CS106B Handout 15S

Winter 2017 February 13, 2017

Practice CS106B Midterm Exam Solutions

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. You're encouraged to ask questions about the exam or its solutions. We're happy to help out!

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 had three hours to complete this whole exam and had to use a pencil and paper. We had 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: Container Classes

Shrinkable Word Ladders

```
bool isShrinkable(const string& word, const Lexicon& english,
                  Vector<string>& shrinkingSequence) {

    /* Confirm that the input is a word; if not, it can't be shrinkable! */
    if (!english.contains(word)) return false;

    /* Remember the words we've used in shrinking sequences so far */
    Lexicon usedWords;
    usedWords.add(word);

    /* Set up our worklist. */
    Queue<Vector<string>> worklist;
    worklist.enqueue({ word });

    /* Search for a shrinking sequence! */
    while (!worklist.isEmpty()) {
        Vector<string> sequence = worklist.dequeue();
        string lastWord = sequence[sequence.size() - 1];

        /* See if we're done! */
        if (lastWord.length() == 1) {
            shrinkingSequence = sequence;
            return true;
        }

        /* Try all ways of shrinking this word. */
        for (int i = 0; i < lastWord.length(); i++) {
            string nextWord = lastWord.substr(0, i) + lastWord.substr(i + 1);

            /* If what we are left with is a word and we haven't used it yet,
             * form the next ladder from it.
             */
            if (english.contains(nextWord) && !usedWords.contains(nextWord)) {
                usedWords.add(nextWord);
                Vector<string> nextSequence = sequence;
                nextSequence += nextWord;
                worklist.enqueue(nextSequence);
            }
         }
     }
     return false;
}
```

Why we asked this question: This question was designed to make sure that you had internalized how the breadth-first search algorithm works. We also wanted to make sure that you were fluent with the fundamental container types represented here – queues, vectors, lexicons, and strings.

We also thought it would be interesting to show you that, as usual, there are many, many different ways that you can solve a problem. Here, we've opted to go for an iterative breadth-first search for shrinkable words rather than using the recursive formulation from class.

Problem Two: Recursive Enumeration (8 Points)

Splitting the Bill

```
/**
 * Lists off all ways that the set of people can pay a certain total, assuming
 * that some number of people have already committed to a given set of payments.
 *
 * @param total The total amount to pay.
 * @param people Who needs to pay.
 * @param payments The payments that have been set up so far.
 */
void listPossiblePaymentsRec(int total, const Set<string>& people,
                             const Map<string, int>& payments) {
    /* Base case: if there's one person left, they have to pay the whole bill. */
    if (people.size() == 1) {
        auto finalPayments = payments;
        finalPayments[people.first()] = total;
        cout << finalPayments << endl;
    }
    /* Recursive case: The first person has to pay some amount between 0 and the
     * total amount. Try all of those possibilities.
     */
    else {
        for (int payment = 0; payment <= total; payment++) {
            auto updatedPayments = payments;
            updatedPayments[people.first()] = payment;
            listPossiblePaymentsRec(total - payment, people - people.first(),
                                    updatedPayments);
        }
    }
}

void listPossiblePayments(int total, const Set<string>& people) {
    listPossiblePaymentsRec(total, people, {});
}
```

Why we asked this question: This question was designed to see whether you were comfortable using recursion to list off all objects of a certain type (here, ways of dividing money across a group of people) even if you hadn't previously encountered the particular structure before.

The insight that we used in our solution is that the first person has to pay some amount of money. We can't say for certain how much it will be, but we know that it's going to have to be some amount of money that's between zero and the full total. We can then try out every possible way of having them pay that amount of money, which always leaves the remaining people to split up the part of the bill that the first person hasn't paid.

The code above essentially represents what you get if you explore all the possible options using the decision tree idea that we discussed in class. We keep track of the present (how much still needs to get paid) and the past (who's agreed to pay which amounts) and then try each possible future step.

Our code makes generous use of the `auto` keyword in C++, which was discussed in Section Solutions 3. The `auto` keyword says "declare a variable, and give it the type of whatever is on the right-hand side of the equals sign." Here, we've used this to avoid writing out in longhand the full name of the map type that we're using. You're welcome to use this on the exam and in the assignments if you'd like – it's a *really* nice language feature!

Problem Three: Recursive Optimization (8 Points)

Drills, Baby, Drills!

```
/**
 * Given two drill routes, returns the amount of time necessary for the longer
 * of the two routes to complete.
 *
 * @param routes The drill routes in question.
 * @return The completion time for the slower of the two routes.
 */
double timeFor(const DrillRoutes& routes) {
    return max(drillRouteLength(routes.forOne), drillRouteLength(routes.forTwo));
}

/**
 * Given a list of sites to be drilled and a partial assignment of which robot
 * should get which drill sites, returns the best pair of drill routes.
 *
 * @param sites The sites to drill
 * @param index The index into the drill site at which to begin
 * @param forOne The drill sites assigned to the first robot
 * @param forTwo The drill sites assigned to the second robot
 * @return The best pair of drill routes to use.
 */
DrillRoutes recBestDoubleDrillRoute(const Vector<DrillSite>& sites, int index,
                                    const Vector<DrillSite>& forOne,
                                    const Vector<DrillSite>& forTwo) {
    /* Base case: If all sites have been assigned to a robot, then the best we
     * can do is to have each robot do the best it can with its route.
     */
    if (index == sites.size()) {
        return { bestDrillRouteFor(forOne), bestDrillRouteFor(forTwo) };
    }

    /* Recursive case: The next drill site needs to get assigned to one of the two
     * robots. Try assigning it to each and see which option works out better.
     */
    auto givenToOne = forOne;
    givenToOne += sites[index];

    auto givenToTwo = forTwo;
    givenToTwo += sites[index];

    auto bestOne = recBestDoubleDrillRoute(sites, index + 1, givenToOne, forTwo);
    auto bestTwo = recBestDoubleDrillRoute(sites, index + 1, forOne, givenToTwo);

    if (timeFor(bestOne) < timeFor(bestTwo)) {
        return bestOne;
    } else {
        return bestTwo;
    }
}

DrillRoutes bestDoubleDrillRouteFor(const Vector<DrillSite>& sites) {
    return recBestDoubleDrillRoute(sites, 0, {}, {});
}
```

Why we asked this question: Although this problem is closely related to the drill problem from Assignment 3, you'll notice that it is ***not*** a question about generating permutations. In fact, the key idea behind the solution we included here is that we already have a function that takes in a list of points and finds the best way to drill them, and so the fundamental question we need to answer is how to figure out how to split the points between the two robots. This type of problem is called a ***partitioning problem*** or an ***assignment problem***: the goal is to take a group of things (here, points) and distribute them into buckets (here, robots). In some ways, this is conceptually similar to the preceding problem (partition the dollars to people) or the Doctors Without Orders problem (assign each patient to a doctor). In fact, it's probably worth adding assignments to your repertoire of structures you know how to generate recursively (along with subsets, permutations, and combinations), since it shows up every now and then.

We liked this particular problem because it's also very closely related to the subsets problem. If you choose one subset of the points to give to the first robot, then the second robot essentially has to take the rest of the points just by default. So even if you didn't think of this problem as a partition or an assignment, you could still make progress on it.

So what's the takeaway? Well, a few things. First, if you look at a problem that's related to something you've seen before, take a minute or two to pause and think about whether the problem actually has the same structure as the one you've seen before. As you saw with the shrinkable word ladders problem, sometimes a new take on an old problem leads to a totally different solution route. Second, it might be a good idea to become comfortable with the idea of generating partitions and assignments, since it comes up a good amount in optimization problems like these. Finally, make sure you're comfortable with generating subsets, since subsets and their variations come up all the time.

Problem Four: Recursive Backtracking (8 Points)

A Team-Building Exercise

```
/**
 * Given a team and a new person, returns whether the new person would work well
 * with everyone on that team.
 *
 * @param team The existing team
 * @param newcomer The new person interested in joining.
 * @return Whether that person would work well with the team.
 */
bool areHarmonious(const Set<string>& team, const string& newcomer) {
    for (string person: team) {
        if (!areHappyTogether(person, newcomer)) {
            return false;
        }
    }
    return true;
}
bool canSplitMerrilyRec(const Set<string>& people, int teamSize,
                        Vector<Set<string>>& teams) {
    /* Base case: if everyone is assigned to a team, we're done, because all the
     * teams collectively have the capacity to hold everyone.
     */
    if (people.isEmpty()) {
        return true;
    }

    /* Recursive case: The next person needs to be assigned somewhere. Try all
     * existing teams to see if they have space and would work well with them.
     */
    string next = people.first();
    for (int i = 0; i < teams.size(); i++) {
        if (teams[i].size() < teamSize && areHarmonious(teams[i], next)) {
            teams[i] += next;
            if (canSplitMerrilyRec(people - next, teamSize, teams)) {
                return true;
            }
            teams[i] -= next;
        }
    }
    return false;
}
bool canSplitMerrily(const Set<string>& people, int teamSize,
                     Vector<Set<string>>& teams) {
    /* Confirm that the people can mathematically be divided into groups. */
    if (people.size() % teamSize != 0) return false;

    /* Set up a bunch of initially empty teams. */
    for (int i = 0; i < people.size() / teamSize; i++) {
        teams += {};
    }
    return canSplitMerrilyRec(people, teamSize, teams);
}
```

Why we asked this question: Continuing a theme from this practice exam, this problem is a great example of a partitioning problem. We hoped that the key recursive insight – that everyone has to be assigned somewhere, and so you can just pick a person and try putting them everywhere – wouldn't be too tricky to come up with given your experience with the Doctors Without Orders problem.

We also included this problem because it's a nice place to use the mark/unmark pattern talked about in the textbook. Notice that our solution works by tentatively adding a person to a group, trying out what happens if we do that, then removing them from the group if it fails. This approach, which we didn't explicitly talk about in class, is another route you can take to solve backtracking problems that require you to also output an answer.

Problem Five: Big-O and Efficiency (8 Points)

The "Science" Part of Computer Science

i. **(3 Points)** Based on the above data, do you have enough information to make an educated guess about which algorithm runs in time O(n), which runs in time O(n^2), and which runs in time O(n^3)? If so, explain which algorithm has which runtime and why you have enough information to make the guess. If not, explain why you don't have enough information to make an educated guess.

With only single data points and with a comparably low value of n, it's hard to make an educated guess about the runtimes of each of these pieces of code, since big-O notation talks about long-term growth rates and we don't have the runtime over multiple different inputs.

ii. **(3 Points)** Based on the above data, do you have enough information to make an educated guess about which algorithm runs in time O(n), which runs in time O(n^2), and which runs in time O(n^3)? If so, explain which algorithm has which runtime and why you have enough information to make the guess. If not, explain why you don't have enough information to make an educated guess.

While we can't say anything for certain, we do have enough information here to make an educated guess.

Looking at the runtime for Package A, we can see that the runtime roughly doubles every time in the input size doubles. For example, going from 100 elements to 200 elements jumps the runtime by just about a factor of two, going from there to 400 elements also jumps the runtime by approximately a factor of two, and from there to 800 elements is another jump of roughly a factor of two. As a result, we can make an educated guess that the runtime of Package A is O(n), since the runtime scales proportionally to the input size.

Looking at the runtime for package B, we can see that for sufficiently large inputs the runtime roughly increases by a factor of 8 as the input size doubles. For example, going from 200 elements to 400 elements produces a roughly eightfold increase in the runtime, as does the jump from 400 elements to 800 elements. We can also see this in the jump from 300 to 600 elements. This means that the runtime grows about eight times faster than the input size, which suggests that the runtime is O(n^3).

We could just say that the runtime of Package C is going to be O(n^2) at this point purely by a process of elimination, but it helps to double-check that the runtime seems to scale by a factor of four every time the input size doubles. Notice that going from 200 to 400 to 800 elements represents the input size roughly quadrupling.

iii. **(2 Points)** Based on the data you have available to you, if you had to make an educated guess about which software package you'd recommend to someone with a data set of size 5,000, which one would you recommend? Why?

I'd recommend Package A. At 800 elements we see that Package A is roughly comparable to the other packages, and since its runtime of O(n) is better than the runtimes of the other two packages, we'd expect that as we extrapolate outward the runtime of Package A would grow much more slowly than the runtimes of the other two packages, so it would almost certainly be the fastest package to use.

Why we asked this question: We've talked about many different aspects of big-O notation, such as how to look at a piece of code and try to determine its computational complexity, how to use big-O notation to predict runtimes, and the limits of what big-O notation does and does not tell you. We'd taught those skills *deductively* by showing you how to mathematically work out runtimes and extrapolate them using quantitative reasoning. This problem asks you to apply those skills *inductively* by trying to work out what sorts of processes would likely account for the behavior that you're seeing. Our hope was that by asking you to reason in the reverse direction from what we'd typically asked, you'd demonstrate what you'd learned in a practical and more uncertain setting.

The major skills here are trying to estimate runtimes by looking at plots and understanding the limitations of what big-O notation can and can't tell you. You *can't* determine big-O runtime just by looking at single data points. You can't guarantee *with certainty* what the big-O runtime of a piece of code is just by seeing multiple data points, but you *can* make a good educated guess based on what you've seen. And when raw numbers are involved, your prediction about what algorithm to use on a given data set should be based on both the theoretical "which has a better big-O runtime?" and "what do the data actually show?"