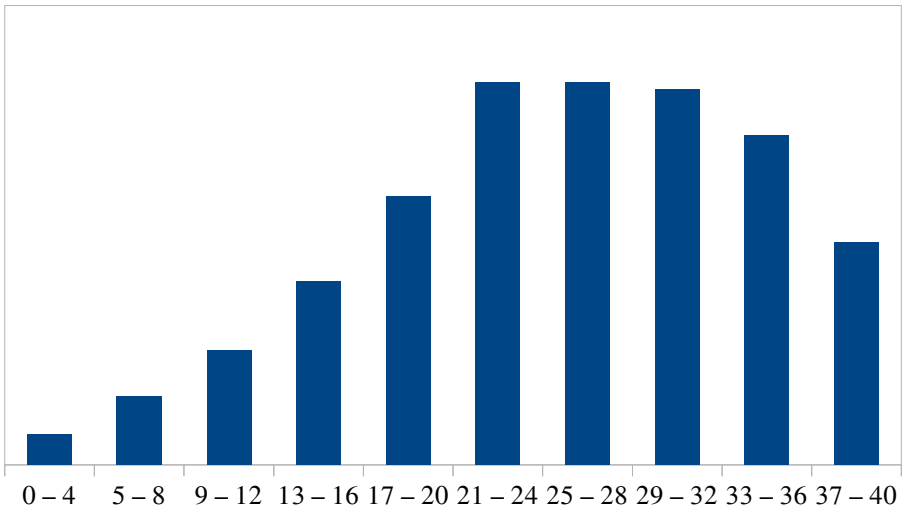
CS106B Midterm Exam Solutions

Here's the grade distribution from the midterm:



To help you get a better sense of how you did on the exam, here are the quintile markers:

80th Percentile: 33 / 40 (83%)
60th Percentile: 28 / 40 (70%)
40th Percentile: 23 / 40 (58%)
20th Percentile: 18 / 40 (45%)

As a reminder, your grade in this class is computed as follows:

35%: Programming Assignments

25%: Midterm Exam

35%: Final Exam

5%: Section Participation

With three of the seven assignments not yet turned in, the final exam coming up, and a few sections remaining, over half of your overall grade in this course is still completely under your control. Regardless of how this exam went for you, you still have plenty of time to patch up any weak spots in your understanding and to improve by the end of the quarter. This means that, regardless of the outcome of this exam, you have a lot of control over how you proceed from here.

Don't think about your performance on this exam as the shape of things to come. Instead, think of it as a (noisy) data point about how you're doing right now. We've seen people bounce back from earning fewer than half the possible points on the midterm to earning perfect scores on the final exam. So spend the time you need to figure out what you're doing that's working, what you're doing that isn't, and what changes you want to make for the last three weeks of the quarter.

Please feel free to reach out to us if you need any help! Your section leader is a great resource; be sure to ask them for input about how to improve. And feel free to come talk to Kate or Keith in their office hours! There's also the CLaIR for any conceptual questions you might have, and you're welcome to drop in with questions on any of the course topics.

Problem One: Container Classes

Social Network Strength, Part I

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.

```
Set<string> allPeopleIn(const Map<string, Set<string>>& network) {
    Set<string> result;
    for (string person: network) {
        result += person;
    }
    return result;
}
Set<string> kCoreOf(const Map<string, Set<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.

```
Set<string> kCoreOf(const Map<string, Set<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. */
    Set<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.

```
Set<string> kCoreOfRec(const Map<string, Set<string>>& network, int k,
                       const Set<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);
}

Set<string> kCoreOf(const Map<string, Set<string>>& network, int k) {
    Set<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:

```
Set<string> kCoreOf(const Map<string, Set<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.
     */
    Map<string, Set<string>> coreNetwork;
    Set<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, `Map`s and `Set`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 `Set`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 `Map` or `Set` using a range-based `for` loop and you remove an element from the `Map` or `Set`, 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 `Map`s and `Set`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 ADTs. There are some great exercises in there, and blocking out the time to work on them can really make a difference.

Problem Two: Big-O Notation

Social Network Strength, Part II

Suppose there's a social network whose current value with its current number of users is $10,000,000.

i. **(1 Point)** *Sarnoff's Law* states that the value of a network is $O(n)$, where n is the number of users on the network. Assuming Sarnoff's law is correct, estimate how much the social network needs to grow to have value $160,000,000. Justify your answer.

The network would need to grow about sixteen times bigger. Since the value of the network scales linearly, to get a 16× increase in value, you need roughly a 16× increase in size.

ii. **(1 Point)** *Metcalfe's Law* states that the value of a network is $O(n^2)$, where n is the number of users on the network. Assuming Metcalfe's law is correct, estimate how much the social network needs to grow to have value $160,000,000. Justify your answer.

The network would need to grow about four times bigger. Since the value of the network scales quadratically, to get a 16× increase in value, you need roughly a 4× increase in size ($(4n)^2 = 16n^2$).

iii. **(1 Point)** *Reed's Law* states that the value of a network is $O(2^n)$, where n is the number of users on the network. Assuming Reed's law is correct, estimate how much the social network needs to grow to have value $160,000,000. Justify your answer.

Four more people need to join the network. Note that $2^{n+4} = 2^n \cdot 2^4 = 16 \cdot 2^n$, so adding four more people increases the value of the network by the needed factor of sixteen.

Why we asked this question: This question was designed to let you show us what you'd learned about using big-O notation to predict rates of growth. We specifically included these three growth rates ($O(n)$, $O(n^2)$, and $O(2^n)$) because they're common in algorithmic analysis.

Common mistakes: By far the most common mistake on this problem was to say, for the case where the network's value is $O(2^n)$, that the network needs to grow four times bigger. A common justification for this answer was the following: since $2^4 = 16$ and we want the network to grow by a factor of sixteen, we need to multiply the size of the input by four. It is indeed true that $2^4 = 16$, but that doesn't necessarily mean that we need to scale the input by a factor of four.

To see why this doesn't work, let's begin with a little math. Let's assume the value of the network is exactly $\$2^n$, where n is the number of people on the network, and assume $n = 10$. The network is therefore now worth exactly $1,024. If we 4× the number of people on the network, it's now worth a whopping $\$2^{40} = \$1,099,511,627,776$. That's an *astonishing* amount of money, way more than a 16× increase! In fact, it's now worth more than *a billion times* as much as it used to be!

The math above shows why scaling by a factor of four *doesn't* work. And the math given in the answer to part (iii) explains that adding in four people *does* work. But that doesn't explain how to arrive at the answer, and so let's walk through a few perspectives.

First, an intuitive explanation: if some quantity is $O(2^n)$, it means that every time you add one more item in, the quantity doubles. That's because 2^n means "multiply two by itself n times," so each addition to n multiplies in another factor of two. Therefore, we can ask – how many times do we have to double $10M to get to $160M? The answer is four times ($10M, then $20M, then $40M, then $80M, then $160M). So we need to double the input four times, and each time a new person joins the value doubles, which means we need exactly four more people to join.

Second, a mathematical explanation. Let's imagine that our value is 2^n. We want to increase the value of our network to 16× where it started. Let's have m be the new size of the network. We then want to solve

$$2^m = 16 \cdot 2^n.$$

A little algebra tells us that

$$2^m = 2^4 \cdot 2^n$$
$$2^m = 2^{n+4}$$
$$m = n + 4$$

and from this we see that we need m, the new number of people, to be $n + 4$.

You might then ask a follow-up question: why is it, then, that when we want to get the quadratic function to 16× in value, we can just say that $4^2 = 16$ and therefore scale by a factor of four? Let's do a similar analysis to what we just did, assuming our network's value is $c \cdot n^2$. We set things up as

$$m^2 = 16 \cdot n^2$$
$$m^2 = 16n^2$$
$$m = 4n$$

and we have that our new network size must go up by a factor of four.

If you had trouble with this problem: There are two important points to keep an eye out for when looking over this problem. First, there's the intuitive, gut feeling you should have with these growth rates. If you see $O(n)$, think "doubling the input doubles the output." If you see $O(n^2)$, think "doubling inputs quadruples outputs." And if you see $O(2^n)$, think "increasing the input by a single unit doubles the output." Those intuitions are powerful ones to have going forward and are great sanity checks on answers to questions like these.

Second, I would recommend looking over the sort of math that we did above here and to make sure you can follow these sorts of derivations. Many of the people who ran into trouble on this problem had developed shorthands for reasoning about big-O notation that were reasonable for some cases (polynomials like n, n^2, n^3, etc.) but which don't translate to other functions. There's nothing wrong *per se* with having these shorthands, but it's important to make sure that you know where they come from and how far those shorthands can be safely extended.

Problem Three: Recursion, Part I

Cosmic Care Packages

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.

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

    /* Track the best option so far. */
    Set<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<Set<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?
 */
Set<string> smallestCarePackageRec(const Vector<Set<string>>& preferences,
                                   int nextPerson,
                                   const Set<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.
     */
    Set<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;
}
Set<string> smallestCarePackageFor(const Vector<Set<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 (Riding Circuit, where you have to choose one of many options and take the best out of all of them), 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. You have many similar examples to draw from here. This problem has echoes of Doctors Without Orders (desserts are kinda sorta ish like patients) and Disaster Planning (the idea of covering one object with another), and the question was how to draw on those inspirations to put things together.

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 recursion 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 `Set` 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, Part II

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?
 */
Map<string, Set<Shift>> bestScheduleForRec(const Set<Shift>& remaining,
                                           const Map<string, int>& hoursFree,
                                           const Map<string, Set<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 Set<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 Map<string, Set<Shift>>& shifts) {
    int result = 0;
    for (string person: shifts) {
        for (Shift shift: shifts[person]) {
            result += valueOf(shift);
        }
    }
    return result;
}
Map<string, Set<Shift>> bestScheduleFor(const Set<Shift>& shifts,
                                        const Map<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 = /* … */

    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 `Set`," 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 `Map<string, Set<string>>` to the recursive call, but the value of the expression `schedule[person] += shift` is a `Set<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!

Problem Five: Recursion, Part III

SlimCity

There are many ways to solve this problem. This first solution works by noting that some building has to go in the first position. We can therefore look at that spot and then try all possible buildings that can fit there, then repeat this process for the second slot, the third, the fourth, etc.

The challenge is to make sure we don't place down any "silly" configurations that can't possibly work. To do this, every time we consider a building, we'll make sure that it's close enough to the buildings before it that it's not completely implausible to place it down. Because of the symmetries in the distances, if we check whether each newly-added building meets the distance requirements of its predecessors, we'll ensure that, overall, everything is placed legally.

```
/* Can the given building be placed at the indicated spot without violating any
 * distance constraints?
 */
bool canPlaceHere(const Vector<string>& cityBlocks,
                  const Map<string, Map<string, int>>& maxDistances,
                  const string& building,
                  int index) {
    /* Check if any prior buildings conflict. */
    for (int i = 0; i < index; i++) {
        /* The distance between these buildings is index - i. */
        if (index - i > maxDistances[cityBlocks[i]][building]) {
            return false;
        }
    }
    return true;
}
bool canPlaceCompactly(Vector<string>& cityBlocks,
                       const Set<string>& buildings,
                       const Map<string, Map<string, int>>& maxDistances) {
    /* Base case: If there are no unplaced buildings, then we're in good shape! */
    if (buildings.isEmpty()) return true;

    /* Recursive case: Some building has to come next. Which one should it be? */
    int index = cityBlocks.size() - buildings.size();

    for (string option: buildings) {
        /* If placing this building wouldn't violate any distance requirements,
         * we can try it out to see what happens.
         */
        if (canPlaceHere(cityBlocks, maxDistances, option, index)) {
            cityBlocks[index] = option;
            if (canPlaceCompactly(cityBlocks, buildings - option, maxDistances)) {
                return true;
            }
        }
    }

    /* Rend our garments, wail, and gnash our teeth. It ain't happening. */
    return false;
}
```

This second solution is based on the insight that if we grab some unplaced building, we know that it has to go somewhere, but we can't say for certain where that is. We could therefore try placing it in each possible slot, seeing if any of them (1) don't immediately violate any distance constraints and (2) can be extended into a full solution. One way of doing this, which relies on the fact that the empty string can't be the name of any building, is shown here:

```
/* Can the given building be placed at the indicated spot without violating any
 * distance constraints?
 */
bool canPlaceAt(const Vector<string>& cityBlocks,
                const string& toPlace,
                int index,
                const Map<string, Map<string, int>>& maxDistances) {
   for (int i = 0; i < cityBlocks.size(); i++) {
      /* If there's something here, confirm there are no conflicts with it. */
      if (i != index && cityBlocks[i] != "" &&
          abs(index - i) > maxDistances[toPlace][cityBlocks[i]]) {
         return false;
      }
   }
   return true;
}

bool canPlaceCompactly(Vector<string>& cityBlocks,
                       const Set<string>& buildings,
                       const Map<string, Map<string, int>>& maxDistances) {
   /* Base case: If there are no unplaced buildings, then we're in good shape! */
   if (buildings.isEmpty()) return true;

   /* Recursive case: Grab some building and try putting it in each spot where
    * it doesn't cause a conflict.
    */
   string curr = buildings.first();
   for (int i = 0; i < cityBlocks.size(); i++) {
      /* Try placing it here if the city block isn't already in use and doing so
       * doesn't violate any constraints.
       */
      if (cityBlocks[i] == "" && canPlaceAt(cityBlocks, curr, i, maxDistances)) {
         cityBlocks[i] = curr;

         /* If this can be extended, great! We're done. If not, we need to
          * pretend we didn't put anything here at all.
          */
         if (canPlaceCompactly(cityBlocks, buildings - curr, maxDistances)) {
            return true;
         }

         /* Oops. Didn't work. */
         cityBlocks[i] = "";
      }
   }

   /* Abandon hope, all ye who enter here. */
   return false;
}
```

Why we asked this question: We picked this question for a few reasons. First, we wanted to include at least one recursive backtracking problem on this exam so that you could show us what you'd learned from working through the Recursion to the Rescue assignment. That included both the patterns involved in trying out lots of options to see if any of them work and the logic to maintain the outparameter to the function.

Second, we wanted to give you a problem, like the Dense Crosswords example from lecture, where it's important to check at each step that the partial steps you're taking are indeed feasible. You might recognize that this problem is essentially a permutations problem – try all ways of arranging the building and see which one comes out best – but most of those permutations contain obvious violations of the constraints. We hoped you'd recognize that it would not be a good idea to place a building anywhere that would immediately cause a conflict.

Common mistakes: Some of the errors we saw on this problem were simple logic errors in confirming that the buildings were sufficiently close to one another. For example, some solutions had simple off-by-one errors (for example, checking whether the distances were *strictly less than* the maximum distance rather than *no more* than the max distance, or whether the distances were *exactly equal to* the maximum distance). Other solutions forgot to account for the case that some city blocks would have no buildings in them, and therefore that some extra logic was required to handle those empty slots. Some solutions forgot to take the absolute value of the difference in indices between buildings. Some solutions had stray errors like misplaced return statements that caused the right logic to cut off too early. Others tried using integers as keys in the `maxDistances` map, usually due to mixing up indices within the array and the strings that happened to be at those indices.

Other errors we saw were more specific to the recursive structure of the problem. Many solutions deferred any checking of the feasibility of the solution to the very end of the search, meaning that the code would spend lots of time focused on solutions that couldn't possibly work. Other solutions would sometimes place buildings on top of other buildings, or would leave blocks empty until the very end.

We saw many bugs that occurred in the context of managing the `cityBlocks` argument. That argument is sized to the number of city blocks and is initially filled with empty strings, so there's no need to add elements to it or remove elements from it. Instead, you just need to overwrite the existing elements there. Many solutions would use `add`/`remove`, which would resize the `Vector` and shift other buildings that used to be in safe locations into positions where they were too far away from the other buildings they needed to be close to.

Additionally, we saw many solutions that would make changes to `cityBlocks` before trying out some recursive call, but which forgot to undo the changes made to that `Vector` before making the next call. This had the effect of "polluting" the city blocks with entries corresponding to "we tried putting the building here once, and it didn't work" that were indistinguishable from "there is actually a building here."

If you had trouble with this problem: As with the other recursion problems here, it's worth taking stock of what aspect of this question you found tricky.

For starters, did you recognize the general recursive strategy to use here? If you were able to identify it as a permutations problem, or you recognized that you could go one slot at a time or one building at a time, great! You're on the right track. If that insight didn't click, that's probably what you want to focus a bit more on. Take some of the recursion problems from this quarter – whether from the section handouts, the practice exams, the assignments, or the lectures – and see whether, for each, you can describe what the recursive insight is. For recursion problems involving searching a recursion tree, draw out that tree, or at least make sure you know what it looks like.

If you did have a good handle on the recursive strategy but messed up the outparameter management, there are a couple of approaches you can try to improve going forward. One option would be to make sure you have a clear and precise understanding of what the signature of a function promises. For example, if a function says "I'll do some work and give you an answer, and if the answer is "yes," I'll fill in the outparameter," look back at your code. What do you do to the outparameter if the answer is "yes?" What do you do if the answer is "no?" The answer to that second one is key – you want to make sure that whatever you do is consistent with how your function believes will happen when it makes its recursive call.

Another option would be to think about treating outparameters like they've been loaned to you. If your function returns false, you should reset the outparameter so that it looks exactly the same as when it was handed into your function. That way, you can be assured that you aren't accidentally copying state across from one function call to another. On the other hand, if the function returns true, then you can edit it however you see fit, since you've promised to fill it in with some extra information!

Finally, if you got tripped up by the distance measurement bit, it could be that it was just plain human error (as in, "oops, I did write that, and that's clearly wrong"). That's something that just getting more general coding practice will help with. On the other hand, if you found that section particularly difficult, there are a few things you can do in the future. For example, this is a problem that is much, much easier to do with some pictures in front of you. That might help you spot, for example, that you need to handle empty slots in the city blocks and that the sign of the distances between buildings is significant.